UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

UNIVERSITY OF PRETORIA
Faculty of Engineering, Build and Information Technology

Department of Computer Science

A framework for cryptography algorithms on mobile devices

by

Johnny Li-Chang Lo

A dissertation submitted in partial fulfilment of the requirements for the degree of

Master of Science (Computer Science)

in the Faculty of Engineering, Built Environment and Information Technology University of Pretoria

2007

Acknowledgement

I would like to sincerely thank the following people for all their assistance during my studies:

- My parents, sister, grandmother and Paul for always believing in me and encouraging me to pursue a higher degree.

- Professor Judith Bishop who has always been a 'Mother Hen' towards my studies and provided me with all the guidance I needed.

- Dr. Stefan Gruner who read my first draft and provided me with the much-needed advice that helped me to improve this dissertation.

- The Techteam, especially Jaco Kroon who helped me to setup a server in the DMZ so that I was able to conduct my experiments.

- Mr. Morkel Theunissen who helped with the demonstration of the SMSSec protocol used in this study.

- Mrs. Sunette Steynberg for her friendly assistance with the renewing of library books that I have hold onto for most of my study duration and finding articles for me that is not readily available in the library.

- To my colleagues (past and present) in the Polelo Research Lab: Basil, May, Kathrin and John for your support.

- To all the developers of open-source cryptography packages. Your efforts really make the digital world so much safer without any monetary cost.

Lastly, many thanks to the National Research Foundation for the NRF Grant-Holder Bursary that funded the mobile devices I used in this study as well as for my travel expenses to conferences.

Abstract

A framework for cryptography algorithms on mobile devices

by

Johnny Li-Chang Lo

Supervisor: University-Professor Dr. Judith Bishop

Mobile communication devices have become a popular tool for gathering and disseminating information and data. With the evidence of the growth of wireless technology and a need for more flexible, customizable and better-optimised security schemes, it is evident that connection-based security such as HTTPS may not be sufficient. In order to provide sufficient security at the application layer, developers need access to a cryptography package. Such packages are available as third party mobile cryptographic toolkits or are supported natively on the mobile device. Typically mobile cryptographic packages have reduced their number of API methods to keep the package lightweight in size, but consequently making it quite complex to use. As a result developers could easily misuse a method which can weaken the entire security of a system without knowing it. Aside from the complexities in the API, mobile cryptography packages often do not apply sound cryptography within the implementation of the algorithms thus causing vulnerabilities in its utilization and initialization. Although FIPS 140-2 and CAPI suggest guidelines on how cryptographic algorithms should be implemented, they do not define the guidelines for implementing and using cryptography in a mobile environment. In our study, we do not define new cryptographic algorithms, instead, we investigate how sound cryptography can be applied *practically* in a mobile application environment and developed a framework called Linca (which stands for Logical Integration of Cryptographic Architectures) that can be used as a mobile cryptographic package to demonstrate our findings. The benefit that Linca has is that it hides the complexity of making incorrect cryptographic algorithm decisions, cryptographic algorithm initialization and utilization and key management, while maintaining a small size. Linca also applies sound cryptographic fundamentals internally within the framework, which radiates these benefits outwards at the API. Because Linca is a framework, certain architecture and design patterns are applied internally so that the cryptographic mechanisms and algorithms can be easily maintained. Linca showed better results when evaluated against two mobile cryptography API packages namely Bouncy Castle API and Secure and Trust Service API in terms of security and design. We demonstrate the applicability of Linca on using two realistic examples that cover securing network channels and on-device data.

Keywords: Cryptography, Mobile Devices, Protocol, Cryptographic Packages, Frameworks, Software Components, Protocol, Standards, Entropy, Small Message Service, Client, Server and Software Application.

Table of Contents - Chapters

Table of Contents – References and Appendixes

List of Tables

List of Figures

List of Equations

List of Works

Johnny Li-Chang Lo and Judith Bishop 2003. Component-based Interchangable Cryptographic Architecture for Securing Wireless Connectivity in Java Applications. In *Proceedings of the 2003 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, *SAICSIT*, Johannesburg, South Africa, 301-307.

Johnny Li-Chang Lo, Judith Bishop, and J.H.P. Eloff 2005. SMSSec: End-to-End SMS Protocol. Poster Presentation. *South African Institute of Computer Scientists and Information Technologists, SAICSIT* Whiteriver, South Africa. **(Best Poster Presentation)**

Johnny Li-Chang Lo, Judith Bishop, and J.H.P. Eloff 2006. SMSSec: an end-to-end protocol for secure SMS. Technical Report.

Chapter 1 - Introduction

"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it." --Marc Weiser

1.1 Background

The growth of wireless networks has brought vast changes in mobile devices, middleware-development, standards, network implementation and user acceptance [87]. Digital chip technology facilitated the development of smaller and lighter mobile devices. Wireless technology - in particular mobile cellular technology that supports packet data transmission - provides benefits to individuals and organizations by enabling them to connect to the internet from almost any place, at any time via a mobile device[1]. Some of the popular wireless mobile devices in use today include mobile phones, personal digital assistants (PDAs), pagers and laptops. There are approximately around 2 billion world cellular subscribers in April 2006 [5] and of those 2 billion there are 130 million 3G users. Given the size of the packet data mobile telephony market, a need arises for a secure means of transmitting, storing and authenticating data. Examples of these applications could be mobile banking, accessing corporate servers, online mobile stock trading and securing on-device data. Applications downloaded onto mobile devices need to be authenticated so that their origin can be confirmed and trusted. The mobile phone network infrastructure itself should also play a role in providing security support for the application at the network layer.

The most important automated tool for communications security is encryption [75]. Cryptography can be seen as a mathematical science that deals with the design of encryption and message authentication algorithms. There exist many different encryption algorithms but most can be grouped into one of two schemes: symmetric-key and asymmetric-key. However, encryption cannot *prevent* interception and data modification on communication channels. Encryption can only *protect* a communication channel against eavesdroppers extracting confidential data. For active attacks such as message modification, Message Authentication Code (MAC) is used to verify the integrity of the message. In order for the mobile telecommunication sector to thrive in the Information Communication Technology era, security is an important aspect.

1.1.1 Mobile Security

With the boom of the mobile internet, security mechanisms have to be in place. The three most important aspects of mobile security are data confidentiality, access control and on-device data security [93]. Data confidentiality means the protection of messages against eavesdroppers. In the context of network security, access control is the ability to limit and control the access to host systems and applications via communication links. In order to prevent nonauthorized attempts from accessing sensitive data stored on a device (for example financial data, passwords, private keys and so on), encryption should be in place to ensure those data are protected.

Connection-based security protocols such as HTTPS, WTLS and TLS are inadequate to cover these three aspects of mobile security. The idea behind connection-based security is to secure everything that passes through a communication channel. This approach has several limitations [30, 93]:

- **Direct connection between the client and a server must be established:** If an application has a number of different intermediaries that provide multiple value-added services, multiple HTTPS connections are piped together which could lead to potential security vulnerabilities at the connecting nodes. Managing public key certificates for each connection could be tedious.

- **All content is encrypted:** Connection-based security will not be required in some application scenarios such as broadcasting stock quotes or getting multilevel approval of a transaction because parts of the communication should be open. Instead, the verification of the authenticity of those quotes and approval signatures are more important. Encrypting all content unnecessarily introduces more processing overhead.

- **HTTPS is inflexible for applications that have special security and performance requirements:** HTTPS lacks support for custom handshake or key exchange mechanisms. For example, it does not require clients to *authenticate* themselves. Stronger security that requires 256-bit symmetric key encryption might not be guaranteed.

Mobile applications often interact with multiple backend servers, pull data from them as needed, and assemble personalized displays for the user. Each data service provider might have their own

[1] We refer “mobile device(s)” in this dissertation as “low-end small mobile devices” which include mobile phones and entry-level PDAs, unless otherwise stated.

user authentication and authorization system which requires a unique way for users to sign on. In such a case, HTTPS might not be necessary.

To cover the three aspects of mobile security, mobile developers need programmatic access to cryptographic algorithms. These algorithms are wrapped in a set of application programming interfaces (APIs) which give the developer the exposure to their functionality. In this way the API allows the developer to secure the data content rather than just the connection within a mobile environment. For securing the content, security mechanisms will often be applied at the application layer.

It is clear that the most popular types of mobile devices to operate on a cellular network are mobile phones and various PDAs. If the mobile device's operating system or platform does not support cryptography packages, a third party cryptographic toolkit[2] can be used. Examples of mobile cryptography packages supported on a particular platform/operating system include: Security and Trust Service API [81], Microsoft CAPI for Windows CE [43] and .NET Compact Framework. Examples of third party mobile cryptographic toolkits include: Bouncy Castle API [3], IAIK JCE-ME [24], Phaos Technology Micro Foundation Toolkit [58], NTRU jNeo for Java Toolkit [55].

1.2 Problems Concerning Cryptography Packages for Mobile Devices

Traditional cryptographic packages for desktops hide the plethora of algorithm-specific functions under a single set of API methods with often complex algorithm-selection criteria, in some cases requiring the setting of up to a dozen parameters to select the mode of operation [20]. As a result, developers tightly couple the application to the underlying cryptography implementation, thus violating the separation of concerns principle. In addition there will be intricate problems relating to how cryptography is applied within an application and the chances of misusing an API method by a developer are increased. Unfortunately, mobile devices are limited in resources such as CPU speed, memory size and persistent storage space; therefore many cryptographic packages created for mobile devices are sometimes a subset of the original cryptography packages for desktops. Mobile cryptography packages have *lightweight* API methods to keep the size small [3, 24, 55, 81] and as a

[2] A cryptography toolkit consists of packages that is bundled together and distributed by a third party software vendor. In this dissertation, we will collectively refer cryptography API supported on the mobile device's operating system/platform and cryptographic toolkits as *mobile cryptography packages*, unless otherwise stated. The reason for this reference is because the problems identified in our problem statement apply to both the cryptographic toolkits and the packages supported on the mobile device's operating system/platform.

result, mobile toolkits do not contain many useful methods to support the utilization and initialization of cryptographic algorithms.

Strong cryptographic algorithms are not always supported in the mobile cryptography package. For example, Windows CAPI for Pocket PC and Windows CE does not have AES and .NET Compact Framework 2.0 does not support SHA-256 [43, 44]. This will be a limitation for applications that require such algorithms. The lack of support of algorithms can also cause the unavailability of certain cryptographic functions. In the Secure and Trust Service API, there is no support for MAC generation. As a result, the developer has to fumble around the APIs and mimic the unavailable functionalities using other methods.

The complexity in API methods is related to how the cryptographic algorithms are *implemented within* the packages. Many of the mobile cryptography packages nowadays are implemented with the algorithms that have several initialization options that can be misused by developers. For example the default initialization vector can be initialized with a blank array of bytes which can be mistakenly reused [16, 79]. Another example is secret key generation using pseudorandom number generators supported in the package. Bouncy Castle API only seeds its pseudorandom number generator using system time, which seed can be easily recovered [74]. Assuming the external API is able to wrap any intricacies on applying cryptography from the developer, if any of the *internal modules and/or components* were poorly specified, interacted or implemented, the entire security system that uses this library would fail. Simply deploying a strong algorithm, such as AES, does not necessarily ensure security in a software product [32]. The bottom line is that implementing and applying cryptography is an intricate process that should not be approached in an ad-hoc fashion. There are *cryptographic fundamentals* that should not be accidentally overlooked. By these fundamentals, we mean the best practices in applying cryptography (irrespective of the environment) for:

- The initialization and utilization of cryptographic algorithms.
- The management of symmetric keys and asymmetric key pairs.
- Authenticating messages.
- Managing digital certificates.
- Generating and verifying digital signatures.

Therefore, mobile cryptographic packagess are more complex to use compared to their desktop counterparts because they are vulnerable on the *inside of the package*, where sound cryptographic

fundamentals are not often applied and *outside* at the API, where reduced methods give rise to complexity in their usage.

Developers should be aware of the consequences of the cryptographic functions they apply, but most importantly, the input data for these functions must be from a trusted source.

Consequently, a mobile cryptography package should meet the following requirement [93]:

- **Speed:** Mobile devices are personal devices that must be responsive. Handling CPU-intensive cryptography tasks, especially asymmetric key algorithms, at an acceptable speed is a big challenge.

- **Size:** Most modern, comprehensive cryptography packages consume several megabytes of storage space. A mobile device might have only 100KB of storage space; therefore the mobile cryptography package should balance features with footprint.

- **Comprehensive algorithm support:** A cryptography package's goal is to support flexible security schemes. Such flexibility comes from the ability to choose a range of algorithms. Symmetric and asymmetric key encryption, password-based encryption, and digital signature services should be supported.

- **Sensible API:** To support a wide range of algorithms through a consistent interface, cryptography package APIs often have multiple layers of abstractions and complex inheritance structures. However, a too complex API will hinder its adoption.

- **Easy key identification and serialization:** In a general-purpose cryptography package, keys for different algorithms must be identified and matched properly on both communication ends. The asymmetric key pair generation process is often too slow on mobile devices. Therefore, the key pairs are pre-generated on the server side and transported onto the device. The API should provide the means to ease and secure this process.

In the context of this research, we implemented a mobile cryptography package based on a framework called Linca[3] [37] that can be used to apply cryptography securely in a mobile application environment, while maintaining a simple API and efficient use of cryptography.

[3] Linca stands for Logical Integration of Cryptographic Architectures. A cryptographic algorithm can be incorporated into the framework in a logical way according to their functionality.

1.3 Linca Overview

Linca has one main core objective and that is to strive for the best combination amongst *security*, *simplicity* and *efficiency* in cryptography for the mobile application environment.

For security, Linca has:

- Strong cryptographic algorithms (such as AES and RSA) using 256-bit symmetric key and 2048-bit asymmetric key.
- Strong hash functions such as SHA-256 and HMAC for message authentication and verification.
- Support for symmetric key and asymmetric key pair management.
- Support for digital certificate and signature management.
- Internal algorithms and components, implemented and interacted with sound cryptographic fundamentals.

In order to maintain efficiency, attention is paid to:

- Key management.
- Choosing the best algorithm in terms of speed in encryption and decryption.
- Ensuring the ciphertext is kept to the same size as the plaintext for symmetric key encryption.
- How asymmetric key algorithm can be used efficiently in the mobile application environment.
- Minimizing the number of initializations for a particular cryptosystem.
- Keeping the entire framework to be as small as possible by reusing the internal components for other cryptographic functions.

Because Linca is a framework, it can be reused as a software component in other mobile applications.

Simplicity applies to the:

- Design of the framework, where cryptographic algorithms and internal components can be easily implemented and maintained for developers who implement the framework. Each cryptographic algorithms and security mechanisms within the framework are developed as a software component, thus the separation of concerns principle can be applied internally.

- External API, where developers using the framework can apply cryptography as simply as possible. A simpler interface definition for cryptographic algorithms can reduce the complexity of a system and increase system security. In this way, the developer can focus on the implementation of the security logic instead of the complexities in the utilization of cryptography. Because Linca is a software component, the internal cryptographic algorithms are loosely coupled with the application separated by the API.

Obviously, security is the top priority and it is not our intention to sacrifice security over simplicity and efficiency, however in this study, we show that it is possible for all three to coexist.

In order to cater for the different computing machines involved in the mobile application environment, Linca API has two implementations:

1. **Linca Mobile:** This implementation is targeted for the mobile device such as smart phones and PDAs.

2. **Linca Server:** This implementation is targeted for the server machine such as a desktop PC, where it is used to communicate with mobile devices.

The only differences between these two implementations are the types of networking supported, management of symmetric and asymmetric keys due to different hardware and software supported on the mobile device and server. The cryptography APIs between the two implementations remains the same to cater for a simpler application process. An extra set of connectivity APIs is supported for the Linca Mobile implementation to hide the complexities from the different types of connections supported by a mobile device. In this research, we will refer to Linca as the mobile implementation, unless otherwise stated.

1.4 Implementation Environment and Limitations

Linca is implemented in Java and executes on the Java 2 Micro Edition platform with the *Mobile Information Device Profil*e (J2ME/MIDP) [80]. MIDP contains API methods for implementing Java applications on a family of small mobile devices such as cellular phones and simple pagers that have the least resources available [30]. At the moment there are two version of MIDP implemented on mobile phones namely versions 1.0 and 2.0. MIDP 1.0 does not have cryptography support and connection-based security protocols. Security in MIDP 2.0 consists of HTTPS and code authentication support, however there is no cryptography support. MIDP 2.0 enabled devices

entered into the market towards the end of 2003. However there is still a mix of MIDP 1.0 and 2.0 phones in the market at the time of this writing.

The mobile devices that will be used for this research are MIDP 1.0 and 2.0 enabled Symbian OS [83] mobile phones connected over the General Packet Radio Service (GPRS) and 3G mobile networks.

Other operating system technologies considered but not used in this research include: Windows CE, Pocket PC 2003 [43], .NET Compact Framework[44], Qtopia [84] and Palm OS [57]. Therefore, the environment where Linca is evaluated in this study is limited to Java enabled mobile phones. Nevertheless, the cryptographic fundamentals applied in Linca are targeted towards any mobile networking environment characterized by high latency and low bandwidth.

A feature not implemented in Linca is the loading of private keys and digital certificates from smart cards, however we do propose support within the framework for such incorporation. What is implemented is a mechanism to load the digital certificate and private keys from a binary file.

1.5 Evaluation Criteria

Evaluation is done during the design and after Linca is implemented. There are three aspects to the evaluation:

1. **Efficiency.** Internal components such as encryption algorithms and symmetric key generators are evaluated during the design and implementation of Linca on an actual mobile device. The encryption algorithms are evaluated in terms of the encryption/decryption speed using different key sizes. Symmetric key encryption is evaluated using 128-bit and 256-bit keys, while asymmetric key encryption is evaluated using 2048-bit. The speed of the implemented symmetric key generating mechanism is evaluated as well.

2. **Security of the symmetric key generation.** The mechanism used in gathering the seed for seeding the key generator, as well as the random byte stream produced by the key generator is evaluated in terms of its randomness using random battery of tests found in the ENT [89] and DIEHARD [40] random testing tools. This evaluation is done during the design and implementation.

3. **Security improvement in applying cryptography.** The security and simplicity in applying cryptography using Linca API is evaluated against Bouncy Castle and Security and Trust Service API in terms of code illustration. Security improvements Linca has made over these two APIs are explicitly identified and explained in terms of the mistakes and vulnerabilities avoided when cryptography is applied.

4. **Design.** This part of the evaluation qualitatively measures the Linca framework in terms of package stability and abstraction metrics [41]. By using these metrics, the package design of the framework can be quantified. This is done after the design and implementation.

1.6 Differences from Other Works and Our Contributions

Linca is similar in concept to *cryptlib* [20], which focuses on the security of the internal object by assigning access control lists. Such techniques might be out of scope for the software supported on a mobile device. In our work, the security is focused on applying sound cryptographic fundamentals internally within the framework and externally at the API, thus giving the benefit of an inside-out approach to security.

FIPS 140-2 [52] and CAPI [47] have already defined guidelines on how cryptography modules should be implemented. However they do not define how cryptography can be applied and implemented in a mobile environment. In our study, we do not define new cryptographic algorithms; instead we developed a framework that ensures:

1. How sound cryptography can be applied *practically* in a mobile application environment.
2. The cryptographic mechanisms and algorithms implemented within the framework can be easily maintained on mobile devices.

In order to illustrate our findings, we implemented a mobile cryptography package based on the Linca framework that can be used as a standalone toolkit. We demonstrate the applicability of Linca using two realistic examples that covers securing network channels and on-device data.

1.7 Outline of the Dissertation

The following outlines the remaining chapters that forms parts of this research:

Chapter 2 – Cryptographic Fundamentals: This chapter presents a literature study on modern cryptographic algorithms. The mistakes that can lead to security vulnerabilities when applying cryptography are analysed. The fundamentals and recommendations on applying cryptography are presented, which will be used in the design and implementation of Linca.

Chapter 3 – An Overview of Mobile Cryptography Packages: This chapter presents an overview on two mobile cryptography toolkits namely Bouncy Castle API and Secure and Trust Service API. Security limitations of these two toolkits are also presented.

Chapter 4 – Linca Framework: In this chapter, we present a high-level architectural overview of Linca in the form of package structures and illustrate how some of these packages enable classes within them to conform into certain architectural styles and design patterns that will lead to a good design.

Chapter 5 – Security Design and Implementation: In this chapter, we present the application of the security fundamentals learnt from chapter 2 into the internal design and API of Linca. Evaluation of the encryption/decryption speed, key generating speed and the security of the key generating mechanism will be presented.

Chapter 6 – External API and Design Evaluation: Practical applications of cryptography using Linca, Bouncy Castle API and Secure and Trust Service API are critiqued using the cryptographic fundamentals studied from chapter 2. The design of these 3 cryptography packages is evaluated by using the package stability and abstraction metrics.

Chapter 7 – Application: We illustrate the applicability of Linca using two real world applications that secure a communication channel and an on-device data.

Chapter 8 – Conclusion and Future Work: In this chapter we identify the future work and conclude by mentioning the contributions made from this research.

Appendix A – Acronyms: A list of acronyms commonly used in this dissertation.

CD-ROM: The CD-ROM accompanying this dissertation contains the electronic format of dissertation, source code for Linca and applications, the Linca API and the statistical results evaluated from the DIEHARD battery of tests.

Chapter 2 - Cryptographic Fundamentals

"Security is a process not a product. If you think technology can solve your security problems, then you don't understand the problem and you don't understand the technology." --Bruce Schneier, Secret and Lies

The use of cryptography dates back to as early as the ancient Egyptians around 4000 years ago [92]. Cryptography was used a tool in both world wars and right until the 1960s where it was mainly used by governments and military to protect national secrets and strategies. Cryptography was not adopted worldwide until the boom of the internet in the early 1990s where cryptography has played an important role in securing the internet. Secure protocols like HTTPS and TLS are still the most widely used mechanisms in securing data communication over the internet. Cryptography is not only used in security protocols, it is also applied at the application level and the importance of applying it securely and effectively on mobile phones is already mentioned in chapter one. There are a number of *cryptographic tools (primitives)* used to provide data security [42]. Examples of primitives include encryption schemes, hash functions and digital signature schemes. Figure 2-1 provides a schematic listing of the primitives considered and how they relate.

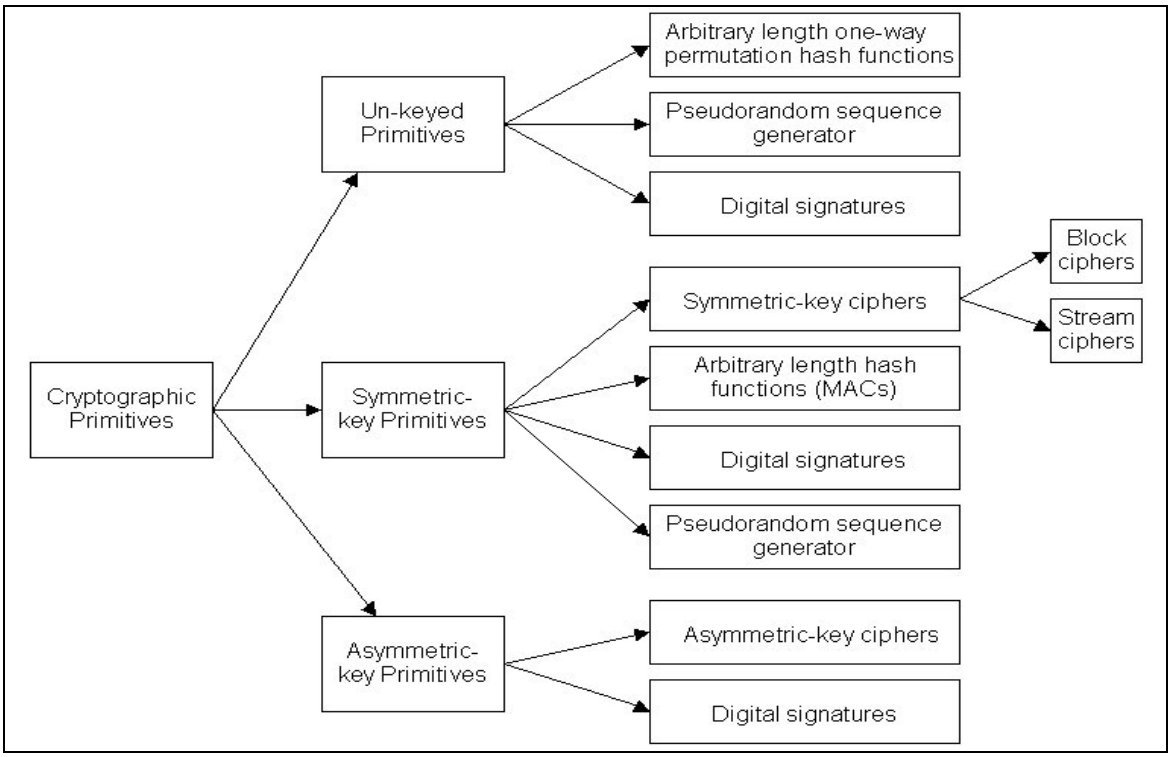


Figure 2-1: A taxonomy of cryptographic primitives (adapted from [42])

Cryptography is used to achieve the following goals:

- **Confidentiality:** To help protect a user's identity or data from being read.
- **Data integrity:** To help protect data from being altered.
- **Authentication:** To ensure that data originated from a particular party.

In this chapter we will present a literature study on the security issues relating to cryptography. By identifying these security issues, one can see that when cryptography is applied, it does not mean the system is totally secure, and by understanding these security issues, one can deduce the fundamentals that are required for applying *effective* and *secure* cryptography.

The **motivation** of this chapter is to present the fundamentals and recommendations in applying cryptography which will be practically used during the in design and implementation of Linca.

2.1 Operations and Symbols

There are a number of operations and symbols that will be used in this dissertation. For encryption and decryption operations, the following denotations are used:

- **$CIPH_K(P)$:** This denotes encryption operation on plaintext *P* using a secret key *K*.
- **$CIPH^{-1}_K(C)$:** This denotes decryption operation on ciphertext *C* using a secret key *K*.
- **$CIPH_{PUB\_A}(P)$:** This denotes encryption operation on message *M* using the public key belonging to entity *A*.
- **$CIPH^{-1}_{PRI\_A}(C)$:** This denotes decryption operation on ciphertext *C* using the private key belonging to entity *A*.

The most frequently used symbols are:

- **IV:** The initialization vector.
- **O_i:** The *i*th output block.
- **NC:** The nonce.
- **T:** The counter.

- ⊕ **:** This denotes the XOR operation.
- **||:** This denotes the concatenation operation. For example if X || Y then the result is XY.

2.2 Symmetric Key Cryptography

Symmetric key encryption was the only type of encryption scheme used prior to asymmetric-key cryptography in the late 1970s [75]. The encryption and decryption process in the symmetric key cipher depends on the same secret key shared between the sender and receiver. This means that the sender and receiver are required to agree upon the secret key before they can communicate securely.

Symmetric key ciphers can be divided into two categories namely *stream* and *block* ciphers. Stream ciphers operate on a plaintext, 1 bit at a time. Block ciphers operate on a fixed size group of bits within the plaintext and are the most widely used type of symmetric cipher. These group of bits are called a *block* and the current generation of block ciphers has a block size of 128-bits (16 bytes).

In this section, we present a study on block ciphers, focusing primarily on the Advanced Encryption Standard (AES) because it is the latest cipher standard set by the U.S government for the encryption and decryption of unclassified sensitive information. Security considerations for using symmetric key cryptography are also discussed.

2.2.1 Advanced Encryption Standard (AES)

In 1997, NIST asked for proposals by means of a competition from the cryptographic community for ciphers that would best fit the AES criteria. NIST specifies that AES must be a symmetric block cipher with a block size of 128-bits (16 bytes) and support for key lengths of 128, 192 and 256-bits. The evaluation criteria include security, computational efficiency, memory requirements, hardware and software suitability and flexibility [75]. A total of 15 proposals were submitted and five were selected as finalists and eventually the *Rijndael* cipher [8] was selected to become the AES in 2001. Rijndael was named after its creators Joan Daemen and Vincent Rijmen. The other finalists were Serpent [1], Twofish [71], RC6 [62] and MARS [4]. The AES does not use the Feistel construct however it still uses the mathematical operations such as substitutions, permutations, XORes and it has multiple rounds [13]. The subsequent rounds are similar and the number of rounds (varying from 10 –14 rounds) depends on the key size. The AES operates on a byte level thus allowing efficient implementation on both hardware and software environments.

For one round operation of the AES, the first operation is to XOR the input plaintext of 16 bytes with the round key. The next process is the byte sub step where each of the 16 bytes is then used as an index into an S-box table that maps 8-bit inputs to 8-bit outputs. The S-boxes are all identical. The third and fourth processes are the shift row and mix column where the bytes are then rearranged in 4 groups at 4 bytes each to be mixed within a liner mixing function (see Figure 2-2). The term linear means that each output bit of the mixing function is the XOR of several of the input bits [13].

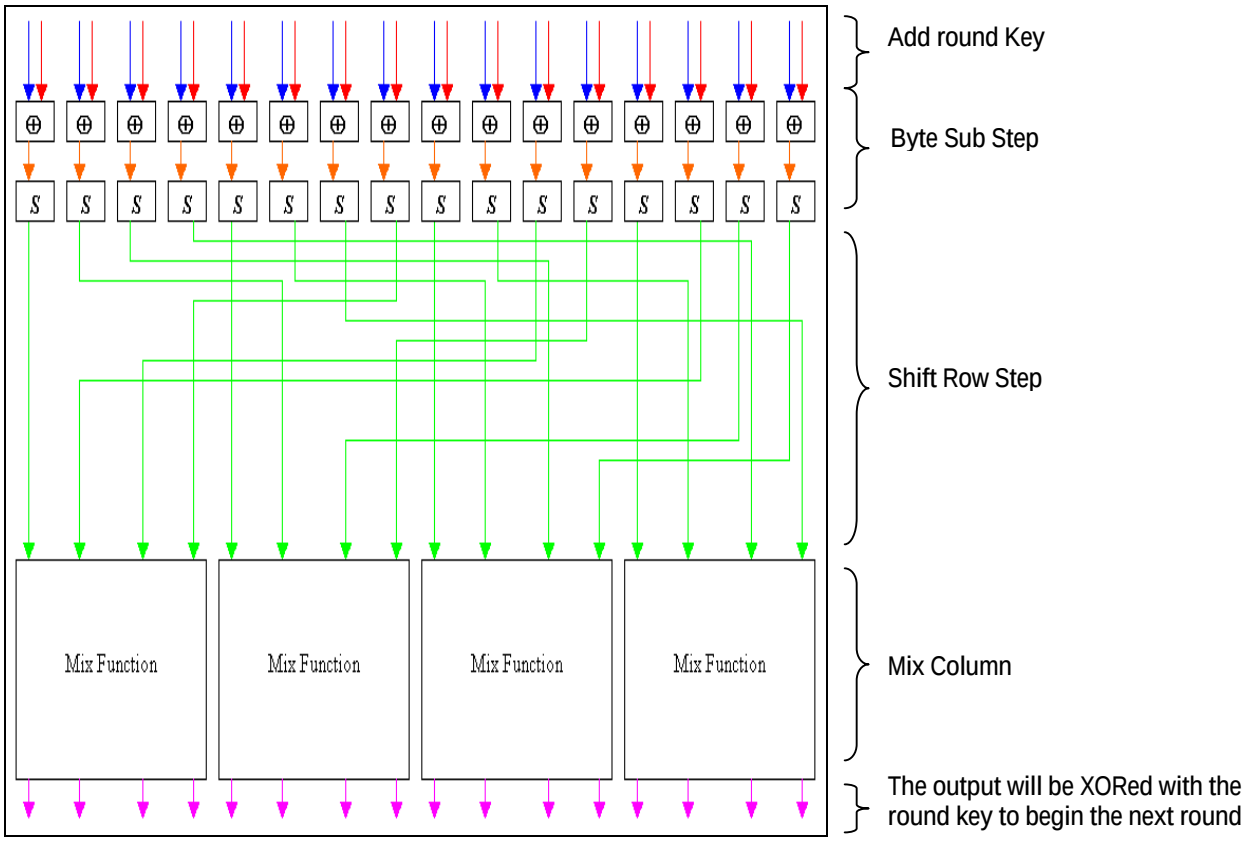


Figure 2-2: A structure of a single round of AES (adapted from[13])

The AES designers demonstrated an attack on 6 rounds and chose 10-14 rounds for the full cipher, depending on the key size. During the selection process, the attacks were improved to handle 7 rounds for 128-bit keys, 8 rounds for 192-bit keys and even 9 rounds for 256-bit keys. This leaves 3 to 5 rounds of security margin. Therefore according to these figures the most effective attack possible for 128-bit keys covers 70% of the cipher.

Another concern about AES is that the encryption function can be represented in a simple algebraic structure. The impact of this could lead to a new mathematical study on recovering the secret key.

In 2002, Courtois [7] found a way to attack the AES using a technique called XSL. At the moment, this attack is still theoretical since the amount of work required exceeds the computation capability offered by today's technologies. However, it does not mean that this attack cannot be improved in the future.

2.2.2 Other AES Finalists not Supported in Linca

During the AES competition, both Serpent and Twofish had a slower performance compared to Rijndael. RC6 and MARS have some uncertainties regarding their security and efficiency. RC6 was broken within 17 rounds out of 20 during the AES competition. MARS was too computationally expensive to implement compared to the other AES finalists. Also, MARS contained a serious software bug that didn't generate the correct S-box according to the original set the MARS team had and the cipher failed to prove it is resistant to linear cryptanalysis.

2.2.3 DES

The preceding standard to AES is the Data Encryption Standard (DES). DES was developed by in the 1970s as a US government standard for protecting non-classified information and was published as a FIPS PUB 46-3 [49]. The reason we are studying DES is because it is still supported in various cryptographic toolkits supported on mobile devices and smart card operating systems [43, 81, 88].

DES is a 64-bit block cipher that uses 56-bit keys in length and has a base design according to the Feistel network.

The possibility of a brute force attack on finding a DES key is realised through a "DES Cracker"[4] built by the Electronic Frontier Foundation (EFF) in June 1998 where it found a DES key in less than three days [75]. Due to successful attacks on DES, it is no longer approved for Federal use since 19th of May 2005 [51].

2.2.4 Security Considerations

In this section, we present some security considerations when using symmetric key cryptography. These considerations contain important principles that form the basic criteria for implementing

[4] The DES cracker machine costs less than $250000 to built. Nowadays, technology has become cheaper and faster therefore cracking DES becomes highly possible.

Linca securely. These *fundamentals* are a list of guidelines which we feel are critical to consider (in terms of security and efficiency) when cryptography is applied. The *recommendations* are the *cryptographic algorithms* that are justified for support in Linca based on the fundamentals.

Cipher Choice and Key Initialization

From our literature study, we can see there are various attack methods used by cryptanalysts to try and find vulnerabilities in block ciphers. Collision problems[5] are more prominent on a 64-bit block ciphers compared to 128-bit block ciphers.

Fundamental 1: Use 128-bit block ciphers

There are still security issues amongst 128-bit block ciphers (see sections 2.2.1 and 2.2.2), however the AES is the new federal U.S. standard for data protection and keeping to standards is recommended for Linca. Another factor is that AES was evaluated to be the most efficient on software and hardware environments, thus making it suitable on mobile devices as well.

Recommendation 1: Use AES cipher for symmetric encryption/decryption

Symmetric key utilization is not the same for every cipher and some certain initialization characteristics can break the security offered by the cipher. For example, DES has a weak and semi-weak key problem that is defined by $CIPH_K(CIPH_K(P)) = P$ and $CIPH_{K1}(CIPH_{K2}(P)) = P$ respectively. DES has 4 weak keys and 16 semi-weak keys [42]. Due to this problem in DES, the developer is required to do a test to see whether or not the current key used is a weak key.

Another symmetric cipher called RC4[6] [70] has a requirement that the same RC4 key should never be used to encrypt two different data streams because the output of the generator is XORed with the data stream[13, 88]. This requirement is not met in the wired equivalent privacy (WEP) standard which uses RC4 to encrypt the wireless communications on 802.11 networks [88]. The problem with WEP is the key structure. Most of the WEP products implement a 64-bit shared key, 40-bits of

[5] If an element can take on N different values, then a fist collision would be expected after choosing about $\sqrt{N}$ random elements. This is known as the ***birthday paradox***.
[6] RC4 is a stream cipher designed by RSA Data Security, Inc. It is essentially designed to be used as PRNG but has been adopted to be used as an encryption algorithm.

which are used for the secret key and 24-bit initialization vector (IV). WEP didn't allocate enough bits for the IV, with the result that the same IV value had to be reused, which in turn results in many packets being encrypted with the same key.

> Fundamental 2: Symmetric key utilization is not the same for every cipher

Key Management

Effective key management spans over key generation, distribution and storage. The security of the symmetric key cipher rests upon an effective key management of the secret key because if the key is compromised the consequence is disastrous.

Before a key is used or distributed it needs to be generated first. A good symmetric key should contain a random string of bits that does not invite any mathematical attacks [70]. It is not advised to use traditional random number generators from various programming libraries because the random data generated can be predictable. These predications can be measured according to statistical analysis [13, 70]. In order to generate random bytes, a cryptographically strong pseudo random number generator (PRNG[7]) should be used. A PRNG can come in handy in generating symmetric keys [5]. A PRNG contains a deterministic algorithm that accepts a random seed for generating an output that cannot be easily exploited by any mathematical analysis. This is achieved by applying cryptographic techniques such as hash functions and/or block ciphers with a relevant mode that mixes these bits. A PRNG produces the same output if the seed that is used to produce that output is reused again. However, even if this is the case, the output should always be non-deterministic for that particular generation by making sure that the adversary cannot predict the seed. If the seed is predictable, an attack in acquiring the internal state of the PRNG becomes more feasible. Once the internal state is acquired, the attacker is able to determine the output of the PRNG.

> Fundamental 3: A cryptographically strong PRNG should use real random data to seed itself and that the internal state of the PRNG is difficult to acquire by an attacker

The seed can be collected from various event sources [26]. The best source of input for mixing would be hardware randomness such as disk drive timing affected by air turbulence, audio input with thermal noise, or radioactive decay. If hardware random sources are not available software random sources can be used. These include system clocks, system or IO buffers, network serial numbers and/or addresses, and user input.

In [74] the author mentioned that one could obtain random numbers from a third party source, over an internet link. One problem with this approach is that all network traffic may be eavesdropped upon, so it should be encrypted, possibly leading to a bootstrap problem. Another problem is that the third party source must be trusted.

If the generator does not gather enough entropy[8] for the seed, randomness in the output cannot be guaranteed. For example, Goldberg and Wagner found a problem with the way Netscape was seeding its SSL V2 protocol [17]. The random events used were the time of the day, process ID and the parent process ID all mixed by a hash function to produce the random output. It was found that these three inputs were not really random at all and at best only 47-bits of entropies were gathered, which allowed Goldberg and Wagner to compromise a real SSL session in 25 seconds by decreasing the number of entropies. The reference implementation of HTTPS for MIDP 2.0 phones, uses only the system time (`System.currentTimeMillis()`) for its seed update [9].

However estimating the amount of entropy collected is difficult if not impossible to do it correctly, therefore a PRNG called Fortuna [13] solves this problem by collecting random sources from various events into 32 pools of strings of unbounded length. Each source distributes its random events over the pools in a cyclical fashion to ensure that the entropy from each source is distributed more or less evenly over the pools and making sure the attacker cannot determine the pooled data.
An effective way to attack the PRNG is to find a flaw within the code or the random source that allows the attacker to recover the internal state. The internal state generally means the output generated from random sources when mixed with a hash function. There are many types of PRNGs namely Yarrow [13], X9.17, DoD key generation [70] to name a few.

[7] Further use of the PRNG acronym in this dissertation is referred to as cryptographically strong PRNG unless otherwise stated.
[8] In the context of cryptography, entropy means the measurement of randomness.

An effective key distribution strategy can be made by means of a key distribution centre [75] and asymmetric key cryptography. However with the latter, care should be given on how the protocol is derived.

Symmetric keys are not recommended to be stored persistently on mobile devices where there is a greater chance of it being stolen. Also the longer the keys are kept the easier it is for cryptanalysis. Depending on the situation, if the key is used for communication between two parties over a network, then it is recommended to use a session key because it limits the chance of an attacker in deriving the correct key. If a key is required to encrypt a file locally on the machine for example, then password-generated keys (PBE) may be used [64] since the key is generated only when the user types in the password (assuming the password itself is not compromised).

> Fundamental 4: Use session keys for network communication and PBE keys for on-device data protection

Key Size

A common way to attack a symmetric key is to apply brute-force for all possible combinations that will lead to deriving the key. Table 2-1 presents the average time required for an exhaustive key search. The units are measured in µs (microsecond). For each key size, the results are shown assuming that it takes 1 µs to perform a single decryption on a processor using 1 key. By using parallel organizations of microprocessors, it is possible to decrypt at greater magnitudes, for example using 1 million keys per microsecond.

Key size (bits)	Number of Alternative keys	Time Required at 1 Decryption/µs	Time Required at 10^6 Decryption/µs
32	$2^{32} = 4.3 * 10^9$	35.8 minutes	2.15 millisecond
56	$2^{56} = 7.2 * 10^{16}$	1142 years	10 hours
128	$2^{128} = 3.4 * 10^{38}$	$5.4 * 10^{24}$ years	$5.4 * 10^{18}$ years
168	$2^{168} = 3.7 * 10^{50}$	$5.9 * 10^{36}$ years	$5.9 * 10^{30}$ years
192	$2^{192} = 6.3 * 10^{57}$	$9.95 * 10^{43}$ years	$9.95 * 10^{37}$ years
256	$2^{256} = 1.2 * 10^{77}$	$1.84 * 10^{63}$ years	$1.84 * 10^{57}$ years

Table 2-1: Average time required for exhaustive key search (adapted from [8, 75])

There are many applications today that are using 128-bit keys. However 128-bit key can give rise to collision attacks such as meet-in-the-middle and birthday. According to [13] most of the block

ciphers allow meet-in-the-middle attacks of some form. In order to reach a 128-bit security design, a *256-bit key* is required. In other words, a system needs to withstand attackers that can perform 2^{128} operations in their attack. To give a better perspective of this concept, a 128-bit key would result in a 64-bit security according to collision attack and 256-bit would result in 128-bit security using the same attack. There are three reasons that might hinder the developer to use 256-bit key size namely performance consideration, cipher design and law restriction.

The key size has an impact on performance for certain block ciphers. For example AES would run slower for a 256-bit key compared to 128-bit while Serpent would remain at the same speed for either key size. Twofish is claimed to have a slower key setup time however encryption and decryption speed remain the same [72]. Block ciphers such as DES and Triple-DES are designed to only support key size of up to 56-bits and 168-bits respectively.

Mobile device applications usually send small amount of data and due to the cryptic nature of smaller messages, it facilitates easier cryptanalytic attempts based on word frequencies.

Fundamental 5: Choose the key size that will give 128-bit security after collision attacks

Will the encryption/decryption speed be too slow when 256-bit key is used on a mobile device? This question will be answered with an experiment conducted in section 5.1.1.

The law can influence the choice of the key size (see section 2.8). For example the U.S. export regulation mandates key size restrictions when exporting security products to other countries. In the context of the research, we do not take the laws governing the usage of the key size into consideration because they do not relate to providing good security principles in cryptography.

2.3 Block Cipher Mode

A block cipher only encrypts and decrypts a fixed block of data. In order to encrypt and decrypt plaintext that is larger than the block size, the block cipher needs to work in conjunction with a block cipher mode. This section primarily focuses on two block cipher modes namely: Counter (CTR) and Cipher Block Chaining (CBC) block cipher modes. These two modes are defined in [48]. The reason why we chose to study these two modes for this research is because they are much

superior in security and efficiency compared to other NIST approved block cipher modes (see section 2.3.3). A comparison between CTR and CBC is presented in section 2.3.4 so that the best mode to use within the mobile application environment can be identified. Security considerations in using a block cipher mode are also presented.

2.3.1 Counter Mode (CTR)

In the counter mode (CTR), the plaintext block *j* is XORed with an output block O_i, which is generated by encrypting a counter *T* with the key *K*, where *j* is the block number out of a total number of *n* blocks per message. The *O* forms the keystream value. For the last block of the message, O_n is XORed with the most significant bits of P_n and the rest of the bits are discarded. The CTR mode turns the underlying block cipher into a stream cipher (see Figure 2-3). This process is defined by the following [48]:

For encryption:
$O_j = CIPH_K(T_j)$ for j = 1….*n* (where *n* is the number of blocks)
$C_i = P_j \oplus O_j$

For decryption:
$O_j = CIPH_K(T_j)$ for j = 1….*n* (where *n* is the number of blocks)
$P_j = C_j \oplus O_j$

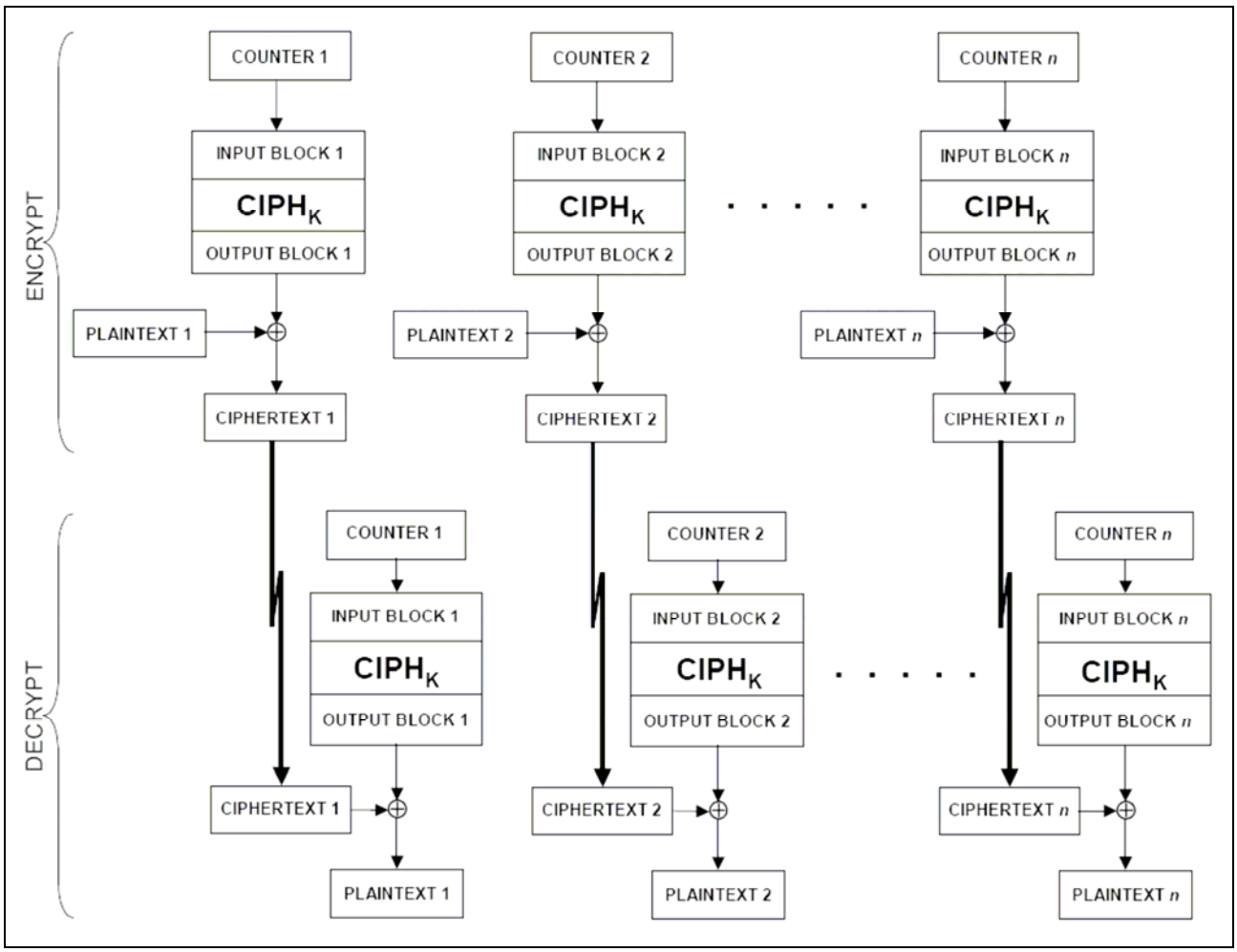


Figure 2-3: Counter mode (adopted from [48])

Counter Generation

A well-designed standard for CTR mode should not be overly prescriptive about how *T* is formed. The most important aspect related to *T* is that it must not be reused [13, 36]. The size of *T* should be as large as the block size. A typical setting for a 128 block cipher would be: T_j = NC || j, where *NC* could consist of 48-bit message number, 16-bits of nonce data, and 64-bits for *j* [13]. In this way, the system is limited to encrypting 2^{48} different messages using a single key and limits each message to 2^{68} bytes.

Efficiency

For CTR, the speed is around the same as a block cipher. CTR can do preprocessing and can take advantage of hardware parallelizing due to the fact that encryption process is independent of the plaintext [36]. The encryption and decryption processes are dependent on the encryption function

only, thus promoting simplicity during implementation. The ciphertext generated would be the same size as the plaintext.

Fault Tolerance

If a bit-flip error occurs in some block of cipertext, after decryption, the error is localized to the corresponding block of plaintext. CTR mode does not cause collision problems since no ciphertext blocks are equal [13].

Security Concerns

The nonce within the counter should never be reused. If it were reused, the counter would not be unique and it would result in an identical keystream value (assuming the key is the same for the entire encryption process). Such an incident would leak information about the entire message.

The security of the CTR mode depends on the strength of the underlying cipher. If the cipher is weak (eg DES), then it is easier to perform differential cryptanalysis.

2.3.2 Cipher Block Chaining (CBC)

In order to produce the first block of ciphertext, an IV is XORed with the first block of plaintext. This input is fed into the encryption cipher and encrypted with a secret key. The subsequent plaintext blocks are XORed with the preceding ciphertext block thus creating a chain. There are two fundamental requirements [13, 75] for CBC mode. The first requirement is that the plaintext bits need to be a multiple of the block size; otherwise this plaintext needs to be formatted by appending extra bits to achieve the desired length. The second requirement is that the IV needs to be unique for each encryption process. If the second requirement is not met, then the pattern of identical first block messages might be discovered between two messages.

This is illustrated by the following: P_1 and P_2 are both plaintext representing some account numbers. C_1 and C_2 are the ciphertext generated after encrypting with the same IV and key. The highlighted parts represent the identical first block of data.

The two input strings:

P_1 = Account:776798949093 7

P_2 = Account:776798944560 1

The sample output encrypted with a 128-bit key using AES/CBC/PKCS7Padding and the same IV:

C_1 = b1263af3d28a8de8b751462420b25a1ed8d3d7a7845d7acd9678c6a6d6971cd

C_2 = b1263af3d28a8de8b751462420b25a1e1bf05c1a7dc6204208c7889fcab3cacb

2.3.3 Other Cipher Modes

There are other block cipher modes in use today namely: ECB, OFB, and CFB. These block cipher modes are inferior compared to CBC and CTR in terms of security and efficiency [13, 75]. In ECB, if two identical plaintext blocks are encrypted using the same key then the resulting ciphertexts are identical. In OFB, a certain key block within the keystream can repeat itself. If this happens, some message blocks will be XORed with the same key block. CFB is a stream cipher mode similar to CTR however it is more complex to implement and not as efficient. A newer mode called OCB has been proposed over the last few years that offers both authentication and encryption. However the problems with OCB are the lack of confidence in the security proof, and that the patent and licensing status of newer modes are not fully understood.

2.3.4 Security Considerations

Encryption modes stop an eavesdropper from reading the network traffic: they do not provide any authentication. Therefore an attacker can still change the message in some way. The decryption function of an encryption mode just decrypts the data and although it might produce nonsensical data, it still decrypts a modified ciphertext to some (modified and possibly nonsensical) plaintext. In almost all situations the damage that modified messages can do is far greater than the damage of leaking the plaintext [13]. For example, suppose a modified ciphertext is decrypted and the output is processed by a software component. If somehow a portion of the output is validated correctly by the component, then the entire message might be treated as valid. In such a situation, it is recommended to produce a MAC of the ciphertext and append that MAC alongside the ciphertext.

Fundamental 6: A MAC should be computed for the ciphertext produced by the encryption mode

A counter and IV share the same security principle and there are various ways to create an IV, however choosing the securest method can be overlooked. Some of these methods are:

- **Fixed IV:** In a fixed IV, the IV is the same value used throughout all of the encryptions. This leads to the problem discussed earlier about encrypting identical first message blocks between two messages.

- **Counter IV:** The idea behind counter IV is that the IV is incremented by 1 after each encryption. For example, IV = 0 is used for the first message, IV = 1 for the second, IV = 2 for the third and so on. The problem here is that the values of each bit may not be enough to change the two identical first block of two messages during XOR.

- **Random IV:** This is a random string generated by a random number generator. There are two disadvantages. The first is that the encryption algorithm must have access to a source of randomness. This would resort to implementing a good random number generator, a task too complicated yet trying to avoid. The second is problem is that the recipient needs to know the IV, therefore the IV is send as the first block before the rest of the ciphertext. The result of this is that the whole message is one block larger, thus adding an overhead if the message is small.

- **Nonce-generated IV:** The best way of generating an IV using a nonce [13]. Typically, the nonce is a message number of some sort, possibly combined with some other information. It appears there are no specific ways in specifying what information to use. For example, the information could be a random string appended with the message direction or it could be some shared knowledge between the sender and receiver. The message numbers are used to keep messages in the correct order and to detect duplicate messages used for reply attacks. The nonce used here should be unique for each encryption used in bidirectional traffic between a sender and receiver. Once the nonce is composed, it is encrypted with the block cipher to produce the IV.

Although the IV does not have to be secretive [70, 75], it still poses a threat because an attacker can fool the receiver into using a different value for the IV. If this happens the opponent is able to invert selected bits in the first block of plaintext. The concept of the nonce generated IV is suitable for generating the counter.

Fundamental 7: Use a nonce-generated IV

2.3.5 CTR versus CBC

The choice for using CTR over CBC for mobile device applications can be governed by the following comparison between CTR and CBC illustrated in Table 2-2 [13]:

Mode	Padding	Speed	Implementation	Robustness	Nonce
CTR	No padding required therefore reduces traffic volume.	Can parallelize the computations arbitrarily, therefore allowing implementations to reach higher speed.	Requires the block cipher encryption function. In this way, it reduces the code size.	CBC might leak some information about the initial plaintext block if nonce is reused.	CBC can use a random IV or a nonce.
CBC	Padding required.	Cannot parallelize the computations.	Requires both the encryption and decryption function to be implemented.	CTR will leak information about the entire message if nonce is reused.	CTR will leak information about the entire message.

Table 2-2: Comparisons between CTR and CBC

From these comparison points, it is clear that CTR is superior in every respect except robustness. However, if the uniqueness of the nonce used in CTR is implemented CTR will leak very little information. Therefore it is important to design a unique nonce-generating mechanism.

Recommendation 2: Use CTR block cipher mode

2.4 Asymmetric Key Cipher

Asymmetric key cryptography[9] was introduced by Diffie and Hellman in 1976 [10]. Contrary to symmetric key cryptography, asymmetric key cryptography involves the use of two separate keys namely public and private-keys and the algorithms are based on mathematical functions such as modulo and logarithmic arithmetic instead of operations on bit patterns. The use of the two keys has profound consequences in three areas of areas of applications namely: confidentiality, authentication and key distribution. However, not all of the asymmetric key algorithms can perform all three applications. Table 2-3 summarises the applications of various asymmetric key cryptographic algorithms. From Table 2-3, we can see that the RSA and Elliptic Curve ciphers are the most versatile. In this section we will focus on the RSA cryptosystem and mention the reason for recommending this cipher. Security considerations in applying asymmetric key cryptography are also discussed.

Algorithm	Encryption/Decryption	Digital Signature	Key Exchange
RSA	Yes	Yes	Yes
Diffie-Hellman	No	No	Yes
DSA	No	Yes	No
Elliptic Curve	Yes	Yes	Yes

Table 2-3: Application of asymmetric key cryptosystem (adopted from [75])

2.4.1 Requirements of Asymmetric Key Cryptography

According to [10] an asymmetric key cryptosystem should fulfil the following conditions regarding encryption and decryption:

1. It is computationally easy for an entity B to generate a pair of public and private keys (*PUB_B* and *PRI_B*).

2. It is computationally easy for a sender A, knowing the public key and the message to be encrypted, *P*, to generate the corresponding ciphertext:

$$C = CIPH_{PUB\_B}(P)$$

[9] Asymmetric key cryptography is also synonymously known as public-key cryptography; therefore these two terms are used interchangeably throughout this dissertation.

3. It is computationally easy for the receiver B to decrypt the resulting ciphertext C using the private key to recover the original message:

$$P = CIPH^{-1}_{PRI\_B}(C)$$

4. It is computationally infeasible for an attacker, knowing the public key *PUB_B* to determine the private key, *PRI_B*.

5. It is computationally infeasible for an attacker, knowing *PUB_B* and C to recover *M*.

2.4.2 RSA Cryptosystem

One of the first public-key cryptosystems ever introduced was developed in 1977 by Ron Rivest, Adi Shamir and Len Adleman at MIT and first published in 1978 [61]. The RSA system is probably the most widely used public-key cryptosystem in the world. It is also very versatile as it provides data confidentiality, digital signatures and key exchange (see Table 2-3). The strength of RSA derives from factoring large numbers [70]. The RSA algorithm is a block cipher in which the plaintext and ciphertext are integers between 0 and $n - 1$ for some n. The simpler description of the RSA algorithm is as follows [70, 75]:

Public-key components:

n = product of two large primes namely p and q, where p and q remain secretive
e = a random number relatively prime to and less than $(p - 1)(q - 1)$

Private-key components:

$d = e^{-1} \bmod ((p - 1)(q - 1))$ the multiplicative inverse of e mod $((p - 1)(q - 1))$

Encryption scheme (RSAEP):

$C = P^e \bmod n$

Decryption scheme (RSADP):

$P = C^d \bmod n$

Digital signature:

$S = P^d \bmod n$
$P = S^e \bmod n = P^{ed} \bmod n$ (to verify the signature)

For RSA to be satisfactory for public-key encryption, the following requirements must be met:

1. The two large primes p and q must remain secretive.
2. It is possible to find values of e, d, n such that $M^{ed} = M \bmod n$ for all $M < n$.
3. It is relatively easy to calculate M^e and C for all values of $M < n$.
4. It is infeasible to determine d given e and n.

Efficiency

Asymmetric key cryptosystem involves mathematical functions that would require more computations compared to symmetric key cryptography. RSA encryption and signature verification are faster if a low value for e is used; however that would be insecure [13].

The recommendation to use the Chinese Remainder Theorem (CRT) for the private key structure can speed up the decryption speed on a single processor. By using the CRT, a factor of 3 to 4 in computing time can be saved in a typical implementation, while the downside is the additional software complexity and the necessary conversions [13]. This speed improvement will be beneficial if RSA is used on a mobile device where CPU speed is limited. The RSA specification using the CRT can be found in [66].

Fundamental 8: Use CRT to speed up decryption speed

Security Issues

Factoring *n*: The best way of resolving the private key is to factorize n to derive p and q in order to recover the rest of the private key's components. As technology and hardware improve, factoring large numbers is becoming easier than it used to be back in the early 70s. Currently 640-decimal-digits is at the edge of factoring technology [67]. Another alternative to factoring n is to determine $(p - 1)(q - 1)$ however the amount of work is the same as factoring n.

IND-CCA insecure: An encryption scheme that is semantically secure under *adaptive* chosen ciphertext attack is said to be IND-CCA secure [66]. Adaptive chosen plaintext attack is a chosen ciphertext attack applied with an extra condition that the attacker is unable to exploit a decryption box that would decrypt for some random challenge ciphertext. The problem with RSA encryption primitives is that it is deterministic therefore failing the IND-CCA security. To alleviate this problem, an encoding function is required to destroy any message structure [59]. There are two encoding functions defined in PKCS #1 v2.1 namely: RSAES-PKCS1-v1.5 and RSAES-OAEP

with RSAES-OAEP becoming the *de facto* standard (RSAES-PKCS-v2.0) due to security problems in RSAES-PKCS1-v1.5 [66]. We will present RSAES-OAEP encoding later in section 2.6.3.

Common modulus attack: A possible RSA implementation gives everyone the same *n*, but different values for *e* and *d*. The problem is that if the same message is ever encrypted with two different exponents (both having the same modulus) and those two exponents are relatively prime then the plaintext can be recovered without either of the decryption exponents [70].

Encrypting small plaintext: If *P* is to be encrypted and *P* is smaller than *n* (in terms of number of bits) then there is a chance that modulo reduction never took place [13]. For example if $e = 5$ and $P < \sqrt[5]{n}$, then $P^e = P^5 < n$. The attacker can recover *P* by simply taking the fifth root of P^5. A practical scenario is if the user encrypts a 256-bit secret key using RSA, then the encrypted key is less than $2^{256*5} = 2^{1280}$ which is smaller than 2^{2048} assuming *n* is 2048-bits.

Low decryption exponent attack: An attack proposed by Wiener [91] states that if *d* is approximately one quarter the size of the modulus *n* and e is less than *n*, then *d* could be recovered. This attack poses no threat if *d* is approximately the same size as *n*.

2.4.3 RSAES-OAEP

The RSAES-OAEP [66] combines the RSAEP and RSADP primitives with the EME-OAEP encoding method. EME–OAEP is based on Bellare and Rogaway's Optimal Asymmetric Encryption scheme [2] and is compatible with the IFES scheme defined in IEEE Std 1363-2000, where the encryption and decryption primitives are IFEP-RSA and IFDP-RSA and the message encoding scheme is EME-OAEP (OAEP stands for "Optimal Asymmetric Encryption Padding").

Message Length

RSAES-OAEP can operate on messages of length up to $k - 2hlen - 2$ octets, where *hlen* is the length of the output of the underlying hash function and *k* is the length in octets of the recipient's RSA modulus. This is a limitation to the message size that could be sent. For example, if *k* is 1024-bits and *hlen* is 160-bits, then the message size is limited to 87-bytes instead of 128-bytes.

Security Issues

Shoup [73] showed that RSA-OAEP does not guarantee security for key sizes used in practice (for example 1024 and 2048-bits) due to the inefficiency of the security reduction. The consequence to this is that the possibility of an easier attack on RSAES-OAEP compared to factorising *n* cannot be excluded. The reduction referred in [73] shows that a 1024-bit modulus just provides a provable security level of 2^{40}, which is clearly inadequate given currently prevalent levels of computing power. It is noted that this does not mean that there is an attack with this low complexity, however one cannot be ruled out by the available proofs of security. Nevertheless PKCS#1 v2.1 mentions that RSAES-OAEP construction is sounder than the *ad hoc* construction of RSAES-PKCS-v1_5 [66], which is another encoding standard defined in the PKCS#1 v2.1.

2.4.4 Other Asymmetric key Algorithms

There are other well-known asymmetric algorithms used in commercial products in conjunction with RSA, however these algorithms have their limitations. These algorithms are Diffie-Hellman[10], ElGamal [11], Digital Signature Algorithm [50] and Elliptic Curve Cryptography [45].

The problem with Diffie-Hellman algorithm (DH) is that it is not as versatile as RSA and key generation for the DH protocol might be too computationally expensive for the mobile device.

ElGamal is as versatile as RSA, however the ciphertext generated is twice the size as the plaintext; therefore it does not favour well in an environment with high latency and low bandwidth.

Digital Signature Algorithm (DSA) is not as versatile as RSA and is slower than RSA in terms of signature verification [70]. Another problem with DSA is that the key size varies from 512 to 1024-bits, therefore requiring a stronger key size beyond 1024-bits is not possible.

The core benefit of Elliptic Curve Cryptography (ECC) compared to RSA is that it appears to offer equal security for a far smaller key size, thereby reducing processing overhead [86]. However various aspects of ECC have been patented by a variety of people and companies around the world, especially the encryption function. Notably, the Canadian company, Certicom Inc. holds over 130 patents related to elliptic curves and public key cryptography in general [46].

Recommendation 3: Use RSA cipher for asymmetric encryption/decryption

2.4.5 Security Considerations

In this section, we present some of the security considerations surrounding asymmetric key cryptography. These considerations contain important principles that form the basic criteria for implementing Linca securely.

Cipher Choice and Key Size

Through our literature survey of asymmetric key crypto systems, RSA is still currently the recommended implementation. However, because of the threats described in section 2.4.2 concerning RSA, one still needs to be wary of its proper utilization, especially the fact that plain RSA encryption is not IND-CCA secure. If the RSA cipher is used on its own to encrypt a message, the same ciphertext will be produced each time the same public key and plaintext are encrypted. Encoding schemes such as the RSAEP-OAEP ensure that no same ciphertext is produced for the same messages. The importance of such encoding is vital since the chances of the message encrypted by a mobile device will be the same because of the small message size. Unfortunately the security reduction of RSAEP-OAEP is inefficient but is nevertheless the most recommended scheme.

Fundamental 9: Use RSAEP-OAEP instead of only RSA cipher for encryption/decryption

As mentioned earlier in section 2.4.2, the advancement in technology makes the factoring of *n* more feasible; therefore to protect data until the year 2030, RSA Labs recommends a 2048-bit key [29]. Also, due to the lengthy process of upgrading RSA keys, 2048-bit is recommended for new applications.

Fundamental 10: Use 2048-bit key as a minimum for RSA cryptography on new applications

Key Management and Public Key Infrastructure

The approaches towards managing asymmetric keys are somewhat different compared to symmetric. For asymmetric keys, the private and public key pairs can be generated temporarily during communication between two or more entities, but in practical systems, public and private-keys are used for a longer duration after they are generated. On mobile devices, care must be taken not to try to generate the asymmetric keys on the device itself due to limited resources such as memory and processing power.

Asymmetric key cryptography is susceptible to man-in-the-middle attacks. A man-in-the-middle attack is when an attacker is able to read, insert and modify at will, messages between two parties without either party knowing that the link between them has been compromised. Therefore, in order for asymmetric key cryptography to work securely, the public-key used for encryption must belong to the legitimate owner that will do the decryption.

There are three ways [70] that one could get hold of public-keys. The first is receiving the public-key from the receiver personally, for example on a digital media (given to the recipient face-to-face) or in a password-encrypted file zip file via e-mail etc. The second method is from a public-key certificate obtained from a Certification Authority (CA) – an entity that is trusted to issue public-key certificates – through their centralized database. This method is the most commonly used within the public-key infrastructure (PKI) and we will discuss it more detail in the next paragraph. Lastly, the public-key of the receiver can be obtained through a distributed key management using introducers if the communication parties do not trust any CA. Introducers are other users of the system who sign their friends' public keys. For example, a company *X* might act as the introducer who signs all their employee's certificates to use within the company, without getting the CA involved. The downside to this approach is that the user of the public-key might not get to know the introducer, therefore there is no guarantee the user will trust the validity of the public-key.

In a more practical setting, asymmetric key cryptography is used the PKI. A PKI consists of protocols, services, and standards supporting applications of public-key cryptography. Among the services likely to be found in a PKI are the following [68]:

- **Key registration:** issuing a new certificate for a public key.

- **Certificate revocation:** cancelling a previously issued certificate. This is often necessary when a private-key is compromised before the expiry date of the corresponding public-key certificate.
- **Key selection:** obtaining a party's public key.
- **Trust evaluation:** determining whether a certificate is valid and what operations it authorizes.

There are certain risks associated with PKI namely the level of trust in the CA and the person who the CA certified on the certificate, and the speed and reliability of the revocation [12, 13]. An actual case concerning the risks of PKI is that VeriSign CA issued two certificates for code signing to a person who claimed to be an employee of Microsoft [6]. Any code signed by these certificates will appear to be signed by Microsoft while in actual fact it is not.

Key Utilization

Key utilization in asymmetric key cryptosystems is different compared to symmetric key cryptosystem [70]. A single public and private key pair is not good enough as there might be a requirement for separate key pairs for encryption and signing. Also, one person could own different public and private-key pairs to be used for various different roles that person possesses. For example, one could sign or encrypt data according to his/her position within a company or some other position in a private organization. The challenge is to keep track of these multiple key pairs and to make sure that they are used according to their intended purpose.

Fundamental 11: Keep track of the key pairs and their usage criteria

2.5 Hash Functions

Hash functions can be used for message authentication, key derivation, pseudorandom-number generation and digital signature generation. A hash function *H* must have the following properties [75]:

1. *H* can be applied to a block of data of any size.

2. *H* produces a fixed-length output.
3. *H(x)* is relatively easy to compute for any given data *x*, making both hardware and software implementations practical.
4. For any given digest *d*, it is computationally infeasible to find *x* such that *H(x)* = *d*. This point defines the one-way property.
5. For any given block *x*, it is computationally infeasible to find $y \neq x$ with $H(y) = H(x)$.
6. It is computationally infeasible to find any pair of messages (x, y), such that $H(x) = H(y)$.

A hash function satisfying the first five points is referred to as a weak hash function and if the 6th property is satisfied, then it is referred to as a strong hash function.

In this section, we will present a study on hash functions, especially the Secure Hash Algorithm due to its robustness. Other one-way hash functions considered but not implemented in Linca due to security weakness is presented in section 2.5.2.

2.5.1 Secure Hash Algorithm (SHA)

The Secure Hash Algorithm (SHA) was designed by the NSA and standardised by NIST and published as FIPS PUB 180 in 1993; a revised version was issued in 1995 as FIPS PUB 180-1 and the current revision as of this writing is FIPS PUB 180-2 with a change notice for SHA-224 in February 2004 [48]. The lists of SHA algorithms are: SHA-1, SHA-256, SHA-384 and SHA-512. Due to the heavier computations in SHA-384 and SHA-512, we feel these two algorithms are not to be recommended to be used on mobile computing devices. This leaves a study focus between SHA-1 and SHA-256. SHA-1 is currently the most widely used algorithm in the SHA family. SHA-1 takes an input message size of less than 2^{64}-bits and produces a digest of 160-bits. SHA-256 also takes an input message size of less than 2^{64}-bits however it produces a digest of 256-bits. The possibility of coming up with two messages having the same message digest is on the order of 2^{80} operations for SHA-1 and 2^{128} operations for SHA-256, while the difficulty of finding a message with a given digest is on the order of 2^{160} and 2^{256} operations for SHA-1 and SHA-256 respectively.

There is a new attack on SHA-1 without using brute force that found collisions within 2^{69} hash operations [90]. The impact of this result brings a step closer to finding collisions in SHA-1 within a feasible 2^{64} complexity.

Due to advances in technology and attacks, NIST plans to phase out SHA-1 by the year 2010 and recommends stronger hash functions to be used. This comment entices SHA-256 to be used in our research.

Unfortunately, SHA-256, SHA-384 and SHA-512 are fairly new for a cryptographic algorithm and extensive cryptanalysis has not been done. Nevertheless these algorithms are standardised by NIST.

2.5.2 Other One-way Hash Functions

A number of hash algorithms including MD4, MD5 and superseded Federal Standard SHA-0 are more prone to collision attacks [13]. In addition the authors in [90] mention that the technique used to attack SHA-1 can be done on MD5 which improves the probability of collisions from 2^{-37} to 2^{-32}.

Recommendation 4: Use Sha-256 as the one-way hash function

2.5.3 Security Issues

Aside from collision problems, there are further security issues involving one-way hash functions [13]:

Length extensions. A message m is split into blocks $m_1,\ldots,m_k$ and hashed into a value d. Another message m' is split into blocks $m_1,\ldots,m_k,m_{k+1}$. Because the first k blocks of m' are identical to the k blocks of message m, the hash value $H(m)$ is merely the intermediate hash value after k blocks in the computation of $H(m')$ thereby giving $H'(H(m), m_{k+1})$. The length extension problem exists because there is no special processing at the end of the hash function computation. The result is that $H(m)$ provides direct information about the intermediate state after the first k blocks of m'. An attacker can construct a few suitable pairs (m, m') and check for this relationship. As a consequence of length extensions, the attacker can append text to m and update the hash to match the new message.

Partial-message collision. Assuming a system that authenticates a message by $H(m \parallel s)$, where s is the authentication key. The attacker can choose the message m, but the system will only authenticate a single message. A perfect hash function will provide a security level of n-bits. The

attacker can find two message pairs (m, m') that lead to a collision when hashed by H. This can be done using a birthday attack of around $2^{n/2}$ steps. The attacker of the system can authenticate m, and replace the message with m'. Because of the iterative nature of one-way hash functions, hashing m and m' results in the same value $H(m \parallel s) = H(m' \parallel s)$ for every S. This means that the attacker does not need to derive s and the whole message ($m \parallel s$) contains a partial collision occurring in m and m'.

2.6 Message Authentication Code

A message authentication code (MAC) is a construction that prevents tampering with messages. MAC shares the first five properties of one-way hash functions described in section 2.5 with the exception that it involves the use of a secret key as an extra input along with the message to be authenticated. This means that instead of the normal routine of applying a hash function to generate a digest d from the input x denoted by $H(x) = d$, we now have $M(s, x) = d$ where M is the MAC function that generates d by taking in secret s and message x as the inputs.

A one-way hash function cannot be used as a MAC because it does not rely on a secret key. One-way hash functions could operate similar to a MAC as $H(s \parallel x)$, $H(x \parallel s)$ or $H(s \parallel x \parallel s)$, however [13] mentions that they would not be secure because having s at the front allows length extension attacks and having s at the end allows a clever key-recovery attack in about $2^{n/2}$ steps. To conclude, a one-way hash function is not designed for use as a MAC.

Fundamental 12: Avoid using one-way hash functions as a MAC

There have been a number of proposals for the incorporation of a secret key into an existing one-way hash algorithm. The approach that has received the most support is HMAC [75]. HMAC has been issued as RFC 2104 [27] and has been chosen as the mandatory-to-implement MAC for IPSec, and is used in other interent protocols, such as TLS and secure electronic transaction (SET) [75].

2.6.1 HMAC

RFC 2104 lists the following objectives for HMAC:

- To use without modifications, available one-way hash functions (in particular functions that perform well in software, and for which code is freely and widely available).

- To allow for easy replace-ability of the embedded hash function in case faster or more secure hash functions are found or required.

- To preserve the original performance of the hash function without incurring significant degradation.

- To use and handle keys in a simple way.

- To have a well-understood cryptographic analysis of the strength of the authentication mechanism based on reasonable assumptions on the embedded function.

The first two objectives treat HMAC as a software component. HMAC is seen as a "black-box" that accepts a one-way hash function to be embedded as part of its implementation. More importantly, if the one-way hash function is "cracked" it can be replaced with a securer one. The last design objective denotes that the strength of the HMAC can be proven secure if the embedded hash function is secure.

The HMAC also provides support to withstand length extension and partial-message collisions [13, 27]. The secret key value used as one of the inputs in HMAC is also protected against key recovery attacks that could be launched by an attacker that does not have any interaction with the system (offline attacks).

2.6.2 Other MAC functions

Other MAC functions worth mentioning are CBC-MAC and UMAC. CBC-MAC is a method of turning a block cipher into a MAC. CBC-MAC is generally considered secure if the underlying cipher is secure. However CBC-MAC is tricky to be used properly as it is dangerous to use the same key for CBC encryption and CBC_MAC authentication [13]. Because CBC is not supported in Linca, CBC_MAC will not be supported anyway.

UMAC family of MAC functions can make use of the uncertainty of the secret value *s* to make a faster MAC function. It can be said that UMAC is faster than HMAC, and there are proofs of security for UMAC. However UMAC is limited by poor digest output size of 64-bits and complexity arising from different implementations [13].

Based on the robustness and ease of use of the HMAC when compared with other MAC algorithms, we recommended it as the MAC algorithm in Linca.

Recommendation 5: Use HMAC as the MAC function

2.6.3 Applying MAC

When entity *A* receives a value *M*(*s*, *m*), *A* knows that someone who knows the secret *s* has approved message *m*. However the problem is that an attacker can capture the MAC generated in *M*(*s*, *m*) and reply back to *A* at a later stage and as a result *A* would still see this as a valid MAC.

The Horton Principle states: *"Authenticate what is being meant, not what is being said"* [13]. This applies in authenticating a message in such a way that the MAC should authenticate not only *m*, but also certain information required that are used to form *m* to make *m* unambiguous. To make a message unambiguous, a message construction should be computed as:

$a_i := M(i \,||\, l(x_i) \,||\, x_i \,||\, m_i)$, where a_i is composed of the MAC value appended with:

- i as the message number,
- x_i as the context data that could comprise of the message number, protocol version number, negotiated field sizes and so on, basically x_i is just a string value that both engaging entities should be able to compute,
- $l(x_i)$ as the function that returns the length of x_i,
- m_i as the message to be authenticated.

Both integer values of i and $l(x_i)$ should be greater than 0 and encoded into their respective endian format according to the underlying machine architecture or platform. The purpose of $l(x_i)$ is to ensure that the string s: = $i \,||\, l(x_i) \,||\, x_i \,||\, m_i$ uniquely parses into its fields and to avoide the various ways to split s into i, x_i, m_i that could cause ambiguity.

Fundamental 13: Authenticate what is being meant, not what is being said

2.7 Public-key Certificates and Digital Signatures

Many different formats for certificates exist, but the most widely deployed is the X.509 version 3 certificates.

The ISO standard for a digital certificate structure is based on the X.509 standard [28]. X.509 defines a framework for the provision of authentication services through the X.500 directory that consists of a repository of public-key certificates. Each certificate consists of the public-key of a user and is signed with the private-key of a CA. X.509 was initially issued in 1988 and a revised recommendation was released in 1993 to address some of the security concerns with versions 1 and 2. A third version was drafted in 1995. X.509 also includes standards for certificate revocation list (CRL) that provides means on how certificates can be revoked. X.509 is currently being used in SSL/TLS, IPSec, Secure/Multipurpose Internet Mail Extensions (S/MIME) and SET.

In this section we will discuss the X.509 certificate structure and presents some security issues with X.509 certificates.

2.7.1 Public-key Certificates

Certified X.509 public-key certificates are created by a trusted CA and placed in a directory by the CA or by the user. The directory server only provides an easily accessible location for users to obtain certificates.

User certificates generated by the CA have the following characteristics [75]:

- Any user with access to the public-key of the CA can recover the user's public key that was certified.
- No party other than the CA can modify the certificate without this being detected.

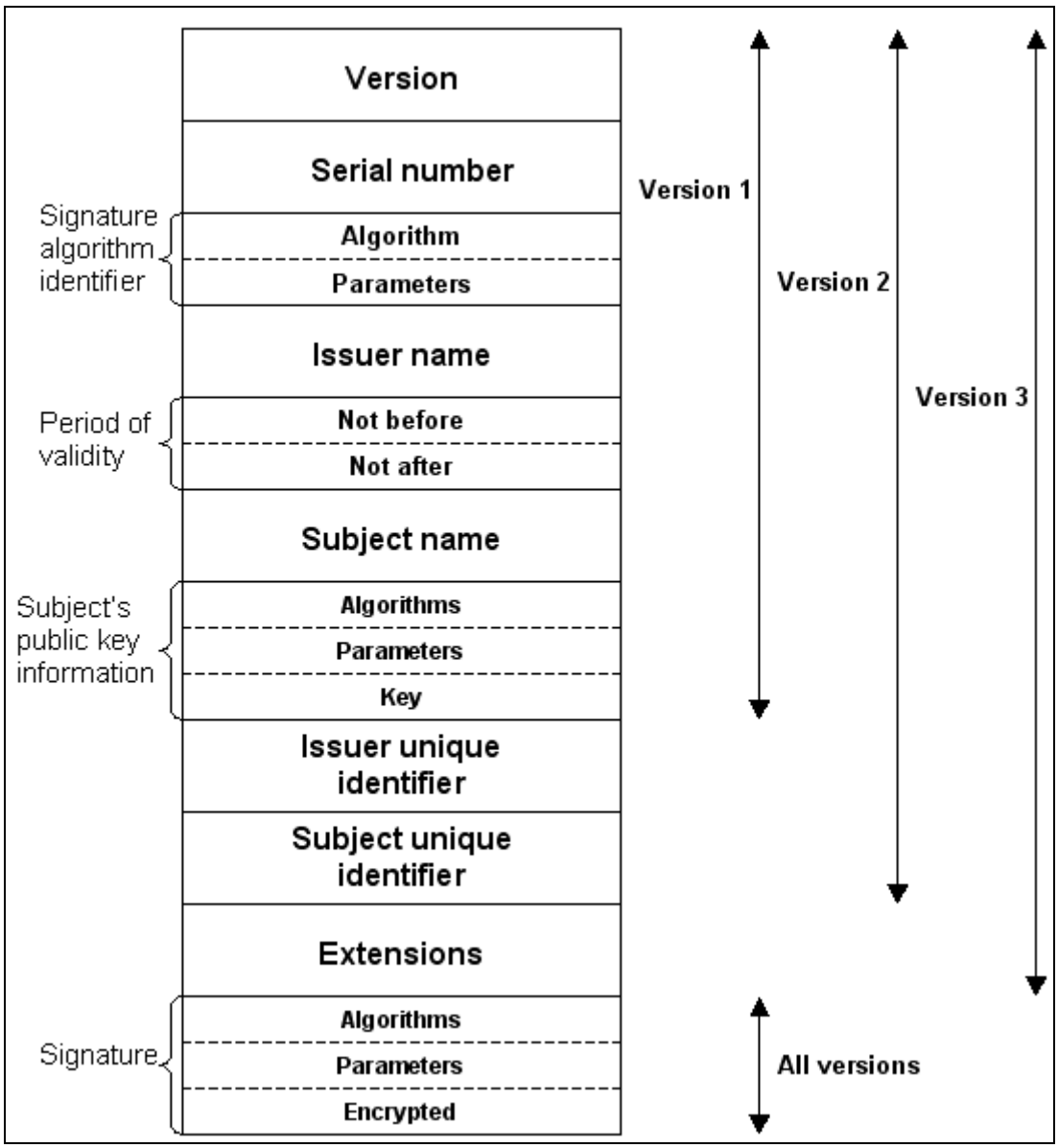


Figure 2-4: X.509 certificate (adopted from [75])

The following elements of the X.509 certificate are summarised from [75]:

- **Version:** This field describes the version of the encoded certificate. The versions include 1, 2 or 3.

- **Serial number:** The serial number *must* be a positive integer assigned by the CA to each certificate. It *must* be unique for each certificate issued by a given CA (i.e., the issuer name and serial number identify a unique certificate).

- **Signature algorithm identifier:** This field contains the algorithm identifier for the algorithm used by the CA to sign the certificate. The contents of the optional parameters field will vary according to the algorithm identified.

- **Issuer name:** The issuer field identifies the entity that has signed and issued the certificate. The issuer field *must* contain a non-empty distinguished name (DN).

- **Period of validity:** The certificate validity period is the time interval during which the CA warrants that it will maintain information about the status of the certificate. The field is represented as a *sequence* of two dates: the date on which the certificate validity period begins (not before) and the date on which the certificate validity period ends (not after).

- **Subject name:** The subject field identifies the entity associated with the public-key stored in the subject public-key field.

- **Subject's public-key information:** This field is used to carry the public key and identify the algorithm with which the key is used (e.g., RSA, DSA, or Diffie-Hellman).

- **Issuer and Subject unique identifiers:** These fields *must* only appear if the version is 2 or 3. The subject and issuer unique identifiers are present in the certificate to handle the possibility of reuse of subject and/or issuer names over time. RFC 3280 *recommends* that names not be reused for different entities and that internet certificates not make use of unique identifiers.

- **Extensions:** The extensions defined for X.509 v3 certificates provide methods for associating additional attributes with users or public keys and for managing a certification hierarchy. The X.509 v3 certificate format also allows communities to define private extensions to carry information unique to those communities. Each extension in a certificate is designated as either critical or non-critical. A certificate using system *must* reject the certificate if it encounters a critical extension it does not recognize; however, a non-critical extension *may* be ignored if it is not recognized.

- **Signature:** The signature field is the hashed value of all the fields within the certificate, encrypted with the CA's private-key. This field includes the signature algorithm identifier.

Security Issues

X.509 standard is a pivotal element for PKI. Versions 1 and 2 of the X.509 certificate have problems in not conveying enough information for identifying the subject and key life cycle management. To alleviate these limitations, version 3 allows the usage of optional extensions.

The last problem relating to X.509 is its original specification. The problem with X.509 specification is somewhat vague and open-ended that caused a lot of confusion about how to implement and work with X.509 certificates. The reader is encouraged to refer to [21] on X.509 implementation problems and guidelines. The certificate support in Linca is recommended to take advantage of the latest specification of X.509.

Fundamental 14: Use X.509 version 3 Certificate

2.7.2 Digital Signatures

Digital signatures provide authentication, authorization and non-repudiation. In this section we will focus on the signature schemes described in PKCS#1 v2.1 because it supports the RSA cipher and the limitations of the signature schemes in PKCS#1 v2.1.

PKCS#1 v2.1 [66] supports two signature schemes based on RSA cryptosystem namely: RSASSA-PSS and RSASSA-PKCS1-v1_5. Although there are no attacks known towards RSASSA-PKCS1-v1_5, the PKCS#1 v2.1 standard recommends RSASSA-PSS in the interest of increased robustness for newer applications. RSASSA-PKCS1-v1_5 is included in PKCS#1 v2.1 for compatibility with existing applications.

Unfortunately the PSS signature scheme has a patent pending applied by the University of California [19]. The University of California has provided a letter to the IEEE P1363 working group stating that if the PSS signature scheme were included in an IEEE standard certain licensing terms regarding PSS would suffice.

These licensing terms are:

- If PSS is included in an IEEE standard, the University of California will, when that standard is adopted, FREELY license any conforming implementation of PSS as a technique for

achieving a digital signature with appendix. No registration fee or other administrative procedure will be required.

- Broader use of the PSS/PSS-R techniques, including the use of PSS-R for achieving a digital signature with message recovery, may be made after acquiring a license. Licenses will be awarded in a non-discriminatory manner under very reasonable terms and conditions.

The PSS signature scheme is specified in the IEEE P1363a draft 10 when PKCS#1 v2.1 was written.

> Recommendation 6: Use a digital signature scheme that is not bounded by patents

Although ECC signature scheme is not bounded by any patents and is more efficient than RSA based digital signature scheme [86, 88], ECC is not widely adopted yet.

> Fundamental 15: The signing algorithm should support RSA and must be easy, secure and efficient to implement

A RSA based signing has a security risk that an attacker can get a legitimate entity A to sign M_3 by generating two messages M_1 and M_2 such that $M_3 \equiv M_1 M_2 \pmod{n}$. If the attacker can get A to sign M_1 and M_2, M_3 can be calculated by multiplying $(M_1^d * M_2^d) \bmod n = M_3^d \bmod n$ [13].

2.8 Export Regulations on Open Source Cryptographic Toolkits

Cryptography is export-controlled for several reasons. Strong cryptography can be used for criminal purposes or even as a weapon of war. During wartime, the ability to intercept and decipher enemy communications is crucial. For that reason, cryptographic technologies are subject to export controls [63].

Software that is open sourced must still comply with the laws regarding export control. Because open source software is so readily copied and distributed worldwide, no open source licence can explicitly require compliance with United States law [14]. According to the Bureau of Industry and

Security in the United States Department of Commerce [85], open source software developed within the United States containing cryptographic code can be exported, provided that a notification is sent to relevant government agencies.

Since it is always possible for rules and restrictions to change, much of the cryptographic software within the open source community is developed and maintained outside the United States [14]. In this way any changes made to the United States export regulation will not affect the open source cryptography software.

2.9 Summary

In this chapter, we conducted a literature study on the security issues of cryptography, covering areas including symmetric key cryptography, block cipher modes, asymmetric key cryptography, hash functions, message authentication codes, and digital certificates. Through this literature study, we notice that applying cryptography is an intricate process and should not be attempted in an ad-hoc fashion, as there are vulnerabilities within cryptography itself that should not be overlooked. Open source cryptographic toolkits are maintained outside the United States to avoid the change on rules and restrictions mandated by the export regulation law on cryptography. The entire chapter is highlighted with cryptographic fundamentals and recommendations in applying cryptography which will be used in the design and implementation of Linca.

These fundamentals, which are critical in security and efficiency, are:

- **Fundamental 1:** Use 128-bit block ciphers.
- **Fundamental 2:** Symmetric key utilization is not the same for every cipher.
- **Fundamental 3:** A cryptographically strong PRNG should use real random data to seed itself and that the internal state of the PRNG is difficult to acquire.
- **Fundamental 4:** Use session keys for network communication and PBE keys for on-device data protection.
- **Fundamental 5:** Choose the key size that will give 128-bit security after collision attacks.
- **Fundamental 6:** A MAC should be computed for the ciphertext produced by the encryption mode.
- **Fundamental 7:** Use a nonce-generated IV.
- **Fundamental 8:** Use CRT to speed up decryption speed.
- **Fundamental 9:** Use RSAEP-OAEP instead of only RSA cipher for encryption/decryption.

- **Fundamental 10:** Use 2048-bit key as a minimum for RSA cryptography on new applications.
- **Fundamental 11:** Keep track of the key pairs and their usage criteria.
- **Fundamental 12:** Avoid using one-way hash functions as a MAC.
- **Fundamental 13:** Authenticate what is being meant, not what is being said.
- **Fundamental 14:** Use X.509 version 3 Certificate.
- **Fundamental 15:** The signing algorithm should support RSA and must be easy, secure and efficient to implement.

The recommended cryptographic algorithms that are supported in Linca which are justified from the fundamentals, are:

- **Recommendation 1:** Use AES cipher for symmetric encryption/decryption.
- **Recommendation 2:** Use CTR block cipher mode.
- **Recommendation 3:** Use RSA cipher for asymmetric encryption/decryption.
- **Recommendation 4:** Use Sha-256 as the one-way hash function.
- **Recommendation 5:** Use HMAC as the MAC function.
- **Recommendation 6:** Use a digital signature scheme that is not bounded by patents.

Chapter 3 - An Overview of Mobile Cryptography Packages

"If you build it, someone will inevitably hack it." --Leander Kahney

As mentioned in chapter 1, in order to provide sufficient security in the mobile application environment, a cryptographic toolkit is needed. In this chapter we give an overview of two well-known cryptographic packages namely Bouncy Castle API and Secure and Trust Service API. These two APIs will be evaluated against Linca in terms of security and design in chapter 6. The motivations for studying these two APIs in this research are mentioned in their respective sections.

3.1 Lightweight Bouncy Castle API

The Bouncy Castle cryptography API (BC) [3] is a free, clean-room, open-source JCE provider which is maintained in Australia. BC developers developed their own lightweight API to be wrapped in BC JCE provider classes. There are two motivations for studying BC in our work. The first is the popularity this API has amongst J2ME developers which is mentioned in many sources [30, 33, 93]. The second reason is the source code can be used to illustrate the security issues in cryptography that were identified in chapter 2. The security in Linca can be improved by learning from these security problems. At the time of this writing, the BC API version we used in this dissertation is *1.33*.

A simplified package structure of the BC package is illustrated in Figure 3-1. The BC package have its own implementation of `java.math.BigInteger`, `java.security.SecureRandom` and `java.io` due to the lack of support of such classes and package on MIDP. The base package that supports the cryptographic algorithms and padding schemes is the `org.bouncycastle.crypto` package. The `org.bouncycastle.asn1` package supports the parsing and writing ASN.1 objects, which is useful in processing X.509 digital certificates.

For implementing an open-source based Pretty Good Privacy (PGP) application, the developer can utilize the classes within the `org.bouncycastle.bcpg` package. The `org.bouncycastle.math.ec` package provides further mathematical support for Elliptic Curve cryptography. The utility classes in `org.bouncycastle.util` can be used for

producing and reading Base64 and Hexadecimal strings. This utility is useful if the ciphertext is required to be displayed as a Hexadecimal string.

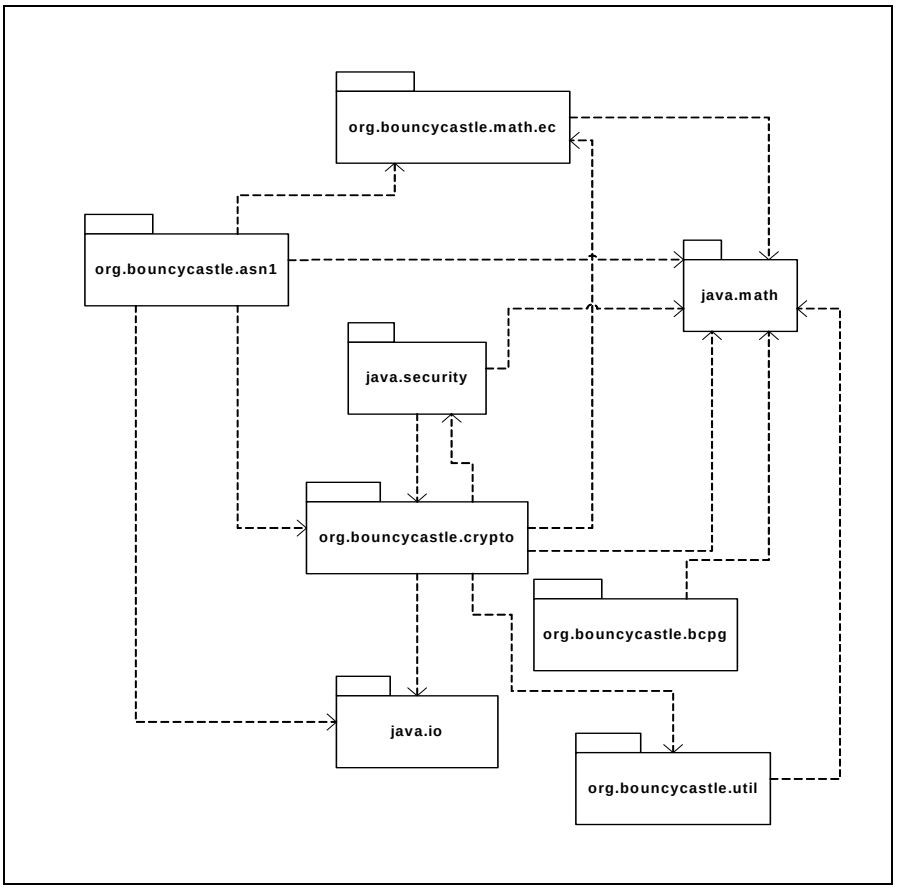


Figure 3-1: Lightweight Bouncy Castle API package structure

BC supports the cryptographic algorithms recommended in chapter 2, however it also contains weak algorithms such as DES. The lightweight API `jar` file is just under 1MB. However, most mobile applications would require only a subset of BC algorithms. The developer can use an obfuscator to shrink the unnecessary classes.

3.1.1 Shortcomings

There are two main shortcomings that BC has, namely the ad hoc development model and a lack of secure key management.

The ad hoc development model has the following problems [93]:

- There are too many algorithms supported and there are not enough volunteer developers to optimise everything. The lack of optimisation results in relatively poor performance, especially for some asymmetric key algorithms.

- The BC API design is flexible but quite complex and beginners find it hard to learn. Some developer-friendly API features are missing. For example, it lacks a set of ready-to-use general-key serialization APIs.

- The BC development community might not respond to e-mail queries by other developers outside the community in a timely manner.

The task of key management in BC is shouldered onto the developer. Aside from the complexity nature of key management, other problems we found are:

- The seed used to initialize the `SecureRandom` class might not be random since the seed is obtained from the system time. This means the chances of deriving the secret key generated for symmetric key encryption is high.

- There are no API methods for loading the certificate and private keys from a smart card. The security benefit of a smart card is that it is tamper resistant, which makes it difficult for an attacker to extract information from it. In this way, a smart card can provide a secure storage mechanism for a private key and a certificate. An example of a smart card on a mobile phone is the SIM card. BC has methods to load certificates and private keys from a file and it is up to the developer to ensure their secure storage and authenticity.

3.2 Secure and Trust Service API

The Secure and Trust Service API (SATSA) is distributed as an optional package in MIDP on certain mobile phones. The SATSA specification [81] states that both SATSA and the application using SATSA must trust the OS. SATSA is dependant on certain services provided by the operating system of the mobile device. Despite initial rejections through the Java Service Request (JSR) process, Sun was granted a rework on this proposal and the SATSA specification is accepted on the 28th June 2004 [81]. The motivation for studying SATSA is because it is the current cryptography package for MIDP 2.0. At the time of writing, we are referring to the SATSA specification 1.0 and the reference implementation (RI) 1.0 from Sun Microsystems [81]. A reference implementation is

a software example of a standard for use in helping other developers to implement their own versions of the standard. The purpose of a reference implementation is generally to increase awareness and familiarization of the specification within the development community.

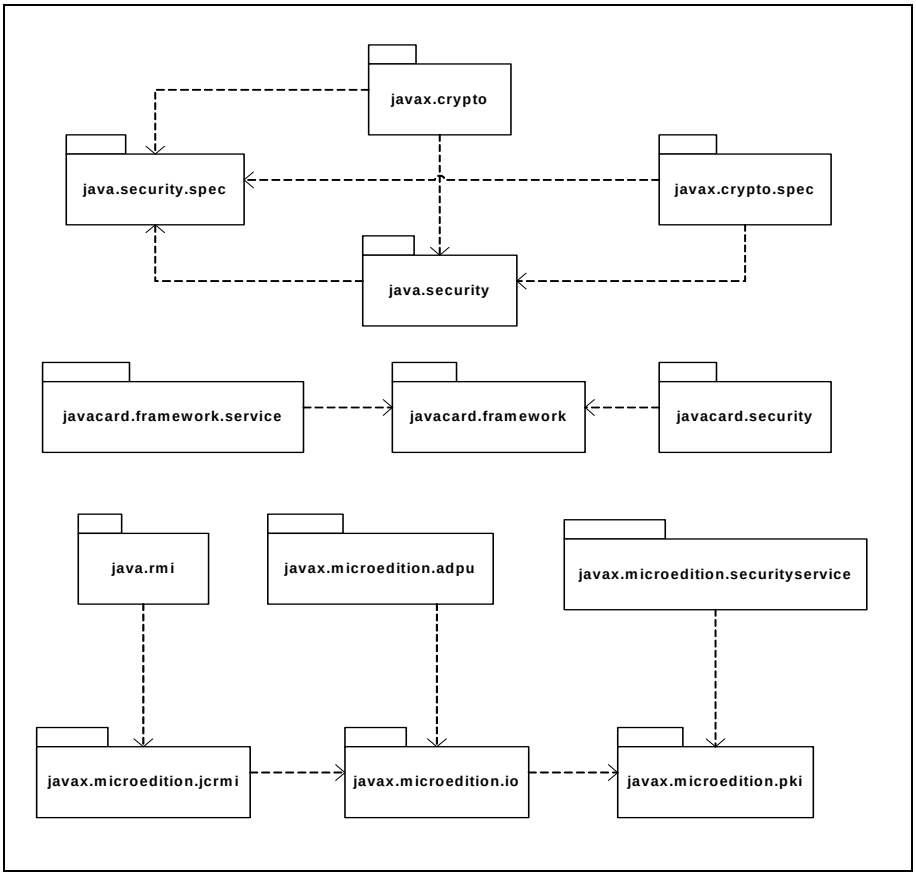


Figure 3-2: Secure and Trust Service API package dependency structure based from SATSA RI 1.0

The Security and Trust Services APIs (SATSA) is an optional package that enables a Java mobile application to [56]:

- Work with public digital certificates, public and private keys, message digests and digital signatures.
- Create, store, and use user-credentials based on X.509 digital certificates
- Encrypt data using asymmetric and symmetric key cryptography.

SATSA relies on a *security element*, which could be an embedded SIM card, removal smart card, or a special hardware inside the device to perform security operations. The exact type of the security element is transparent to the developer since the security element is determined by the SATSA implementation for the particular device.

The SATSA specification defines four distinct APIs:

- SATSA-APDU allows applications to communicate with smart card applications using a low-level protocol. Packages belonging to this API are: `javax.microedition.apdu`, which defines the APDU (Application Protocol Data Unit) protocol handler for ISO7816-4 communication to a smart card device.

- SATSA-JCRMI provides an alternate method for communicating with smart card applications using Java Card Remote Method Invocation (JCRMI). Packages belonging to this API are `java.rmi`, `javacard.framework`, `javacard.security` and `javax.microedition.jcrmi`. The `java.rmi` package used in this API is a subset of the `java.rmi` package in J2SE.

- SATSA-PKI allows applications to use a smart card to digitally sign data and manage user certificates. Packages in this API are: `javax.microedition.pki` and `javax.microedition.securityservice`. The `javax.microedition.pki` package defines classes to support basic user certificate management, while the `javax.microedition.securityservice` defines classes to generate application-level digital signatures that conform to the Cryptographic Message Syntax (CMS) format.

- SATSA-CRYPTO is a general-purpose cryptographic API that supports message digests, digital signatures and ciphers. The packages that belong in this API are: `java.security` and `javax.crypto`. The `java.security` and `javax.crypto` packages provide the classes and interfaces for the security framework and cryptographic operations respectively. Their subpackage `spec` provides classes and interfaces for key specifications and algorithm parameter specifications.

The Generic Connectivity Framework (GCF) defined in the `javax.microedition.io` is modified to include new APDU and JCRMI connections for smart cards.

3.2.1 Shortcomings

SATSA API has the following shortcomings:

- Signature verification process is not as easy as signature creation. SATSA generates signed messages using the CMS format. However to verify the signature using the public key, the application is required to parse the input into data and signature parts respectively.

- Signature generation and signature verification is not handled in the same way in terms of how information is presented to the user. When data is signed by the user, the underlying SATSA implementation takes control of the user interface (UI) ad presents the user with the certificate to be used for signing, along with the data to be signed. The user can then be confidant that the data signed is the intended data, and not something else. Signature verification is just as important, however the *application* must present details about the signature to the user. This means that trust is placed in SATSA during signing and not verifying.

- SATSA cannot verify certificates. The developer must implement the certificate verification process and the public-key extraction from the certificate, along with presenting the certificate and signed data to the user.

- Private keys stored on smart cards cannot be used for signing since there are no methods available to enable this process.

- The secret key generated by SATSA for symmetric key encryption is based from a key material. This material is usually a user's PIN or some kind of a password. The problem here is that this scheme is only suited for on-device data encryption since the constructed key is not a session key.

- As mentioned earlier, SATSA is only supported on certain phone models and phones without SATSA will have to use third party cryptographic toolkits such as BC.

- SATSA supports limited cryptographic algorithms. For example it does not support SHA-256 or any MAC computing algorithms.

3.3 Summary

In this chapter, we presented an overview of Bouncy Castle API and Secure and Trust Service API and discussed their shortcomings. By learning from both API's strength and shortcomings will help with a securer design for Linca.

Chapter 4 - Linca Framework

"Small is Beautiful." --E. F. Schumacher

Software architecture is the process of designing the global organization of a software system, including dividing software into subsystems, deciding how these will interact, and determining their interfaces [35]. In this chapter, we present a high-level architectural overview of Linca in the form of package structures and illustrate how some of these packages enable classes within them to conform to certain architectural styles and design patterns that will lead to a good design.

Most of the packages within Linca can be seen as *software components* and the rest are utility modules. The aim is to create a collection of reusable software entities as building blocks for applications, thus component software allows complete applications to be created out of small pieces of software.

Every software component within Linca supports a *provided interface*, which defines the operations that must be implemented for that particular component. Interactions are possible amongst these components through their interfaces and this ensures the components are unaware of each others' implementations. Future updates on any components can take place without affecting each other. Linca mainly comprises of cryptographic and various supporting components that interact with each other to perform cryptographic services.

Figure 4-1 shows that Linca contains five main packages namely: `core`, `crypto`, `io`, `math` and `util`. The entire Linca framework is packaged in the root package called `linca`. These packages will be discussed in more detail in their respective sections in this chapter. The *application* package can be seen as any piece of software that uses Linca. Depending on the nature of the application, it can be a single software component (for example a protocol) or the entire security system (which could be composed of a number of components).

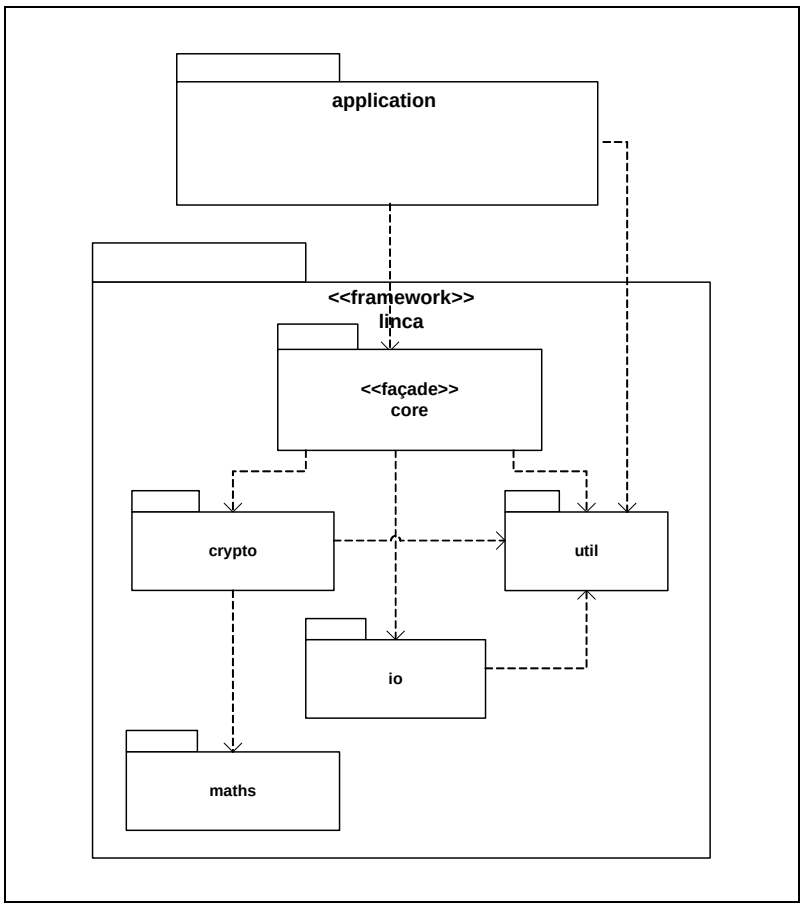


Figure 4-1: The Linca framework

4.1 Architectural Patterns

An architectural pattern allows one to design flexible systems using components that are independent of each other [35]. Some of the components within Linca are organized into object-oriented and layered architecture styles. These two styles play an important role in encapsulating how Linca the framework is structured.

4.1.1 Object-Oriented Organization

Object-oriented organization encapsulates data and their primitive operations inside an object abstract data type, which interacts with other objects through function or method invocations. In this way, object-oriented systems can hide object implementations so that any changes made to the object will not affect the user (the invoker). The object-oriented organization is illustrated in Figure 4.2.

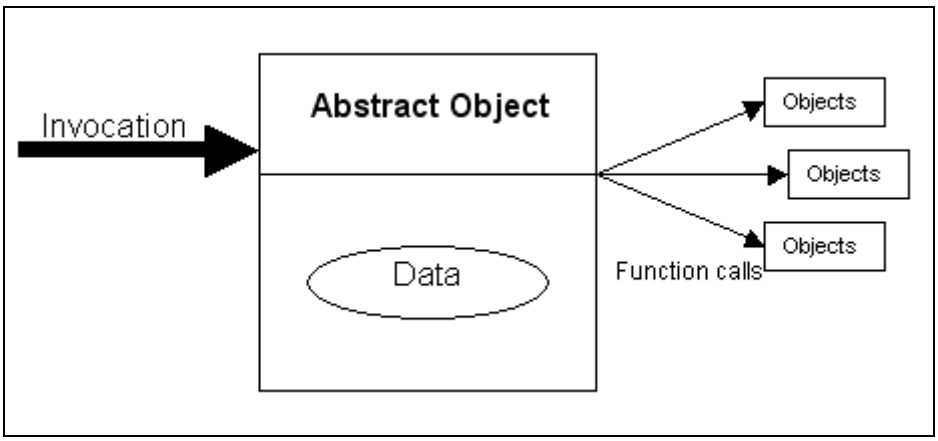


Figure 4-2: Object-oriented organization

4.1.2 Layered Systems

Components are stratified into layers, where data and/or services provided at one layer are available only to the layers above (see Figure 4-3). Should one replace a component beneath another component, the upper component will not be affected.

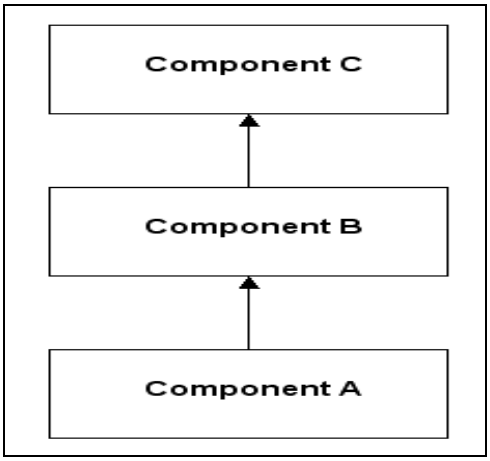


Figure 4-3: Layered system

4.2 Design Patterns

A pattern is the outline of a reusable solution to a general problem encountered in a particular context and a design pattern is a pattern that useful in the design of software [35]. Design patterns are applied within some components that enable a flexible design on deploying an encryption scheme, applying effective cryptography and creating a networking connection. The three main design patterns used are: the Factory, Façade and Delegation design patterns. These three patterns discussed in this section are summarised from [35].

4.2.1 Factory Pattern

The Factory design pattern allows the developer to add a new application-specific class `<<ApplSpecificClass>>` into a system that contains a reusable framework, which creates objects as part of its work. The Factory pattern allows the framework to instantiate the class, without modifying the framework.

The framework delegates the creation of instances of `<<ApplSpecificClass>>` to a specialized class `<<ApplSpecificFactory>>`. The `<<ApplSpecificFactory>>` implements a generic interface `<<Factory>>` defined in the framework. The `<<Factory>>` declares a method whose purpose is to create some subclass `<<AppSpecificClass>>` of a class called `<<GenericClass>>`. This is illustrated in Figure 4-4.

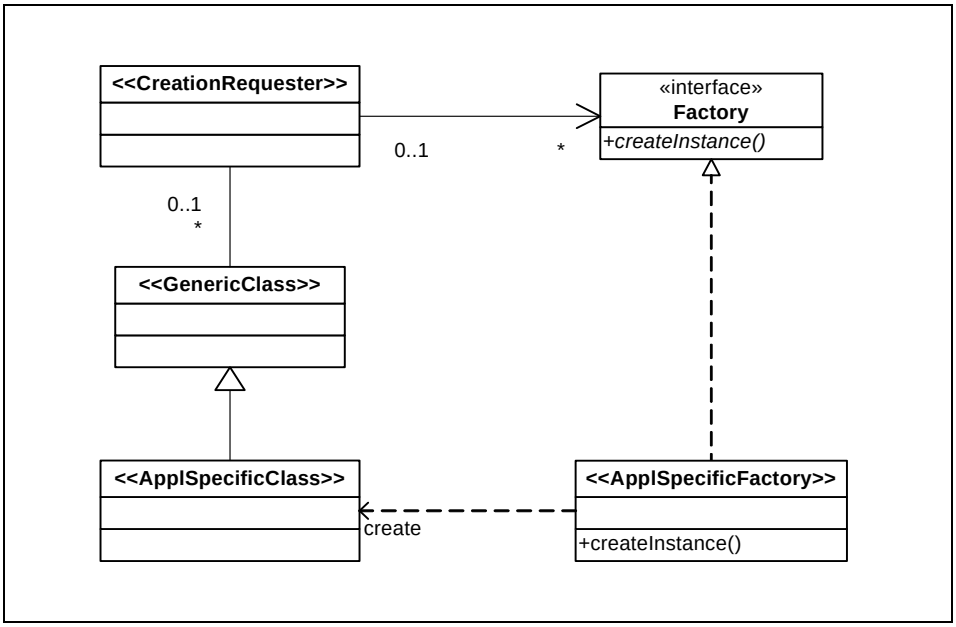


Figure 4-4: Factory design pattern template (adopted from [35])

4.2.2 Delegation Pattern

The Delegation pattern enables one class to access methods of another class without using inheritance as the situation may not be appropriate or because not all methods need to be reused.

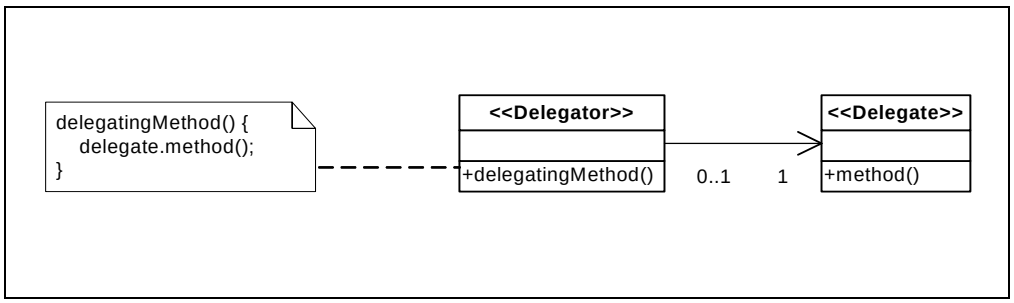


Figure 4-5: Delegation design pattern template (adopted from [35])

Normally, in order to use delegation, an association should already exist between the `<<Delegator>>` and the `<<Delegate>>`. This association may be bidirectional or else unidirectional from `<<Delegator>>` to `<<Delegate>>`. However, it may sometimes be appropriate to create a new association just so that one can use delegation – provided this does not increase the overall complexity of the system. In Figure 4-5, the `<<Delegator>>` class calls a method in the `<<Delegate>>` class, which has an association to the `<<Delegator>>` class.

4.2.3 Façade Pattern

Often, an application contains several complex packages. A developer working with such packages has to manipulate many different classes. The Façade design pattern simplifies the view that developers have of a complex package.

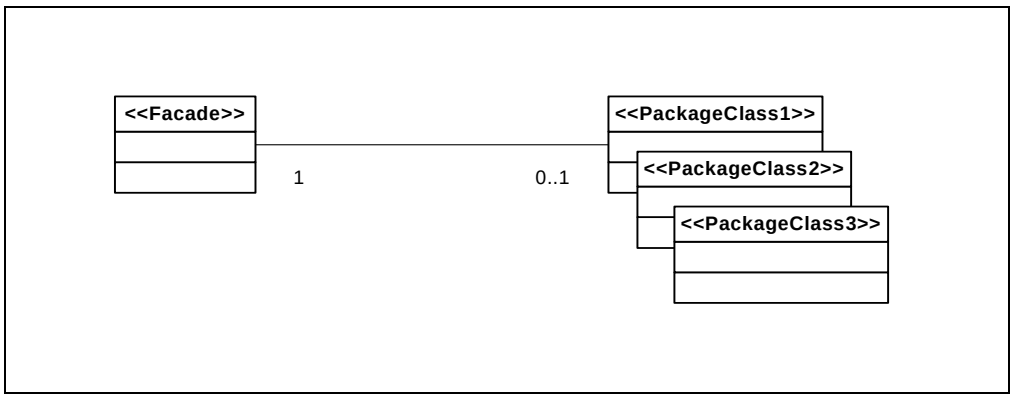


Figure 4-6: Façade design pattern template (adopted from [35])

By creating a `<<Façade>>` class, one can simplify the use of the package (see Figure 4-6). The `<<Façade>>` will contain a simplified set of public methods such that most other subsystems do not need to access the other classes in the package. The net result is that the package as a whole is easier to use and has a reduced number of dependencies with other packages.

4.3 Package Denotations and Definitions

UML notations are used to illustrate how packages within Linca are structured. In order to explain the relationship between these packages, two distinct package notations are used (see Figure 4-7).

Figure 4-7: Package denotations

According to the Fusion Software Process [23], the component identified in the architecture phase can be denoted by UML *subsystem* notation. The UML *component* notation is inappropriate for architectural descriptions because it models implementation level entities such as executables and source code modules. A UML subsystem is treated as a unit with a specification, implementation and identity [69]. Therefore a subsystem package can be seen as a software component prior to its deployment.

In Figure 4-7, ***Notation A*** denotes the component that is being focused upon in terms of the dependencies by other components denoted by ***Notation B***.

The package names are often used solely instead when describing their relationship amongst each other for simplicity. For example, when we describe a certain relationship between package *A* and package *B*, we write, "*A requires B to generate a value*" instead of "*implemented classes within package A require classes in package B to generate a value.*"

4.4 Crypto Package

The `crypto` package holds all the cryptographic primitives recommended in chapter 2. In this section we will discuss these cryptographic primitives in more detail, by grouping them into four functions namely: message authentication, symmetric key cryptosystems, asymmetric key cryptosystems and digital signatures. These functions are discussed in sections 4.4.2, 4.4.3, 4.4.4 and 4.4.5 respectively.

The individual packages within `crypto` that contain these cryptographic primitives are presented in section 4.4.1.

4.4.1 Packages within Crypto

Name	Description
`authentication`	Consists of one-way hash function and MAC algorithm that enables message authentication.
`skcipher`	Consist of a symmetric cipher that encrypts and decrypts one block of data at a time. The block size is determined by the specification of the cipher.
`mode`	Consists of a block cipher mode, which can be used in conjunction with symmetric cipher that encrypt and decrypt at least one block of data.
`symmetrickey`	Consists of a symmetric key generator that generates two types of keys namely: password-based key (PBK) and session key.
`akcrypto`	Consists of a component that determines the asymmetric encryption scheme.
`akcipher`	Consists of an asymmetric cipher that encrypts and decrypts one block of data at a time. The block size is determined by the specification of the cipher.
`encoder`	Consists of an encoding function that is responsible for destroying mathematical structures within a specific asymmetric cryptosystem.
`asymmetrickey`	`asymmetrickey` consist of components that enables the loading of public and private keys from hardware or software, depending on the underlying machine for example a back-end server or a mobile device.
`signature`	Contains a digital signature generator and verifier based on public and private keys.

Table 4-1: Cryptographic primitives used in Linca

In Table 4-1, the reader might wonder why there is only a single cryptographic algorithm supported within the packages as opposed to other cryptographic APIs where there are many cryptographic schemes to choose from. The reason behind Linca's approach is a security consideration and it will be discussed in chapter 5. The components within `crypto` manage their own internal data and provide well-defined interface methods that allow interactions with other components.

4.4.2 Message Authentication

Hash functions do not only provide functionality to authenticate messages, they can also be used during the generation of pseudorandom numbers and digital signatures. The `authentication` package consists of a hash function and a MAC algorithm that enables message authentication. The hash function and MAC algorithm has interfaces that define their implementation.

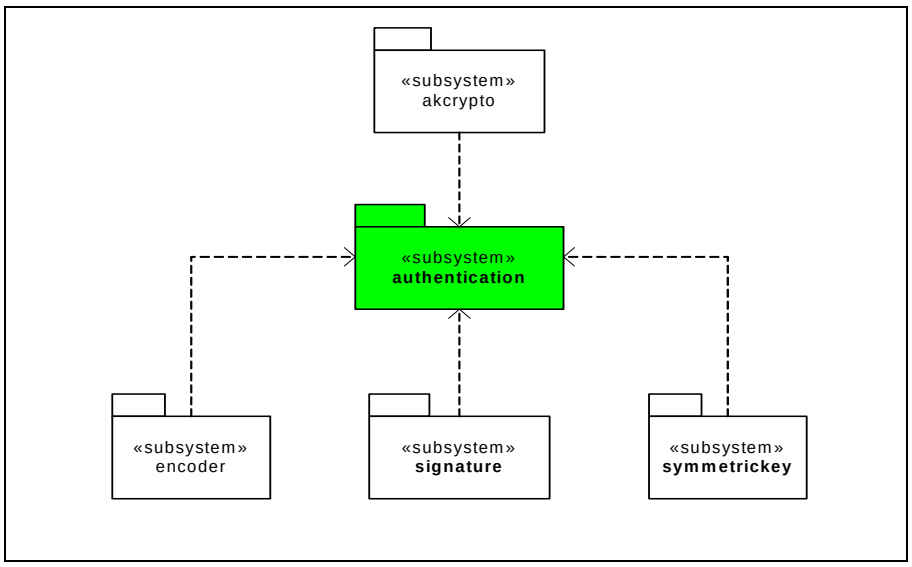


Figure 4-8: `authentication` package structure

Although `authentication` is an independent package within `crypto` (see Figure 4-8), the `encoder`, `signature` and `symmetrickey` packages are dependant on the one-way hash function (this will be discussed under their respective sections). Package `akcrypto` does not invoke methods from `authentication`, but rather passes an instance of the implemented hash function to the underlying asymmetric encryption scheme.

4.4.3 Symmetric Key Cryptosystem

The symmetric key cryptosystem (see Figure 4-9) within Linca is composed of:

- A symmetric cipher,
- a block cipher mode,
- and a key generating mechanism.

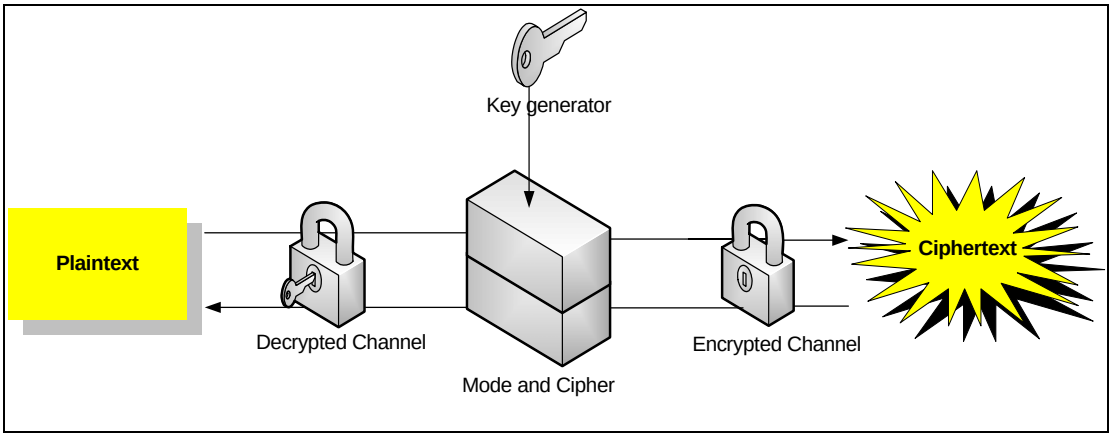


Figure 4-9: Symmetric-key cryptosystem in Linca

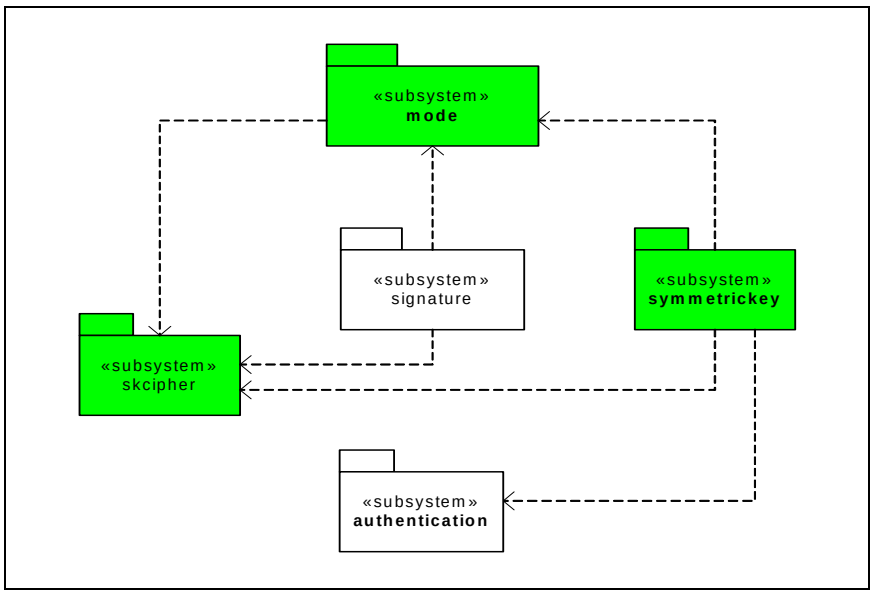


Figure 4-10: Symmetric key cryptosystem package structure

In Figure 4-10, an instance of the symmetric cipher is wrapped in `mode` to enable encryption and decryption on at least one block of data. Therefore method calls are invoked on the interface class in `mode` instead of `skcipher`. The `symmetrickey` requires `mode` to generate either a PBK or a session key of *n*-bit through an encryption process. The key used by the `mode` is generated through the one-way hash function in `authentication` using random bits gathered from an entropy source. The classes within `symmetrickey` do not invoke any method calls from `skcipher`, however there is a dependency because of `skcipher`'s collaboration with `mode`. A class within `symmetrickey` would marshal pieces of data that will be used to construct the secret key. We will call these collective data the *key generating parameter*. The key generating parameter and symmetric key generation are further discussed in chapter 5.

4.4.4 Asymmetric Key Cryptosystem

The asymmetric key cryptosystem (see Figure 4-11) is composed of:

- An asymmetric cipher,
- an optional encoding function,
- and a key loading mechanism.

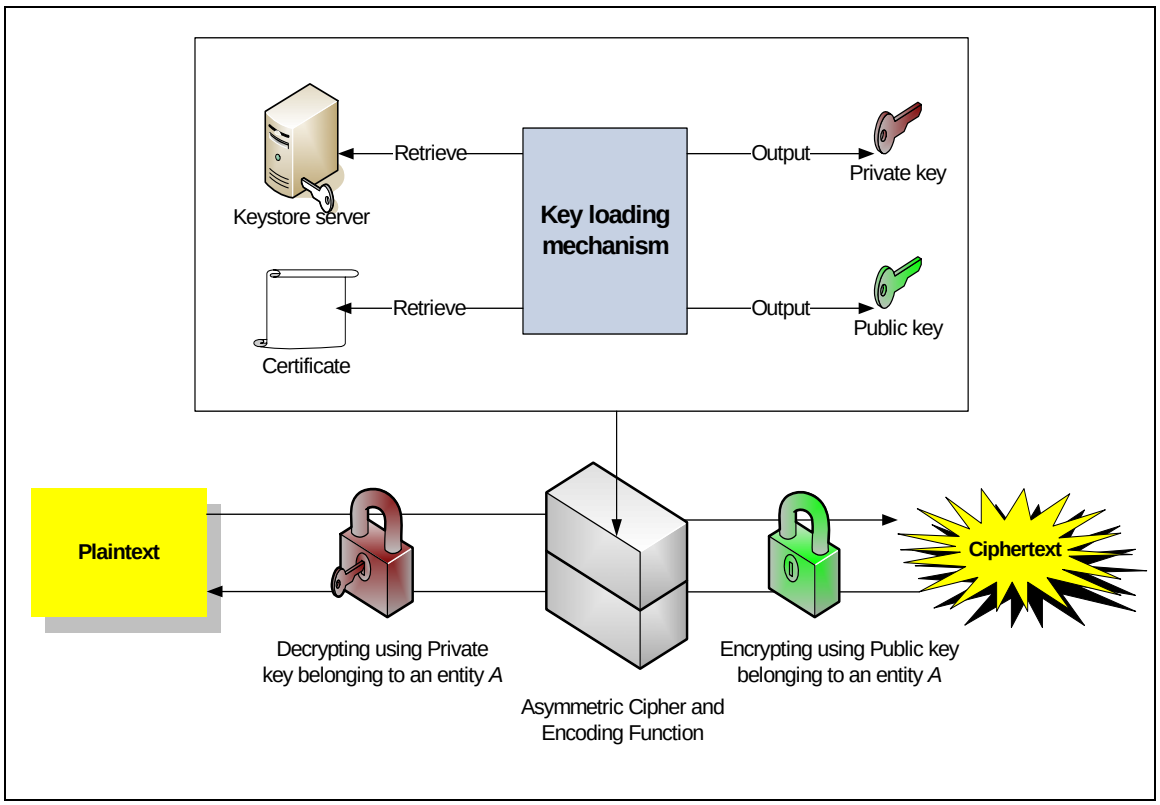


Figure 4-11: Asymmetric key cryptosystem in Linca

The importance of an encoding function is already explained in section 2.4.5 (see Fundamental 9). However, not all asymmetric ciphers require an encoding function, for example ElGamal or Elliptic Curve. The purpose of `akcrypto` is to give Linca the flexibility for updating to the *securest* asymmetric encryption scheme available without affecting other components. In order to achieve this, the implementation class within `akcrypto` follows the Delegation pattern.

In Figure 4-12 the `AsymmetricKeyCryptoImpl` class delegates method calls to either an encryption scheme used with an encoding function or just the asymmetric cipher that is secure enough to use without an encoding function. In this way, the components within `core` are not affected.

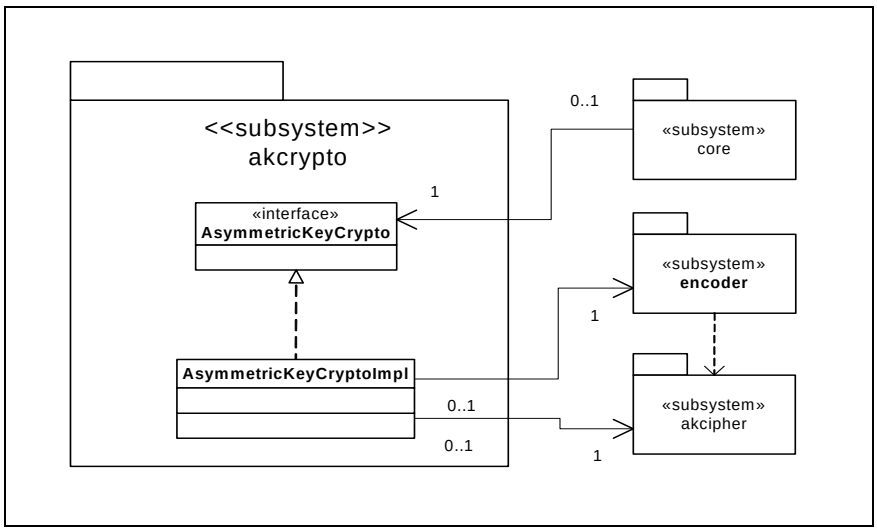


Figure 4-12: Internal encryption scheme loader structure

Because `akcrypto` delegates the method calls to the underlying encryption scheme, it is mostly dependent on the components involved in the asymmetric cryptosystem. However it is interesting to note there is a dependency on `authentication`. According to [66] the `encoder` requires an underlying asymmetric cipher and a one-way hash function in order to function and as a result, the encoder will be dependant on `authentication`.

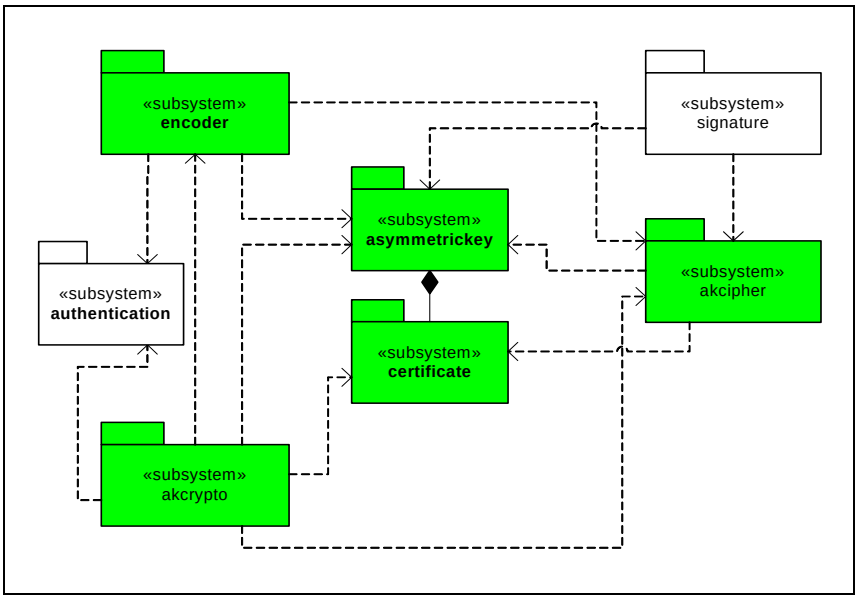


Figure 4-13: Asymmetric key cryptosystem package structure

The `akcipher` package contains the cipher that provides asymmetric encryption and decryption services. Packages such as `akcrypto`, `encoder` and `signature` will depend on `akcipher` to perform their respective cryptographic functions. The `akcipher` package will depend on

`asymmetrickey` and `certificate` to load the public and private keys from a particular storage environment.

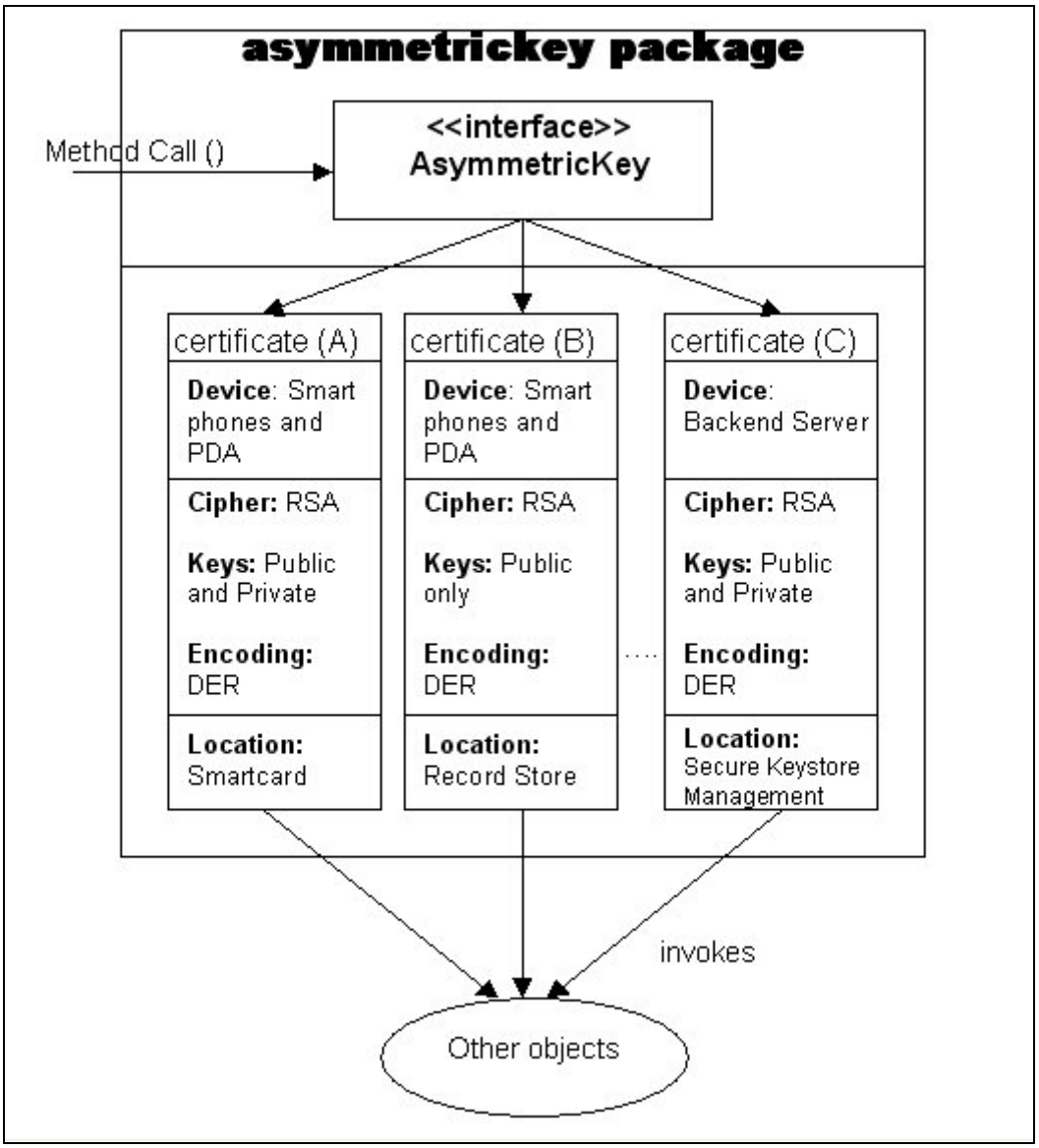


Figure 4-14: Asymmetric key loading architecture

The asymmetric key loading mechanism is adopted from the object-oriented model. The purpose of the key loading mechanism is to manage public and private keys. By using the object-oriented model, it enables `asymmetrickey` to retrieve the public and private keys from any underlying machine type and location (see Figure 4-14).

Linca can be implemented for either a mobile device or a desktop device so that interoperability can be achieved when it is used for enterprise mobile applications; therefore there will be different implementations for each environment. These different implementations must keep the same package structure illustrated in Figure 4-13 to ensure the benefits of the architectural principles

Linca provides. The location of public and private keys can be from hardware or software and could be related to the underlying machine. Examples of hardware locations are smartcards or embedded microchips, while software can be from a secure database or a specific key storage mechanism (for example the standard Java `KeyStore`).

The key-loading classes within Linca are located in the `certificate` package. The `certificate` loads the certificate and private key streams from a specific location and parses the streams into the mathematical elements that constitute the public and private keys for the particular asymmetric cipher.

Suppose in the future that X.509 standard and RSA cipher are replaced with a more secure and efficient implementation, then `asymmetrickey` can be updated by only replacing the components within `certificate`, without affecting any other components.

4.4.5 Digital Signatures

The digital signature mechanism in Linca is provided by `signature` and must enable:

- an entity *A* to compute a signature S_A from a message *m*; and
- an entity *B* to verify S_A *is* computed by entity *A*.

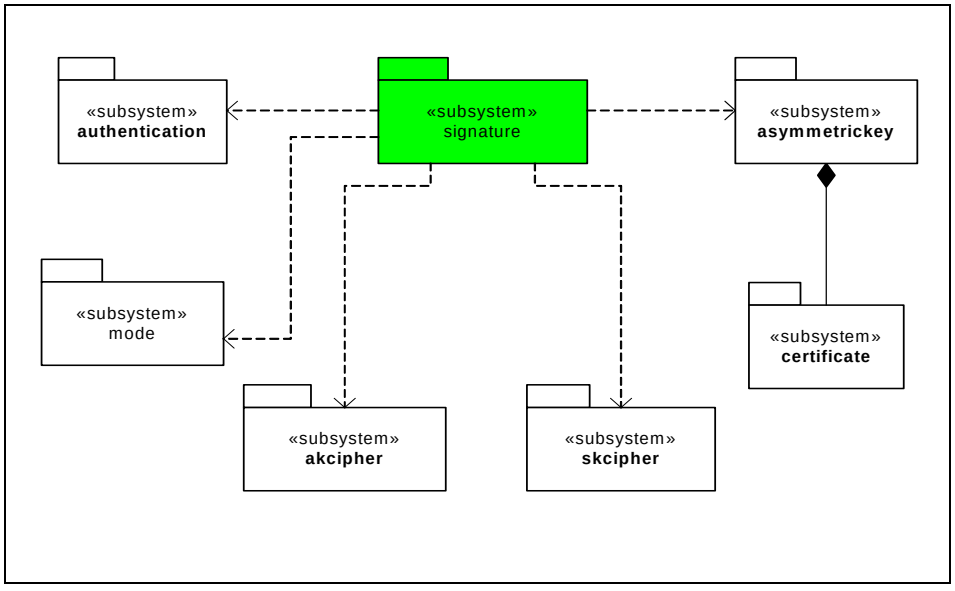


Figure 4-15: `signature` package structure

The `signature` package structure is illustrated in Figure 4-15.

Digital signature generation and verification in Linca involves the usage of asymmetric key cryptography. Encrypting a message using a private key creates a digital signature of the message. A digital signature algorithm would require collaborations with `akcipher`, `authentication`, `asymmetrickey`, and `mode`. It is interesting to see why `signature` would be required to collaborate with `mode`, and the reason is to keep the design as generalized as possible. For example the digital signature-generating algorithm requires the usage of a PRNG and some of the PRNGs might require the usage of a block cipher mode. Although Linca does not contain a PRNG component, `signature` will require `mode` in order to bridge the functionalities offered by a PRNG.

4.5 Mathematic Module

The mathematic module provides a facility for large integer arithmetic calculation that is used in asymmetric key cryptography. Large integer comprises hundreds of decimal digits that are suitable to represent mathematical elements within the asymmetric cipher cryptosystem.

Linca's `math` module provides the facility for `asymmetrickey.certificate` and `akcipher` to perform large integer arithmetic operations specifically for ciphers supported within the framework (see Figure 4-16). These operations includes the addition, subtraction, multiplication, modulo and division of large integers.

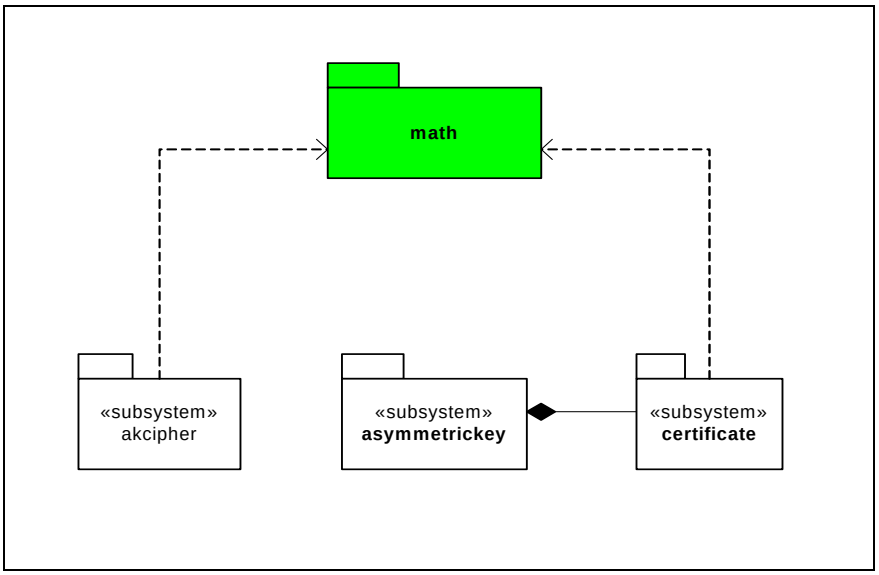


Figure 4-16: `math` package structure

4.6 Utility Module

The `util` package has a class that is responsible for converting:

- `integer` and `long` data types into byte arrays and vice versa.
- Unicode characters into hexadecimal representations.

Literal values such as numbers and strings must be converted into byte arrays before they are passed into a cryptographic function. Unicode characters that are given as outputs after encryption or MAC operation can be represented by a string of hexadecimal values. The `util` package can be reused by other components in Linca or at the application because it does not depend on any other packages (see Figure 4-17).

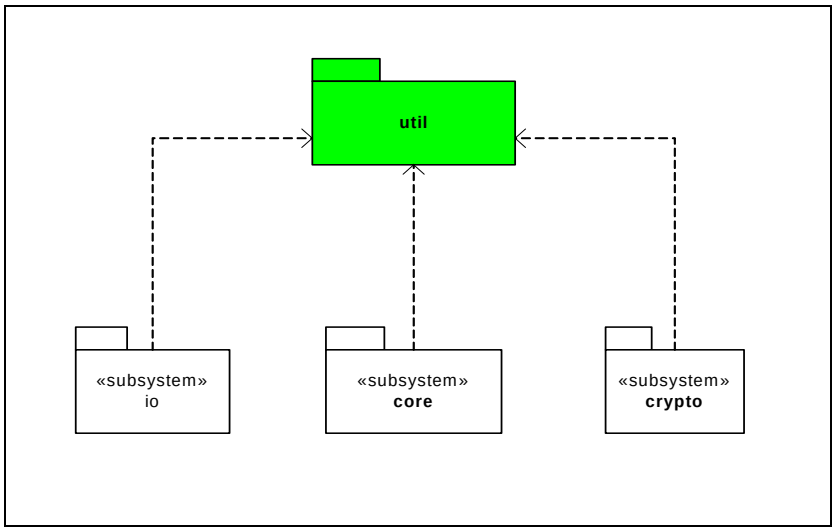


Figure 4-17: `util` package structure

4.7 Linca Connectivity Component

The Linca's connectivity component (LCC) is a sub-framework within Linca that provides networking functionalities to the application when messages are transported over-the-air. The main advantage of the LCC is to give developers the flexibility to implement proper coding practice when using the I/O package that is supported on different makes of mobile phones. For example, the Nokia *Developer Platform 2.0: Known Issues* whitepaper [54] provide recommendations for coding techniques for developers when using specific packages.

One particular guideline concerning networking is illustrated in Figure 4-18. In the wrong approach, the function `read(byte[]a)` might not read the number of bytes equal to the size of the array argument (`a.length`) before returning and will indicate the number of bytes that were

successfully read as a return value because of considerations for optimizing the `read` function's implementation.

```
1   // Defining the connections
2   HttpConnection hc = null;
3   DataInputStream dIn = null;
4
5   byte [] source = byte[512];
6   hc = (HttpConnection) Connector.open("http://localhost:8080");
7   dIn = new DataInputStream(hc.openInputStream());
8
9   //Part A: Incorrect approach in reading a byte stream
10  int bytes = dIn.read(source);
11  dIn.close();
12  hc.close();
13
14  //Part B: Correct approach in reading a byte stream
15  int offset = 0;
16  int bytes = 0
17  while(true) {
18   bytes += dIn.read(source, offset, source.length - offset);
19   offset += bytes;
20   if (bytes == -1 || offset >= source.length) break;
21  }
22  dIn.close();
23  hc.close();
```

Figure 4-18: Problems with reading input streams

The right approach is to read one byte at a time into the buffer. These code guidelines are necessary because different mobile phone vendors will have their own method implemented differently and will most likely publish their own recommended coding guidelines for using their implementation.

Another benefit LCC provides is separation of networking related code from the main application, thus promoting the separation of concerns principle.

The LCC is not a critical component for Linca's functionality and therefore it is optional as developers can implement their own code to handle networking. The LCC package structure is illustrated in Figure 4-19.

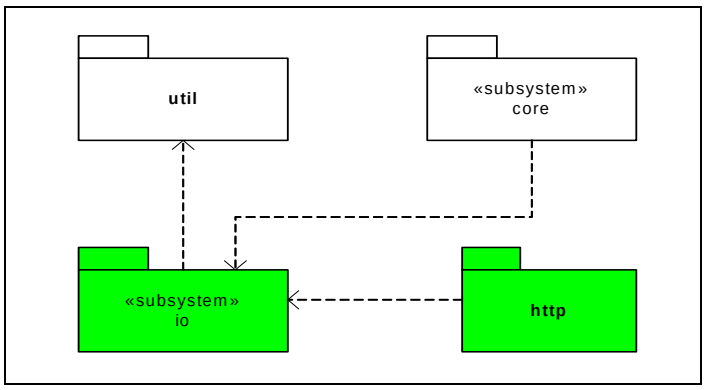


Figure 4-19: `io` package structure

Linca should have the HTTP connection package implemented by default because it is the most commonly used connection type. The HTTP package contains classes that wrap the usage of HTTP connection functions supported by the platform so that it can be incorporated into the LCC. We call these packages *connection packages*.

For better extensibility, the LCC follows the factory design pattern so that other connection types (for example Bluetooth, SMS, FTP etc) can be easily incorporated when required by third party developers (see Figure 4-20).

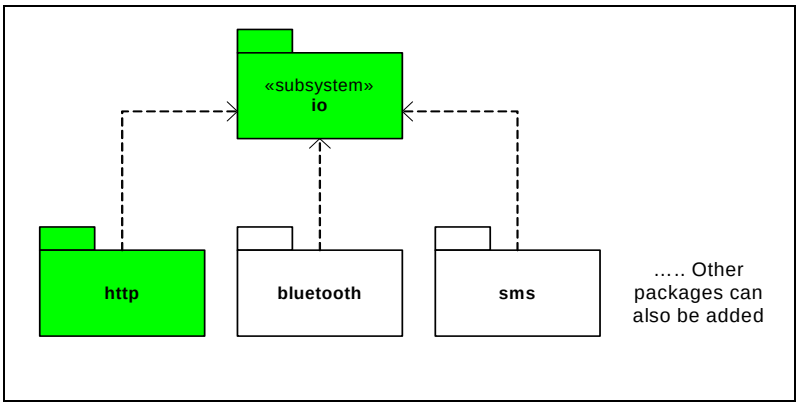


Figure 4-20: Extensibility offered by the LLC

4.8 Core Component

The core component has the responsibility to simplify the usage of the cryptographic algorithms within the `crypto` and `io` packages. Through simplicity, a securer way of applying cryptography can be realized because one can isolate unnecessary operations that could weaken the application of cryptography. To simplify the view that developers have of complex packages, the `core` component is designed using the Façade design pattern.

Unlike other cryptographic toolkits for mobile devices discussed in chapter 3, Linca's `crypto` package can be effectively incorporated into a façade component due to the logical arrangements of cryptographic algorithms (in terms of their functionality) and simpler interface definition.

Once the classes within the core package are implemented, they do not have to be replaced as frequently as the classes within `crypto` or `io`. In this way, developers can focus on the maintenance of algorithms within `crypto` and `io`. The internal structure of `core` component is illustrated in Figure 4-21.

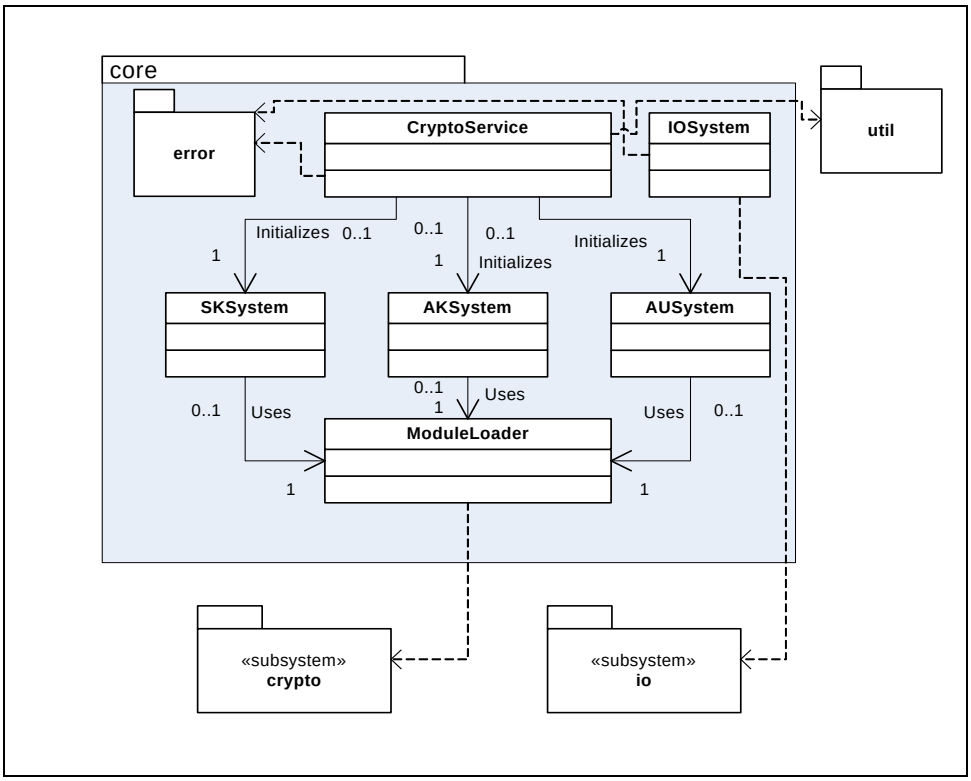


Figure 4-21: Internal structure of the `core` package

The `ModuleLoader` class is responsible for enabling the instantiation of cryptographic primitives to be utilized by three façade classes namely: `SKSystem`, `AKSystem` and `AUSystem`. For simplicity purpose, we will term these three classes *crypto façade*s. The crypto façades are responsible for:

- Logically arranging the cryptographic operations relating to symmetric key cryptography, asymmetric key cryptography and message authentication respectively.

- Instantiating the necessary cryptographic primitives required within the cryptosystem.

- Providing the methods for a securer utilization and initialization of those cryptosystems.

The application interacts with the `CryptoService` and `IOSystem` classes which function as the main façade classes over the crypto façades and `io` package respectively. `CryptoService` has the main responsibility to ensure the secure application of Linca's cryptosystems and the `IOSystem` ensures a simpler usage of the `io` package. The `CryptoService` class has other functionalities where it:

- **Simplifies error correction and exception handling.** All exceptions thrown from the crypto façades are propagated into `CryptoService` and operation exceptions are re-thrown as: `CryptoException` or `MessageNumberException`. In this way, the exception layers in Linca are simplified. These two exceptions are further discussed in section 5.5.6.

- **Provides a logical access to cryptographic service.** `CryptoService` enables developers to manage the cryptographic functionalities required by the system. Sometimes, not all cryptographic functions are required within a system. For example if an application is only required to generate a MAC the developer is only required to invoke the method to initialize `MDSystem`.

4.9 Summary

In this chapter, we have given a high level overview of Linca's internal software architecture through package dependency UML diagrams. The usage of some architectural styles and design patterns ensure Linca conforms to good design principles. These principles were discussed in section 4.8. In the next chapter, we will focus on the security principles and mechanisms that will be built into the current software architecture.

Chapter 5 - Security Design and Implementation

"You can never be too rich or too thin." --Barbara Hutton

Linca's main challenge is achieving a secure cryptographic implementation while maintaining a simple and efficient design for it to be supported on any mobile device. Several objectives are required to meet this challenge:

- **Maintenance of sound cryptographic principles.** It is no good for components within Linca to be implemented using poor cryptographic principles. Solid cryptographic principles studied from chapter 2 will be incorporated into the implementation of Linca.

- **Security through the economy of architecture.** The package dependency structure presented in chapter 4 maps the architecture of the Linca framework. Through a simple architecture, the developer can easily test, build and check each cryptographic component to make sure they are secure according to sound cryptographic fundamentals.

- **Openness.** Because the entire implementation for Linca is open source, the security of the framework does not relied upon obfuscation.

- **Efficiency.** The cryptographic components within Linca are implemented according to good cryptographic principles, yet efficiency for mobile devices is considered.

- **Ease of use.** Linca's ease of use is targeted for the developers who implement the cryptographic algorithms within the Linca framework and as well as those that use Linca as a cryptographic component after its implementation.

In this chapter, we present the security design and implementation considerations for each component within Linca so that these objectives can be achieved. If Linca is to be deployed effectively, it is required to be implemented separately for a mobile *client* and a desktop *server*. The reason behind this requirement is due to the two different environments affecting the management of symmetric and asymmetric keys. Throughout this chapter, we will make the necessary distinctions for the different implementations where applicable. The interface for *both* implementations remains the same.

5.1 Algorithm Choice and Key Size

From our literature study conducted in chapter 2, choosing the correct cryptographic algorithm and key size is very important, yet this importance might be easily overlooked. The algorithms within Linca should enhance a system's security and efficiency, with security being given at a higher priority than efficiency. It is no good to develop Linca with the inclusions of weak crypto algorithms. Table 5-1 illustrates the symmetric, asymmetric and MAC algorithms that are supported in Linca. These algorithms are based on the recommendations presented in chapter 2.

Algorithm	Key size	Description
AES_CTR (Rijndael)	256-bit	Symmetric key cryptography using the AES (Rijndael) cipher and Counter Mode.
RSAES-OAEP	2048-bit	RSA cryptosystem using the OAEP encoding scheme
HMAC_SHA-256	256-bit secret value	Message authentication using HMAC and the underlying hash function SHA-256

Table 5-1: Cryptographic algorithms used in Linca

5.1.1 Performance Analysis

We did a performance analysis in comparing the encryption and decryption time and speed using different key sizes on a particular phone model. The specifications of the mobile phone that we used to conduct this experiment are as follows:

- **Manufacturer:** Nokia 6600
- **CPU Architecture:** ARM4T
- **CPU Speed:** 104 MHz
- **Flash Size:** 6139 KB
- **RAM Size:** 379 KB
- **Memory Card:** 31066 KB

The way in which we conduct the performance analysis is by comparing the encryption/decryption duration and the encryption speed of the encryption algorithms. The ciphers used are:

- **AES128:** Plain AES cipher using 128-bit key.
- **AES256:** Plain AES cipher using 256-bit key.
- **AES256_CTR:** Counter Mode with AES cipher using 256-bit key.

- **RSAES-OAEP_2048:** RSA cipher with OAEP encoding scheme using 2048-bit key.

All the ciphers are implemented in Java. The AES is implemented according to an optimization in [15] where a total of 2 KB of static tables is used for round pre-computation. The RSA cipher is implemented using the CRT.

The encryption/decryption duration is measured by the time taken to encrypt/decrypt 512 KB of data for AES and 191 bytes for RSAES-OAEP over an average of three tries. AES128 and AES256 both encrypt and decrypt up to 16 byte blocks until 512 KB is reached. The encryption speed is measured by taking the entire data encrypted in kilobytes divided by the time taken to encrypt/decrypt and the results are summarised in Table 5-2.

Algorithm / Key size	Encryption Duration (milliseconds)	Decryption Duration (milliseconds)	Encryption Speed (KB/s)
AES128	3203	3062	200KB/s
AES256	4214	4129	121KB/s
AES256_CTR	5104	4791	100KB/s
RSAES-OAEP-2048	693	5411	0.36KB/s

Table 5-2: Performance analysis of encryption ciphers used in Linca

In terms of time, the difference is not that much between 128 and a 256-bit key if the data transferred is small, which is often the case for mobile applications. But as 3G become more prevalent and affordable, network traffic is bound to increase. It is interesting to observe that the encryption time for RSA is not that high as decryption. This will benefit the mobile device if it only encrypts the data, leaving the decryption task for a backend server. However, the reader should note that we only computed one block of data using asymmetric cryptography.

5.2 Block Cipher Mode

The CTR mode requires the counter to be unique. To ensure this property, Linca follows the concept of a nonce-generated IV (see Fundamental 7). The block cipher mode is a component within Linca, therefore generating the counter should be handled within the `mode` package so that it does not cause any dependencies. Ideally, the package should only contain an interface and a class that implements the interface (see the mode interface in the Linca API on the CD-ROM).

We recommend the counter used in Linca should consists of the following elements:

- 8-bytes of additional information.
- 4-bytes allocated for the message number.
- 4-bytes allocated for the block number.

The additional information can be a system time or a simple random number. The purpose of the random information is to ensure the *uniqueness* of the nonce within the entire encryption system. In this way, the chances of reusing the nonce with the same key are minimized. The additional information form the *mode parameter*. The message number used in the counter ensures that the counter is used once per message between two parties. Typically the message number starts at 0, however [13] suggest that it should start at 1 to avoid extra code implementations for maintaining the state of the message. Therefore it is vital that the message number should never be allowed to wrap around back to 0 as this would destroy the uniqueness property. The entire counter is encrypted in AES using 256-bit key before it is XORed with the plaintext, which ensures any structures within the nonce is destroyed.

Our recommendation of using a 4-byte message number instead of an 8-byte is justified according to the mobile application environment. For example, it is not yet feasible for the number of messages to exceed $2^{31} - 1$ per session and transmitting or storing $2^{31} - 1$ bytes of data using one[10] unique symmetric key.

5.3 Key Management

In this section, we describe the implementation of symmetric and asymmetric key managers in Linca.

5.3.1 Symmetric Key Generator

A key generation facility should be able to generate a key consisting of *n*-bit random bytes. The reason for not being a fixed value is adaptability. For example, if there will be a requirement for 512-bits in the future, the generator should be able to adapt. Currently the recommended key size is 256-bit (or 32 bytes).

From the description of a PBE based key generator in chapter 2, one might think that it will suffice for it to be used as a key generating mechanism in Linca. The most common PBE scheme in use today is PKCS#5 v2.0 [65] which is defined in PKCS#12 v1.0 under the section '*Deriving-Keys and IVs from Passwords and Salt*'. The problem relating to PKCS#5 v2.0 and PCKS#12 v1.0 is that the key length generated using PBKDF1 is bound to the length of the hash function output, which will limit the key length. The second problem is that both specifications involve ciphers and hash functions that Linca does not support. The third problem is that PBE keys cannot be used as session keys. The last problem is that the specification is rather complex to implement.

As mentioned in chapter 3, the seed used by a PRNG in cryptographic packages for mobile devices might not contain enough randomness. Therefore, the main objective of the key generator in Linca is to define an improved mechanism for gathering randomness and generating a pseudorandom stream of bits required by a cryptographically strong PRNG.

By adhering to Fundaments 2, 3 and 4, Linca's symmetric key generator has the following the requirements:

- The generator should be able to generate a cryptographically strong pseudorandom bit string that does not have a specific bound in output length, provided that it will be adequate for generating at least a 256-bit key.
- The key can be used for encrypting a communication channel or on-device data.
- If the generator is required to utilize cryptographic algorithms for its implementation, then the only algorithms it will use will be the ones supported in Linca.
- The output of the generated key must not be exposed to the `core` package so that the actual key is not exchanged over the network as this will degrade the reliability of the security system [18].
- The generator must be efficient and simple to implement.

Implementing a cryptographically strong PRNG like the Fortuna (see section 2.2.4) on a mobile device is a challenging task for two reasons. The first is that entropy sources are fairly limited and hardware random sources are not easily accessible. This is attributed to the lack of OS specific or platform APIs that provide access to these random sources. For example MIDP does not have access to determining the CPU clock speed or battery voltage. The second problem is that the

[10] The maximum integer value in Java is $2^{31} - 1$ (signed integer). Therefore the maximum number of messages and the size of the message is dependant on the message number and block number within the counter respectively. The counter is unique per key.

implementation techniques required by Fortuna are too complex for a small device. For example it is not feasible for the random sources on a mobile device to add a random event into the pool without a mediator component responsible for listening for an event from the random sources. The mediator must use the native OS API to access the underlying hardware or software random sources on the mobile device and calling the method in Fortuna to add the random data into the pool. The extra implementation of this mediator component creates additional overhead and complexity. From the objectives described in the previous paragraph, Linca's symmetric key generator is not used as a PRNG because it is encapsulated within the `crypto`, therefore it is streamlined to generate pseudorandom bits required by components within `crypto` that the application will need.

Linca's key generator is split into two parts, namely the accumulator function and the key generating function. The internal structure of the symmetric key-generating component is illustrated in Figure 5-1. Before the two components are discussed, we would like to present an evaluation of the randomness of the random sources that are used to seed the key-generating algorithm.

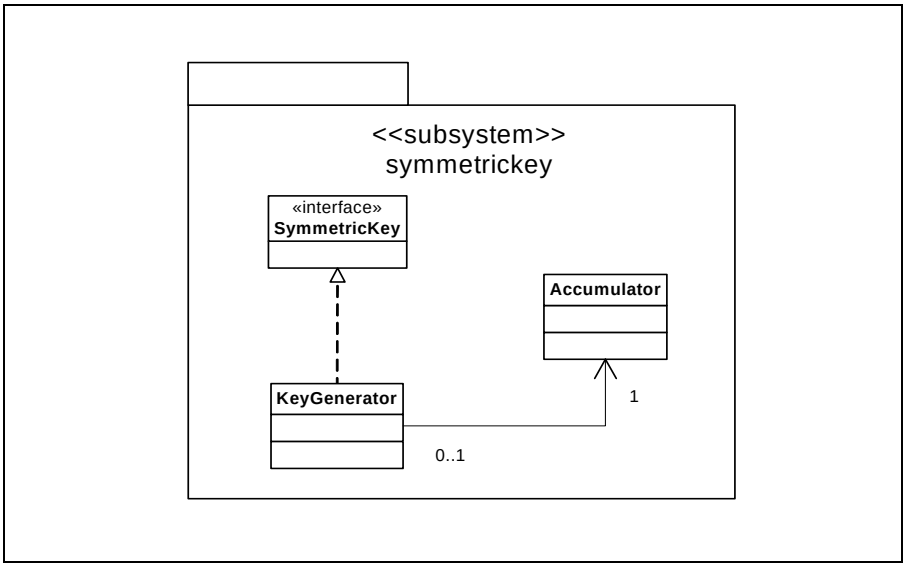


Figure 5-1: Symmetric key generator

Evaluating Random Sources

Despite the lack of genuine random sources on mobile devices, there are various reasonable random sources one can still find; in particular current mobile phones are equipped with a microphone that enables sound recordings. Other sources include accessing the total and available size of memory, and the system time. Out of these sources, sound input probably provides the best source of entropy. Designing from a Java perspective, CLDC has APIs that enable developers to gain access to system time and memory sizes. The Mobile and Multimedia Application Programming Interface (MMAPI) provides sound recordings through a built-in microphone on the device.

Sound sources recorded from a microphone on the mobile device are said to be the best source of randomness. The statistical properties of the data sets will vary to some extent depending on what frequency the radio is tuned to, which radio was used, even from time to time on the same frequency and radio as well as the noise level of the environment. To measure the randomness, we used a tool called *ENT* written by John Walker [89]. The type of tests conducted by the ENT tool are listed as follows:

- **Entropy and Optimum compression:** The information density of the contents of the file, expressed as a number of bits per character. The entropy is better if the value is close to 8-bits. The optimum compression is just another measure for information density, which indicates how much a file containing the sequence can be compressed. For a random file, the compression percentage should be as low as possible.

- **Chi-square distribution:** The chi-square distribution is calculated for the stream of bytes in the file and expressed as an absolute number and a percentage, which indicates how frequently a truly random sequence would exceed the value calculated. Knuth [31] recommends that one consider anything between 10% and 95% as acceptable for the sampled sequence.

- **Arithmetic mean:** This is simply the result of summing all the bytes in the file and dividing by the file length. The data are close to random if the value is 127.5. If the mean departs from this value, the values are consistently high or low.

- **Monte Carlo Pi value:** The Monte Carlo value of Pi is the result of a Monte Carlo algorithm to find the value of Pi. A truly random sequence should have a high degree of accuracy in the Pi value.

- **Serial Correlation:** The serial correlation measures the extent to which each byte in the file depends upon previous byte. The value can be positive or negative and for a truly random sequence, a value close to 0 is expected [31].

We conducted an experiment to determine the randomness of the sound stream when recorded through the device's microphone. The recording device is a Nokia N80 mobile phone. The recorded sound stream is sent to a server via HTTP so that the results can be gathered for statistical analysis.

The encoding format and the quality of the record settings (supported on the device) are summarised in Table 5-3. The quality of the record settings ranges from 1 (*poorest*) to 10 (*best*).

Setting[11]	Encoding Format	Sampling rate	Bit Depth	Channels
1	PCM	8000	8	1
2	PCM	8000	16	1
3	PCM	8000	16	2
4	PCM	11025	16	1
5	PCM	11025	16	2
6	PCM	16000	8	1
7	PCM	16000	16	1
8	PCM	16000	16	2
9	PCM	22050	16	1
10	PCM	22050	16	2

Table 5-3: Record settings experimented on a Nokia S60 series third edition mobile phone

We recorded *10 sound stream samples (of 5 seconds each)* for each setting and after the samples were analysed from all the recorded setting using the ENT tool, it was found that Setting 1 produced the best statistical result (see Figure 5-2). From Figure 5-2, one can see that the results can be improved in terms of randomness.

```
Entropy = 6.980312 bits per byte.

Optimum compression would reduce the size
of this 77868 byte file by 12 percent.

Chi square distribution for 77868 samples is 319630.35, and randomly
would exceed this value 0.01 percent of the times.

Arithmetic mean value of data bytes is 126.0178 (127.5 = random).
Monte Carlo value for Pi is 2.439821236 (error 22.34 percent).
Serial correlation coefficient is 0.177644 (totally uncorrelated = 0.0).
```

Figure 5-2: Statistical analysis of a five second sound stream (Lowest recording quality)

A skew correction is required to ensure the recorded sound output has a higher probability of random distribution of bits. In [22], it refers to a method where the least significant bit of each sample is gathered and turned into a stream of bits to ensure a high level of entropy. The remaining bits go through a skew correction algorithm, which ensures a good distribution of 0s and 1s in the sound stream. The skew correction algorithm used is based on *transition mapping* [26], which analyzes two bits at the same time. If the two bits are equal (00 or 11), they are discarded and if

[11] Setting 2 is the default record setting of a Nokia N80 and it might vary on other different models and makes of the mobile device.

they are not equal (10 or 01) the first bit is selected. Another method takes the parity bit over each sample and the remaining parity bits are also subjected to skew correction using transition mapping [26]. Both methods are said to work better using low-quality recorded samples. We conducted experiments on two record settings to determine the randomness of the sound stream after applying the two sound stream correction methods (see Table 5-3). We recorded *10 sound stream* samples (of 5 seconds each) using the Nokia N80 (Series 60 third edition) mobile phone and the corrected sound stream of 1024 bytes was sent to a server for statistical analysis on its randomness. The *two* record settings are based on the *lowest* and *default* recording quality setting on the Nokia N80 (Settings 1 and 2 respectively shown in Table 5-3). The recording environment is done in a noisy room (where there are people talking and music playing) and a quiet room (where only a faint ambient light sound is heard through a human ear). The purpose of comparing these 8 experiments is to demonstrate which schemes are superior across the different record settings and environments. In this way, a practical recommendation for ensuring randomness in the sound stream can be realised.

Experiment	Default Recording	Lowest-quality Recording	Noisy	Quiet	Least significant bit	Parity
A	X			X		X
B	X		X			X
C		X		X		X
D		X	X			X
E	X			X	X	
F	X		X		X	
G		X		X	X	
H		X	X		X	

Table 5-4: Experiments conducted to measure the randomness of the sound stream

We evaluated the experiments according to the programming effort, average time taken for correction and overall statistical results. For programming effort, the scheme is evaluated according to the implementation and computation complexity. The average time taken to correct the sound stream for all samples is done in ***milliseconds***. We consider a time of less than 15 seconds, which includes the recording time of 5 seconds, as tolerable. If the time exceeds 15 seconds, it is discarded and marked as not applicable (N/A) and the statistical results are not analysed. A summary of the statistical results for all the samples is also presented. The findings to our experiments are illustrated in Table 5-5.

Experiment	Programming Effort	Average time taken for correction	Overall statistical results
A	Hard	?	N/A
B	Hard	7773	There are 5 samples, each with a Chi-square distribution value outside 10% and 95%.
C	Hard	?	N/A
D	Hard	8045	There is 1 sample with a Chi-square distribution value outside 10% and 95%.
E	Easy	7748	There are 9 samples, each with a Chi-square distribution value outside 10% and 95%.
F	Easy	7477	All samples have their Chi-square distribution values outside 10% and 95%.
G	Easy	12432	There are 9 samples, each with a Chi-square distribution value outside 10% and 95%.
H	Easy	7536	There is 1 sample with a Chi-square distribution value outside 10% and 95%.

Table 5-5: Overall experimental results

The conclusion was that Experiment H has the best overall result (see Table 5-3). It was interesting to note that the lower the recording quality and the more noise level in the room gave the best result in terms of statistical result (see Figure 5-3) and speed. The statistical result shown in Figure 5-3 is the best result out of the 10 processed samples in experiment H and the rest of the results can be found on the CD-ROM. By comparing Figures 5-2 and 5-3, one can see that the correction was effective. The average time taken to correct the sound sample is 7536 milliseconds. Therefore, Experiment H concludes that the sound correction was the most effective when done in a noisy environment. From a developer's point of view, the least significant bit value in a byte can be retrieved by performing an `AND` operation on the byte value with `0x01`, which is much simpler and efficient than retrieving the parity bit.

```
Entropy = 7.813719 bits per byte.

Optimum compression would reduce the size
of this 1024 byte file by 2 percent.

Chi square distribution for 1024 samples is 245.5, and randomly
would exceed this value 50.00 percent of the times.

Arithmetic mean value of data bytes is 128.8633 (127.5 = random).
Monte Carlo value for Pi is 3.105882353 (error 1.14 percent).
Serial correlation coefficient is 0.057228 (totally uncorrelated = 0.0).
```

Figure 5-3: The best overall statistical sample taken in Experiment H

In Experiment G, the statistical result was good but slightly poorer compared to H because there are 9 samples with a Chi-square distribution value outside 10% and 95%. Aside from the poor Chi-square distribution value in the samples, the rest of the ENT tool evaluation criteria were good. The average time took to complete the correction was 12432 milliseconds, which is still within our tolerable target. The results for experiment G can be found on the CD-ROM

Accumulator

An `Accumulator` class (see Figure 5-1) collects random data such as system time, available memory size and sound streams. The output of the sound stream is corrected by the method described in Experiment H. The sources are hashed using SHA-256 to generate a 32-byte hash (see Figure 5-4). The sequences in which the sources are hashed are done randomly so that it makes the ordering of the hashed sources more difficult for the attacker to guess. The source code of the `Accumulator` is found on the CD-ROM.

The random sources for the Linca Server of the Accumulator class might be different from the one described in Figure 5-4, however the implementation principle is the same.
To restrict access to the method that implements the gathering of entropies from the random sources, the method signature should use a `protected synchronized` modifier so that the `KeyGenerator` class is the only class that is able to invoke it. The `synchronized` modifier ensures a thread-safe property.

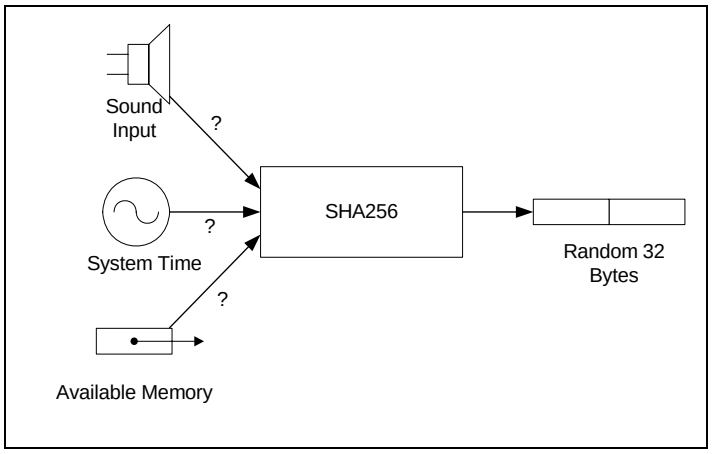


Figure 5-4: Accumulator output

The Generator

The `KeyGenerator` class is able to utilize the random source gathered by the `Accumulator`. As mentioned earlier, Linca's symmetric key generator should handle the generation of PBE keys

as well as session keys. Due to this requirement, the `KeyGenerator` class contains two methods to ensure this, namely `stretch()` and `generateKey()`. The `stretch()` method is illustrated in Figure 5-5.

The `stretch()` and `generateKey()` methods described in Figures 5-5 and 5-6 respectively are presented in pseudo code because we want to illustrate the implementations of these two methods theoretically. The reader can also use the pseudo code to map the concepts presented, into another programming language. This would be useful for implementing Linca for another platform, for example the .NET Compact Framework. The Java source code for the `stretch()` and `generateKey()` methods are implemented in the `KeyGenerator.java` class which can be found on the CD-ROM.

```
void stretch

   input:  byte []      b  //Hashed random sources from the Accumulator
           StringBuffer t  //User's authentication data
           integer      q  //Number of hash iterations


   output: byte [] s  //A seed used to generate cipher blocks

{  //begin void
     s0 ← 0     //make sure s0 is empty
     r ← 0      //r is an global variable of type integer
     if q > 0 {
        r  ←  q
        for i  = 1….r {
          si  ←  SHA-256(si-1 || t || b)
        }
        return sr

     else {
        do {
         si  ←  SHA-256(si-1 || t || b)
         r  ←  r + 1
        } until (1000 milliseconds) //stretch duration
        return sr
     }
} //end void
```

Figure 5-5: The `stretch` method

The purpose of the `stretch()` method is to increase the security of a limited-entropy password or pass-phrase and to generate a temporary key for use in the `generateKey()` method.

The two inputs *t* and *q* requires further explanation. The input *t* is used to authenticate the user. This can be a password, pass-phrase, security token and so on. In addition to the entropy sources supplied by the `Accumulator`, *t* can be used as another entropy source. By supplying *t*, the final generated key can be reconstructed. When *t* is used, it is vital to add a *salt* value (denoted by *b*) in order to ensure the randomness is kept in case *t* does not have enough entropy. In doing so, a random *s* can be generated using *b* because it is a fresh random string of bytes. The stretching of *t* is done *q* times by hashing *t* and *b* using SHA-256 with the previous result. The || denotes hashing of t and using SHA-256. The size of *r* is chosen such that computing *s* takes 200-1000 milliseconds on a user's equipment [13] and is therefore a never a fixed value. Therefore by not having a fixed *r* value, *r* can increase accordingly to the computation speed. This concept applies to the mobile device as computing speed of the mobile device will increase in the future as well. In this way, it makes password attacks more difficult on the mobile device because the attacker has to compute *r* hash computations for each password.

The output of the stretch function basically is a seed *s* of size 256-bits that will be used as a *temporary key* in the `generateKey()` method illustrated in Figure 5-4.

```
//Before the function is called, make sure of the following:
 assert that the generator is seeded
 assert c == null OR c == getModeParam() //where c is the mode param
  assert s.length >= 32  //the seed s is derived from stretch()

void generateKey


   output: byte [] k //Pseudorandom string in a byte array size of 32

 { //begin void
      z ← e  //initialize z as a blank plaintext byte array of size 32
      //Apply symmetric key encryption methods in Linca
      installMode(SC, c) //Symmetric cipher instance is denoted by SC
      initMode(true, s, r)
      processBytes(z, k)
      return k
 } //end void
```

Figure 5-6: The `generateKey` method

Unfortunately by keeping *r* in proportion with the computing speed has a limitation within a mobile application environment due to the different CPU speeds of a mobile device and a desktop machine. For example, if the key is generated by a desktop machine for encrypting a piece of data that will be sent to a mobile device, the decryption time on the mobile device will be unacceptable because *r*

would be too large for the re-computation of *s*. To circumvent this problem, the developer should reduce the stretch duration for the Linca Server implementation to compute *r* in proportion to the capability of the mobile device. For example, for *r* to have a value just over 1000, a stretch duration of 50 milliseconds[12] is required for the Linca Server implementation.

The purpose of the `generateKey()` method is to generate a pseudorandom key using AES256_CTR. The benefit of having `KeyGenerator` depending on `mode` is so that should AES be replaced with a securer standard, the encryption function used in `generateKey()` would be updated as well. The methods `installMode()`, `initMode()` and `processBytes()` are methods within `mode` that applies encryption to an initial blank array of bytes *k*. If *c* is `null`, then `mode` will self generate *c*. The result is an encrypted *k* of 256-bits (32-bytes). In providing extra security, the output *k* is never used in the application, but is used within the `core` package. Therefore there should be a method implemented in `KeyGenerator` to return a set of parameters to ensure that the secret key could be re-generated by a legitimate communicating entity. These parameters are collectively called the *key generating parameter* and are grouped together in this particular order:

- The iterative hashing count *r*.
- The seed *s* generated by the `stretch()` method.
- The counter *c* generated by the mode that is used within `KeyGenerator`.

To regenerate the secret key, one should be in possession of the key generating parameter and authentication data. The same ordering should be preserved so that processing these parameters can be standardised.

In order to determine the randomness of the byte stream produced by `generateKey()`, we generated a 15 MB sample by extracting the code from Linca into a separate method and executed that method by seeding the block cipher mode with a hashed value of the system time[13] using SHA-256. The reason for separating the code from Linca is because the `generateKey()` method is not accessible to the application. The execution platform is J2SE because it is impractical to generate a 15MB file using a mobile phone or on the emulator. The generated byte stream output is saved into a file. The content of the file was evaluated using the ENT tool and the DIEHARD tool. The result is presented in Figure 5-7. From Figure 5-7, one can conclude Linca's key generator have passed

[12] Linca Server was tested on an Intel Duo Core 2.0GHz processor.
[13] The seed used in this experiment does not have to be securely gathered using the `Accumulator` because the generator is not required to generate a secret key. The generator is required to generate random data of 15MB in size.

the test. The results generated by the `generateKey()` on a desktop machine is suitable for this analysis because the seed used is hashed in the same way as the `Accumulator` class (see Footnote 13). The result from the DIEHARD tool can be found on the CD-ROM.

```
Entropy = 7.999988 bits per byte.

Optimum compression would reduce the size
of this 15728640 byte file by 0 percent.

Chi square distribution for 15728640 samples is 260.55, and randomly
would exceed this value 50.00 percent of the times.

Arithmetic mean value of data bytes is 127.5300 (127.5 = random).
Monte Carlo value for Pi is 3.141439819 (error 0.00 percent).
Serial correlation coefficient is -0.000045 (totally uncorrelated = 0.0).
```

Figure 5-7: Statistical analysis of a 15MB file produced by the `generateKey()` method

Security Issues

An important security concern is how the attacker can get hold of the key generating parameters, mode parameter and the authentication. For the Linca Server, vulnerability occurs when the key generating parameter, mode parameter and the generated secret key are swapped to virtual memory on the hard disk from the memory. This can happen to the heap memory in the Java Virtual Machine as well. The most effective solution for overcoming this problem is to encrypt the entire disk partition or to ensure that the server is protected behind a firewall. For mobile devices running the Symbian OS, the memory management engages in a linear memory model for all running programs [83]. Virtual memory in this context does not mean that Symbian OS makes use of program swapping to hard disks, but that all programs during linking, locating and execution appear at the same virtual address; and that they cannot access each other's memory.

If the user's authentication data is entered on the device through the keypad, a keylogger application could possibly obtain this authentication data. Keylogger applications can be distributed in applications downloaded from unknown sources or contain mobile viruses. It remains to be seen how the mobile platform copes with viral threats.

The record store system for MIDP ensures that the record generated by a Java application is only accessible by itself [9, 33]. This means that no other Java applications running on the device may write or delete a record store. If the key generating and mode parameters are required to be stored in the record, then only the application that generated those parameters is able to access it.

Even if the application is memory and persistent storage safe, the user's authentication data must not be stored on the device and it must be at least 8 characters in length. For instance, if an attacker can guess the password easily the encrypted data stored on the mobile device is jeopardised. In addition to securing the authentication data, Linca only accepts the authentication data as a `StringBuffer` so that the developer can erase the buffer content once the secret key is generated. Clearing this information as soon as possible makes a heap-inspection attack from outside the virtual machine more difficult [77].

5.3.2 Asymmetric Key Loader

Instead of generating public and private key pairs on handheld devices, we recommend to load keys from a particular secure source. These sources can be from the user's subscriber information module (SIM), an encrypted persistent key storage on a flash disk, a secure database and so on. By default Linca retrieves the public key from a X.509 version 3 digital certificate. The format of the certificate is in binary and is based on the ASN.1 notation. This notation defines how to specify the contents while the encoding rules define how the content is translated into binary form. The binary encoding of the certificate is defined using Distinguished Encoding Rules (DER), which is based on the more general encoding rules called Basic Encoding Rules (BER). Both BER and DER provide a platform-independent method of encoding objects such as certificates and messages for transmission between devices and applications.

The other encoding format aside from DER is Base64. This is an encoding method developed for use with Secure/Multipurpose Internet Mail Extensions (S/MIME) which is a popular standard method for transferring binary attachments over the internet. Base64 is not supported in Linca.

The Loader

The implementation of the asymmetric key loader in Linca is dependent on:

- The asymmetric cipher supported.
- The certificate structure and encoding method.
- The location of the stored certificate.

The loader in Linca should be implemented to load ASN.1 DER encoded certificates with RSA public key. The RSA cipher itself is implemented using CRT for a faster processing speed (see Fundamental 8).

The asymmetric key loader consists of a main interface `AsymmetricKey` that is implemented by `KeyLoader`, which contains the code that enables the loading of the asymmetric key pairs. The `RSAPrivateKeyImpl` and `RSAPublicKeyImpl` classes wrap the mathematical constructs of RSA key pairs and are returned by `KeyLoader`. The `Other_Classes` denotes the classes required by `KeyLoader` that enables the loading of the key pairs from specific locations. The reader should note that it is not compulsory for the loader to load the private key especially on a mobile device where the private key might not be stored for security reasons.

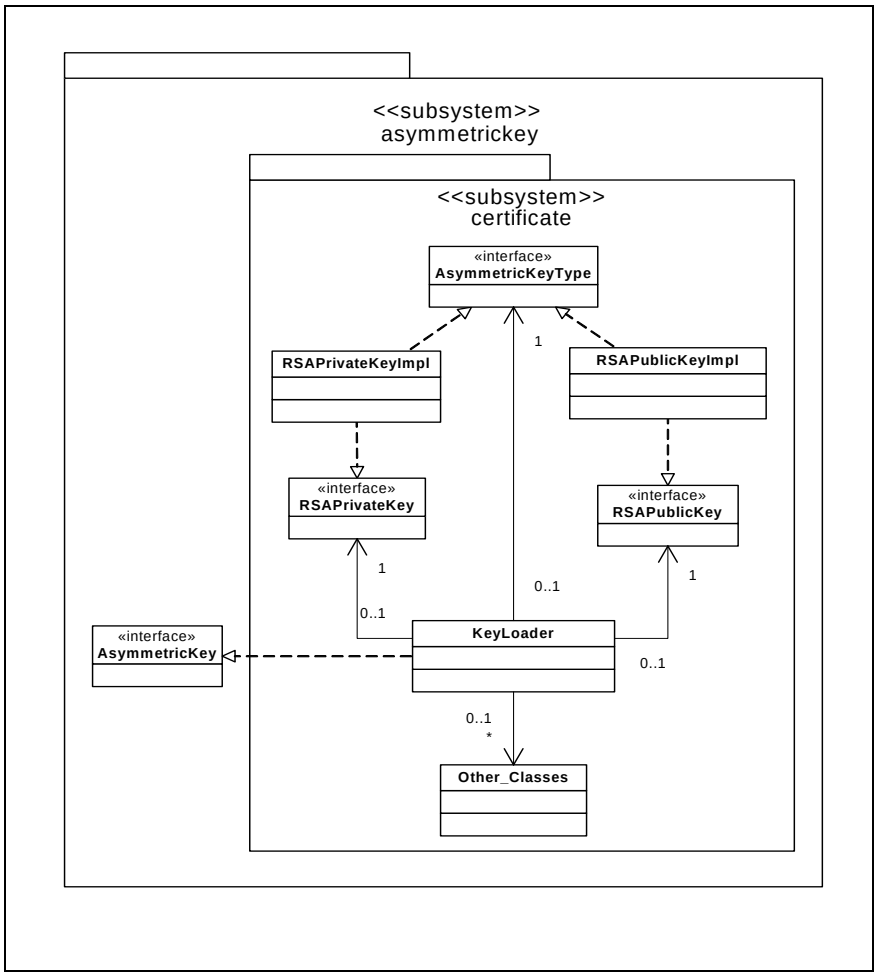


Figure 5-8: Asymmetric key loader

Loading the key pairs from a server machine is quite different from a handheld device because the key storage location can vary. For example, the key pairs can be loaded from a secure database on a server or from a SIM card on the mobile phone. The choice of the location is most likely dependent on the security policy.

Security Considerations

The most frequently used key in asymmetric key cryptography on a mobile device is the public key that belongs to the server which the device is communicating to. It is seldom that a private key belonging to either the client or the server is used because it could lead to a higher security risk when stored on the phone. However, private keys are useful when the mobile client is required to provide a digital signature or ensuring data integrity.

There are three strategies on how to use public and private keys (key pair) on a handheld device:

1. Generate fresh key pairs on the device.
2. Load the key pairs from a SIM or a separate smart card.
3. Store the X.509 certificate file in a DER format and encrypt the corresponding private key file within the MIDlet suite.

In the first strategy, it is not practical to generate the key pair on handheld devices due to resource limitations. The securest method would be to load the key pair from a smart card, but unfortunately, not all mobile cryptographic APIs are able to access the SIM. The most probable solution is to encrypt the actual private key using a PBE key prior to deployment and store it as a file along with the certificate within the MIDlet suite.

Linca should support at least one of these three protocols for loading the key pairs on a mobile device:

1. **File:** This is the default method for loading the key pairs. The public key is stored within the X.509 certificate as a DER format file. The corresponding private key conforms to ASN.1 and is encrypted as binary file (see Figure 5-9).

2. **APDU:** Mobile applications can communicate with a smart card using a protocol based on Application Protocol Data Units (APDUs). This protocol is defined by ISO 7816-4 and described in the Java Card Development Kit documentation [82].

3. **JCRMI:** Mobile applications can communicate with a smart card using a protocol based on Java Card Remote Method Invocation (JCRMI). JCRMI is a distributed computing protocol, which means it allows applications that a re not located on the card to use objects that exists on the card. Applications use a remote interface to work with an object on the card.

At present only the file protocol is implemented.

The Linca desktop implementation of the key loader contains an extra protocol to load the key pair:

- **Keystore:** This protocol enables the server application to load the key pair from a secure key store. For example, such a keystore could be the Keystore facility provided by J2SE.

Table 5-6 lists the key loading protocols with example URIs for loading the key pair.

Protocol	URI Format	Example
File	**file:**<location>	**file:**/rsacert2048.cer
APDU	**apdu:**<slot>;<target>	**apdu:**0;target=SAT
JCRMI	**jcrmi:**<slot>;<AID>	**jcrmi:**0;AID=A0.0.0.67.4.7.1F.3.2C.3
Keystore (private key)	**keystore:**<location>;<alias>;<storepass>;<keypass>	**keystore:**.keystore;mystor;yI8T4;Se4&1t
Keystore (public key)	**keystore:**<location>;<alias>;<storepass>	**keystore:**.keystore;johnnystor;yI8T4;

Table 5-6: Key loading protocol with example

For the file protocol, the private key has to be encrypted using Linca prior to deployment on the mobile device by following the format illustrated in Figure 5-9.

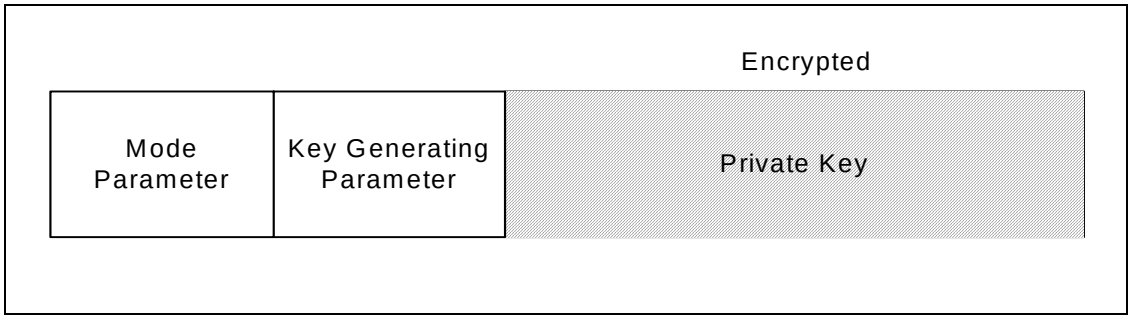


Figure 5-9: Encrypted format of a private key used in Linca

The developer should take note of the ordering of the mode parameter and key generating parameter when appending these values with the encrypted private key. This is to ensure the correctness in the parsing of the fields for decryption.

The last important security consideration during file key loading is that ensuring the public key is loaded from authentic sources so that man-in-the-middle attacks cannot be performed. The most effective way to ensure the certificate is not tampered with is by signing the MIDlet suite that contains the certificate by the issuer of the application. In this way, the entire suite is verified by the Java Application Manager against the root certificate prior to installation.

During APDU operation, the `<slot>` indicate the slot number where the smart card is inserted. The `<target>` indicates the card identification number. The `<target>` can either be an Application Identifier (AID) or (U)SIM Application Toolkit ((U)SAT). An AID uniquely identifies a smart card application using a 5 to 16 hexadecimal bytes where each byte value is separated by a "." character.

During JCRMI operation, the `<slot>` is used in the exact manner as for APDU. The only difference is that JCRMI only supports AID. If the URI is the same for accessing the certificate and private key using either APDU or JCRMI, then the same URI for the respective protocol can be passed into the asymmetric key cryptosystem's initialization parameters for loading the public and private keys. For more information on the APDU and JCRMI specifications the reader is encouraged to refer to [80].

When the Keystore protocol is used, the format follows the standard of the `keystore` mechanism provided by J2SE. The `<alias>` is the logical name of the key store. The `<storepass>` and `<keypass>` are passwords for protecting the keystore and the private key within the keystore respectively. The `KeyLoader` class should provide auxiliary security by:

- Validating that the certificate is not expired since the issue date.
- Ensuring that the key supported is at least 2048-bits.
- Ensuring that the certification version is 3.

5.4 Digital Signature

The digital signature algorithm in Linca has the following criteria:

- The strength of the algorithm relies on the asymmetric cipher supported within Linca.
- The algorithm does not have any patent involvement and should be efficient to implement.
- The generated signature will have a random mapping for each message m.

There is a good technique implemented for digital verification by using the RSA cipher [13]. We modified the algorithm to suit the operations within Linca. The idea behind this algorithm is to use a pseudorandom mapping to expand $H(m)$ (where H is the hash function and m the message) to a random number s in the range $0,\ldots,n-1$. The signature s is then computed as $s^{1/e} \pmod{n}$. This algorithm is presented in Figure 5-10.

```
void MsgToRSACipher

   input:  PubKey  v <n> //RSA public key v, with n as the modulus
           byte [] m    //Message to be converted to a value s

   output: byte [] s

 { //begin void
     byte [] x ← ϵ //length is the maximum input size for RSA
     byte [] z ← ϵ //length is the maximum input size for RSA
     byte [] t ← SHA-256(m) //Seed with the hash of the message
     byte [] c ← ϵ //make sure the counter is a blank array
     Integer k ← ⌊ log2n ⌋ // floor function

     //Generate a random mapping of the message
     installMode(SC, c) //Symmetric cipher instance denoted by SC
     initMode(true, t, 1) //set the message number to 1 as default
     processBytes(z, x) //symmetric key encryption
     initAsymmetricKeyCipher(true, v) //init the asymmetric cipher
     s ← processBytes(x) //RSA operation mapped to x mod 2^k
      return s
  } //end void


 void generateSignature

    input:  PriKey   w<n, d, e>//RSA private key components
            byte []  m   //Message to be signed

    output: byte []  σ  //Signature for m

 {//begin void
     byte [] s ← MsgToRSACipher(n, m)
    initAsymmetricKeyCipher(true, w) //init the asymmetric cipher
    σ ← processBytes(x) //RSA operation mapped to s^1/e mod n
    return σ
 } //end void

void VerifyRSASignature

  input: PubKey  v<n, e> //RSA public key with e as the exponent
         byte [] m      //Message that is suppose to be signed
         byte [] σ      //Signature on the message

   output: true if signature matches false otherwise

 {//begin void
    s ← MsgToRSANumber(n, m)
    initAsymmetricKeyCipher(true, v) //init the asymmetric cipher
    byte [] b ← processBytes(σ) //RSA operation mapped to σ^e mod n
     if(b == s) {return true} else {return false}
 } //end void
```

Figure 5-10: Signature Generating Algorithm using RSA

As shown in Figure 5-10 the algorithm will generate random pairs of (s, $s^{1/e}$) for a set of messages m_1, m_2,…m_i. This is because m is hashed with a hash function H in order to provide a key for initializing the block cipher mode. The mode parameter can be initialized to 0 for an easier implementation. The encryption process provides a random mapping. As long as H is secure, $H(m)$ can only be affected by trial and error. Anyone can create pairs of the form (s, $s^{1/e}$) for random s values, so this provides no new information that helps the attacker to forge a signature. However, for any particular message m, only someone (for example entity A) who knows the private key can compute the corresponding (s, $s^{1/e}$) pair, because s must be computed from $H(m)$, and then $s^{1/e}$ must be computed from s, which requires the private key. Therefore, anyone who verifies the signature knows that entity A must have signed it.

The limitation of this algorithm is that it is not widely adopted in any standard compared to the likes of RSASSA-PSS, RSASSA-PKCS1-v1_5 and the DSA for example. Aside from this limitation, the signature-generating algorithm in Figure 5-10 is secure and easier to implement (see Recommendation 6 and Fundamental 15).

5.5 Obfuscation Support

Because mobile devices have limited memory resources, applications designed for mobile devices should be as compact as possible. An obfuscator is a useful tool for minimizing the size of an application. Obfuscators, originally designed to foil attempts to reverse engineer compiled bytecode, can now perform any combination of the following functions [30]:

- Renaming classes, member variables, and methods to more compact names.
- Removing unused classes, methods, and member variables.
- Inserting illegal or questionable data to confuse decompilers.

There is a wide spectrum of obfuscators [34] that can be used to shrink the size of compiled classes in a MIDlet suite JAR. However, there are limitations to bear in mind prior to obfuscating. A major limitation that influences the implementation of Linca is how the obfuscators handle `Class.forName(variable)` where `variable` contains the class name. The reason for this limitation is that it is generally not possible to determine the possible class name values because the referenced classes are required to be preserved in the shrinking phase so that the string arguments are properly replaced in the obfuscation phase. The disadvantage of providing a string literal in `forName()` is that dynamic loading of classes by specifying the class names within a file cannot

take place. Therefore, the developer has to load classes by providing the string literal within the `forName()` method. This affects the `ModuleLoader` class within the `core` package.

5.6 Security provided by the Core Package

The methods within the `CryptoService` class enables a more effective way to apply cryptography at the application layer by preventing the developer from accessing complex lower level interfaces within the `crypto` package. In this way, the developer is encouraged to access simpler methods within `CryptoService` that promote a more secure and simpler technique to applying cryptography.

5.6.1 Main Initialization

In order to utilize the `CryptoService` class, a static method within `CryptoService` can be used to return its instance (see Figure 5-11).

```
public static CryptoService getCryptoService(boolean isRemote)
```

Figure 5-11: CryptoService initialization method

The `isRemote` parameter indicates whether Linca is being applied in a mobile networking environment. This has an impact on how the message number is applied during the encryption and decryption of messages (see sections 5.6.3 and 5.6.4). One `CryptoService` instance is able to initialize the symmetric and asymmetric cryptosystems *once* only. In this way, each `CryptoService` instance is able to manage its own secret and private keys separately, thus making them thread safe.

5.6.2 Message Authentication

It is difficult to anticipate what kind of information the developer would use in order to make the message unambiguous (See Fundamental 13). Therefore our approach is to request the developer to supply this information through the `extras` parameter along with the `secret` during initialization. The input to the `extras` parameter is versatile as it accepts an `Object` array consisting of types such as byte arrays or Strings. The security values in the extras parameter can be used only once. If it is reused a `CryptoException` is thrown. To change the message meaning

within the same instance, the `resetMessageMeaning(Object [])` method is used. In this way, the developer can still use the same `CryptoService` instance to create MACs for multiple messages.

```
public void initMACComponent(byte [] secret, Object [] extras) throws CryptoException{..}
public void generateMAC(Object [] data, byte [] mac_out) throws CryptoException{..}
public void generateMAC(byte [] data, byte [] mac_out, int count) throws CryptoException{..}
public void resetMessageMeaning(Object [] extras) throws CryptoException{..}
public void compareMAC(Object [] data, byte [] mac_in) throws CryptoException{..}
public void compareMAC(byte [] data, byte [] mac_in) throws CryptoException{..}
public int getMACOutputSize() throws CryptoException{..}
public String getMACSystemName() throws CryptoException{..}
```

Figure 5-12: Hash function methods

The recommended length of the `secret` should at least be 32-bytes, however it can be an arbitrary length since `secret` is hashed 10 times using SHA-256 before the value is used by the HMAC function. Assuming the key generating parameters is protected, a typical secret value might consist of the key generating parameter appended with the user's PIN.

The MAC function will be dominantly used within the mobile application because the one-way hashing is applied within the internal components of Linca that handles symmetric key and digital signature generation. In this way, it becomes unnecessary to use one-way hash functions within the application layer.

5.6.3 Symmetric Key Cryptography

Linca does not accept any secret key during the symmetric key cryptosystem initialization. In this way, the secret key cannot be misused in the application layer. The symmetric key cryptosystem is initialized by the user identification, key generating parameter and mode parameter. The `usr_id` is the information a user uses to authenticate him/herself. This information could be a passphrase, password, PIN number and so on. If the symmetric key cryptosystem is initialized for the first time by the initiating entity, the `key_param` and `mode_param` parameters can be initialized to `null`. This informs the symmetric key generator and mode components to self generate the required parameters respectively. The key generating and mode parameters can be obtained from the `getSymmetricKeyParam(byte [])` and `getModeParam(byte [])` methods.

```
public void initSymmetricKeyCipherComponent(StringBuffer usr_id, byte [] key_pram, byte [] mode_param)
                                            throws CryptoException{..}

public void symmetricKeyEncryption(int msgnum, byte [] plaintext, byte [] ciphertext)
                                   throws CryptoException, MessageNumberException{..}

public void symmetricKeyDecryption(int msgnum, byte [] ciphertext, byte [] plaintext)
                                   throws CryptoException, MessageNumberException{..}

public void getModeParam(byte [] mode_param_out) throws CryptoException{..}

public void getSymmetricKeyParam(byte [] key_param_out) throws CryptoException{..}

public void getModeParamSize() throws CryptoException{..}

public void getSymmetricKeyParamSize() throws CryptoException{..}

public int getSymmetricKeyCipherOutputSize (int len, boolean forEncryption) throws CryptoException{..}

public String getSymmetricKeyCryptosystemName() throws CryptoException{..}
```

Figure 5-13: Symmetric key cryptosystem methods

During encryption and decryption it is important to keep the message number unique for each operation. The message number is passed into the `initMode(boolean, byte [], int)` method defined by the `BlockCipherMode` interface in `mode` (see the Linca API available on the CD-ROM). In order to prevent the message number from wrapping back to 0 or going out of sequence, `CryptoService` has an internal ticking mechanism that increments the message number starting from 1. If any message number that wraps back to 0 or went out of sequence during encryption or decryption, a `MessageNumberException` will be thrown.

Recall that when `CryptoService` is initialized, a `boolean` value is passed via `isRemote`. If the value passed is `true`, it means Linca is applied in an application that requires secure communication over the mobile network. If the value is `false`, then Linca is applied to secure data locally on the mobile device. When Linca is applied to secure data over the network, the message sequence number is managed in a remote manner (see Figure 5-14). By using this technique, the messages sent across the network can be protected against replay attacks.

When Linca is used to secure data locally on the device, the message sequence is managed only during encryption since the mode parameter is required to be unique during encryption (see Figure 5-15). Therefore it is not necessary to manage the message number during decryption. The developer should store the corresponding message numbers for each encrypted message, the key generating parameter and mode parameter persistently on the device so that the message can be decrypted at a later stage.

```
//CryptoService instance initialized by the initiator
…
StringBuffer pswd = new StringBuffer(auth);
CryptoService cs1 = CryptoService.getCryptoService(true);
cs1.initSymmetricKeyCipherComponent(pswd, null, null);

//the initiator encrypts two messages to be sent over the network
…
cs1.symmetricKeyEncryption(1, plaintext1, ciphertext1);
cs1.symmetricKeyEncryption(2, plaintext2, ciphertext2);
…
cs1.symmetricKeyDecryption(1, ciphertext1, output1);   //Illegal!

/*Only the receiver who has initialized its CryptoService with the received key and mode
parameter is able to decrypt the ciphertext */
…
…
StringBuffer pswd = new StringBuffer(auth);  //the receiver knows the password
CryptoService cs2 = CryptoService.getCryptoService(true);
cs2.initSymmetricKeyCipherComponent(pswd, keyparam, modeparam); //using key and mode parameters
received
cs2.symmetricKeyDecryption(1, ciphertext1, output1);
cs2.symmetricKeyDecryption(2, ciphertext2, output2);
…
//if the receiver decides to send an encrypted message, it will resume from message number 3
cs2.symmetricKeyEncryption(3, plaintext3, ciphertext3);
```

Figure 5-14: Managing message number over the network

```
//CryptoService initialized for persistent data encryption on the device
…
StringBuffer pswd = new StringBuffer(auth);
CryptoService cs1 = CryptoService.getCryptoService(false);
cs1.initSymmetricKeyCipherComponent(pswd, null, null);

//the user encrypts two messages to be stored persistently
…
cs1.symmetricKeyEncryption(1, plaintext1, ciphertext1);
cs1.symmetricKeyEncryption(2, plaintext2, ciphertext2);

//at a later stage, the messages are retrieved
…
cs1.symmetricKeyDecryption(1, ciphertext1, output1);
cs1.symmetricKeyDecryption(2, ciphertext2, output2);
```

Figure 5-15: Managing message number locally on the device

5.6.4 Asymmetric Key Cryptography

The asymmetric key cryptosystem can be initialized in two ways, namely: public key initialization and key pair initialization (public key and its corresponding private key). The public key initialization uses the `initAsymmetricKeyCipherComponent(String, boolean)` method which accepts the certificate location and a `boolean` condition to denote whether the public key within the loaded certificate should be used to verify digital signatures. In this way, the developer can explicitly manage which key pairs are used for encryption or signing (see Fundamental 11). Therefore, if `processSignature` is set to `true`, it automatically initializes the signature

processing components within Linca. The format of the certificate location and private key location is dependent on the key loading protocol (see Table 5-6). Because the public key is dominantly used on the mobile device, the public key initialization method is mostly called instead of the key pair initialization.

The key pair initialization has extra parameters that require the location input of the private key, as well as the identification password required to decrypt the private key (see section 5.3.2). This method is useful if the loader could access the private key within the SIM of the mobile client.

The message number is incremented universally during the encryption and decryption of messages between asymmetric and symmetric key cryptosystems.

The encrypted/decrypted result is returned as a byte array reference instead of a copy (as is the case with symmetric key encryption/decryption) is due to the difficulty in pre-determining the size of the output for different asymmetric ciphers.

```
public void initAsymmetricKeyCipherComponent(String cert_loc, boolean processSignature)
                                        throws CryptoException{..}

public void initAsymmetricKeyCipherComponent(String cert_loc, String prikey_loc, StringBuffer usr_id,
                                          boolean processSignature) throws CryptoException {..}

public [] byte asymmetricKeyEncryption(int msgnum, byte [] plaintext)
                                    throws CryptoException, MessageNumberException{..}

public [] byte asymmetricKeyDecryption(int msgnum, byte [] ciphertext)
                                    throws CryptoException, MessageNumberException{..}

public int getAsymmetricKeyInputSize(boolean forEncryption) throws CryptoException{..}

public String getAsymmetricKeyCryptosystemName() throws CryptoException{..}

public String getCertificateInfo(int item) throws CryptoException{..}
```

Figure 5-16: Asymmetric key cryptosystem methods

The certificate structure can be printed by using the `getCertificateInfo(int)`, where the parameter item is used to indicate which information to print. For a X.509 certificate the value returned is the printable version of the distinguished name (DN) from the certificate. An X.509 DN is a set of attributes where each attribute is a sequence of an object ID and a value. For string comparison purposes, the following rules define a strict printable representation [80]:

- There is no added white space around separators.
- The attributes are in the same order as in the certificate; attributes are not reordered.

- If an object ID is in the table below, the label from the table will be substituted for the object ID, else the ID is formatted as a string using the binary printable representation above.
- Each object ID or label and value within an attribute will be separated by a "=" (Unicode U+003D), even if the value is empty.
- If value is not a string, then it is formatted as a string using the binary printable representation above.
- Attributes will be separated by a ";" (Unicode U+003B)

The order of the information in which the certificate is printed, follows the X.509 format (see Table 5-7).

Item	Certificate Information
0	Print all
1	Version
2	Serial Number
3	Signature Algorithm
4	Issuer
5	Not Before
6	Not After
7	Subject

Table 5-7: Printing order of certificate information

5.6.5 Digital Signature

The most fundamental aspect of handling asymmetric key pairs in Linca is enabling the developer to choose which pairs are going to be used for digital signature processing or asymmetric key cryptography or both.

```
public void initSignerComponent(String cert_loc) throws CryptoException{..}

public void initSignerComponent(String cert_loc, String prikey_loc, StringBuffer usr_id)
                            throws CryptoException{..}

public void generateSignature(byte [] data, byte [] signature_out) throws CryptoException{..}

public boolean verifySignature(byte [] data, byte [] signature_in) throws CryptoException{..}

public int getSignatureOutputSize() throws CryptoException{..}
```

Figure 5-17: Signature processing methods

If `processSignature` is set to `false` during asymmetric key cryptosystem initialization, then the asymmetric keys are only used for encryption and/or decryption. The developer can separately initialize another public key or key pairs for digital signature processing using `initSignerComponent(String)` and `initSignerComponent(String, String, StringBuffer)` respectively. A `CryptoException` will be thrown if the asymmetric key cryptosystem and signer component are both initialized to use key pairs. The format of the key loading protocol is exactly the same as described in Table 5-6.

5.6.6 Error Checking and Exception Handling

The classes within the `core` enable the developer to utilize the crypto components more easily, and if the method is used incorrectly, exceptions `CryptoSystemException` or `MessageNumberException` are thrown. Further parameter validation by the components within `crypto` and `io`, provides a more reliable error tracking when their exceptions are propagated to `CryptoService`. Figure 5-18 describes the format of an error message thrown by a Linca system.

```
Error at <Class Name><Method Name>: <Error Message>
```

Figure 5-18: Error message format

The components within `crypto` and `io` classes are responsible for validating the parameter inputs according to their defined interface methods. For example the component within `mode` will do a check on the length of the mode parameters during initialization. Exceptions raised by the `crypto` and `io` components are: `InvalidArgumentException`, `RuntimeException` and `IOException`. These exceptions are re-thrown and are caught in `CryptoService`, which are ultimately thrown as a `CryptoException`.

5.7 Networking

The `IOSystem` class in the core package offers a simplex networking facility that offers a more versatile way to create networking connections in a mobile application environment. A new networking connection can be created by calling the `newIOConnection(String String)` method. The URI takes in the form of the protocol type that the current implementation within `io` package offers. The name of the protocols can be queried using the `getProtocolTypes()`

method. The protocol URI is dependant on what is supported on the platform or operating system. Linca does not have to support all of these protocols, however HTTP should at least be implemented within the `io` package.

```
public void newIOConnection(String URI, String protocolType) throws CryptoException{..}
public void send(int msgnum, byte [] message StringBuffer usr_id) throws IOException{..}
public [] byte receive() throws CryptoException{..}
public void closeIOConnection() throws CryptoException{..}
public String getProtocolTypes() throws CryptoException{..}
```

Figure 5-19: IOSystem class methods

When an encrypted message is composed, it is sent via the `send(int, message)` method over the selected communication protocol. The `send()` method should enable the sending of the message number, message length and the encrypted message in three consecutive byte streams over the communication protocol. The `receive()` method should do a check on the received message numbers and message length so that they are within the positive bounded value of the integer representation[14]. If the integers are within the legal bound limits, the message number and encrypted message are appended together and returned to the caller. The receiver can do further processing with the returned message comparing the received message number and its expected message number. If both message numbers do not match, then the receiver can discard the received message without having to decrypt it, thus saving processing time especially for a mobile device.

Different hardware platforms have different representation of integers. Therefore the implementation is targeted towards the supported platform. For example, the Java platform has a big-endian representation.

After the sender and receiver have finished communicating, the connection can be closed via the `closeIOConnection()` method.

The networking facility within Linca is targeted for the mobile device rather than the server.

[14] The positive integer bound in Java is 2^{31}-1.

5.8 Summary

In this chapter, we presented a balance between security and efficiency during the design and implementation of cryptographic components within Linca. Through our guidelines, we have shown that Linca offers solid security by applying effective cryptographic principles within the internal cryptographic components. Solid cryptographic fundamentals were applied in the justification of cryptographic algorithm choice and their initialization, key management, and signature generation and verification. While solid security principles are applied with Linca's components, we also presented several guidelines on efficient implementation where possible to suit the mobile computing environment. Lastly, the methods within the `CryptoService` and `IOSystem` classes illustrated the possibilities in applying effective cryptography in an application. The usages of these methods are illustrated in chapter 7.

Chapter 6 - External API and Design Evaluation

"All exact science is dominated by the idea of approximation." --Bertrand Russell

An evaluation is conducted on the Linca framework by means of:

1. Demonstrating the security improvements Linca has over Bouncy Castle API and Secure and Trust Service API.
2. Evaluating the architecture and design using a set of qualitative measurements.

6.1 Security Improvements

Security improvements are demonstrated by comparing Linca against BC and SATSA. The critiques on security improvements are based on the cryptographic fundamentals studied in chapter 2 and chapter 5.

6.1.1 Generating MACs

```
1   // We know the pin
2   byte [] pin = "34265".getBytes();
3
4   // The message to MAC
5   byte [] toMAC = "Balance: $345,467,66:".getBytes();
6
7   // Initialize the HMac
8   HMac digest = new HMac(new SHA256Digest());
9   CipherParameters param = new KeyParameter(pin);
10  digest.init(param);
11
12  // Create the MAC
13  digest.update(toMAC, 0, toMAC.length);
14  byte [] mac_out = new byte[digest.getMacSize()];
15  digest.doFinal(mac_out, 0);
16
17  return mac_out;
```

Figure 6-1: Generating MAC using Bouncy Castle API

BC has a HMAC implementation, however the interface does not enforce the authenticating of additional data with the message (see Figure 6-1 and Fundamental 13). The only way to authenticate the message further with additional data is to call the `update` method and placing those data into that method. The problem with this approach in our opinion is that it gives the developer an option to not use additional data; therefore a developer might just call the `update` method once on the message only.

```
1   // We know the pin
2   byte [] pin = "34265".getBytes();
3
4   // The message to MAC
5   byte [] toMAC = "Balance: $345,467,66".getBytes();
6
7   //Initialize the MAC
8   MessageDigest digest = MessageDigest.getInstance("SHA-1");
9   digest.update(pin, 0, pin.length);
10  digest.update(toMAC, 0, toMAC.length);
11
12  //Create the buffer to hold the MAC
13  byte [] mac = new byte[20];
14  digest.digest(mac, 0, mac.length);
15
16  return mac;
```

Figure 6-2: Generating a one-way hash digest as a MAC using Secure and Trust Service API

SATSA-CRYPTO also does not guide the developer to authenticate additional data and it is more complex to use because the developer has to allocate the exact buffer size to hold the digest (see Line 13 of Figure 6-2). This means that the developer has to be aware of the characteristics of the hash function. Unfortunately SATSA-CRYPTO does not support any MAC algorithm thus violating Fundamental 6 and 12. In order to prevent message tampering over the communication network, the developer has to use CMS, which is more computationally expensive compared to using a MAC.

```
1   // We know the pin
2   byte [] pin = "34265".getBytes();
3
4   // The message to MAC
5   byte [] toMAC = "Balance: $345,467,66".getBytes();
6
7   // Initialize the MAC
8   CryptoSystem cipher = CryptoSystem.getCryptoSystem(false);
9   cipher.initMACComponent(pin, extras);
10
11  // Initialize the buffer and create the MAC
12  byte [] mac_out = new byte[cipher.getMACOutputSize()];
13  cipher.generateMAC(toMAC, mac_out, 1);
14
15  return mac_out;
```

Figure 6-3: Generating a MAC using Linca

Linca overcomes the limitations of the previous APIs by allowing the developer to specify the additional data during MAC initialization (see Line 9 of Figure 6-3). Further security enhancements were discussed in section 5.6.2.

6.1.2 Symmetric Key Cryptography

```
1   // The plaintext to encrypt
2   byte[] toEncrypt = "PIN Request: OK".getBytes();
3
4   // Initialize the cipher with a random AESIv
5   BufferedBlockCipher cipher = new PaddedBufferedBlockCipher(new SICBlockCipher(new
6                                                          AESEngine()));
7   ParametersWithIV piv = new ParametersWithIV(new KeyParameter(AESKey), AESIv);
8   cipher.init(true, piv);
9
10  // Create the buffer to store the result
11  byte [] ciphertext = new byte[cipher.getOutputSize(toEncrypt.length)];
12
13  // Do the encryption
14  int len1 = cipher.processBytes(toEncrypt, 0,  toEncrypt.length, ciphertext, 0);
15  cipher.doFinal(ciphertext, len1);
16
17  // Initialize the cipher to do decryption
18  cipher.init(false, piv);
19  byte [] plaintext = new byte[cipher.getOutputSize(ciphertext.length)];
20
21  // Do the decryption
22  int len2 = cipher.processBytes(ciphertext, 0,  ciphertext.length, plaintext, 0);
23  cipher.doFinal(plaintext, len2);
```

Figure 6-4: Symmetric key encryption and decryption using Bouncy Castle API

Symmetric key encryption in BC is very versatile as it offers a wide variety of ciphers and block cipher modes. In this way, older encryption algorithms used in legacy server systems could be compatible with newer front-end mobile applications that support such an algorithm. However, under this versatility certain complexities could hide under this versatility, relating to the much-needed security fundamentals. This complexity could entail the decisions on which key size and encryption cipher to use as well as their initialization criteria (see Fundamentals 2, 4, 5 and 7).

SATSA-CRYPTO API contains only a few symmetric ciphers and block cipher modes [81], and it does not support the CTR mode. Assuming the developer made the correct cipher and key size choices based on the fundamentals discussed in section 2.4.4, BC and SATSA-CRYPTO has limitations to guide the developer in applying the ciphers securely and effectively. These limitations are illustrated in Lines 7-8 in Figure 6.4 and Lines 46-55 in Figure 6.5, which demonstrate the initialization of the symmetric cryptosystem using the key and IV. The key and IV should be generated from a cryptographically strong PRNG.

```
1   // Assuming we have retrieved the password
2   String passStr = "de23ISS5";
3   byte [] ciphertext = null;
4
5   // The plaintext to encrypt
6   byte[] toEncrypt = "PIN Request: OK".getBytes();
7
8   //We have to lengthen the password into 16 bytes
9   while (passStr.length() < 16 ){
10        passStr = passStr.concat(" ");
11   }
12   passwd = passStr.getBytes();
13
14  // Create a key from the password
15  Key key = new SecretKeySpec(passwd, 0, passwd.length, "AES");
16
17  // Initialize the cipher
18  Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
19
20  // Create the IV
21  IvParameterSpec iv = null;
22
23  // Add 2 bytes to encode the length of the plaintext as a short value
24
25  byte[] plaintextAndLength = new byte[toEncrypt.length + 2];
26  plaintextAndLength[0] = (byte)(0xff & (toEncrypt.length >> 8));
27  plaintextAndLength[1] = (byte)(0xff & toEncrypt.length);
28
29  // Build the new plaintext
30  System.arraycopy(toEncrypt, 0, plaintextAndLength, 2, toEncrypt.length);
31
32  // Calculate the size of the ciperthext considering the padding
33  int blocksize = 16;
34  int ciphertextLength = 0;
35  int remainder = plaintextAndLength.length % blocksize;
36  if (remainder == 0) {
37       ciphertextLength = plaintextAndLength.length;
38  }
39  else {
40       ciphertextLength = plaintextAndLength.length - remainder + blocksize;
41  }
42
43  byte[] cipherText = new byte[ciphertextLength];
44
45  // AESIv is randomly generated
46  iv = new IvParameterSpec(AESIv, 0, AESIv.length);
47
48  // Reinitialize the cipher in encryption mode with the given key
49  if(iv != null) {
50      cipher.init(Cipher.ENCRYPT_MODE, key, iv);
51    }
52  else {
53        cipher.init(Cipher.ENCRYPT_MODE, key);
54  }
55
56  // Do the encryption
57  cipher.doFinal(plaintextAndLength, 0, plaintextAndLength.length, cipherText, 0);
58
59  return cipherText;
```

Figure 6-5: Symmetric key encryption using Secure and Trust Service API

As mentioned in chapter 3, the problem with the PRNG in BC is that it is seeded using the system time and SATSA-CRYPTO does not have a PRNG. Both APIs shoulder the IV management onto the developer, which could result in additional complexity. In Figures 6-4 and 6-5, the IVs are randomly generated which violates Fundamental 7. Due to the reason that the developer has to manage the IV and there is a possibility where this IV might be reused. For example, BC's block cipher mode initialization can accept a `KeyParameter` object because it implements `CipherParameters`. In this way, BC defaults the IV to 0 which could be reused with the same key. This problem is illustrated with the code in Figure 6-6.

```
1   public void init(boolean forEncryption, CipherParameters params)
2                   throws IllegalArgumentException {
3
4       this.encrypting = forEncryption;
5
6       if(params instanceof ParametersWithIV) {
7           ParametersWithIV ivParam = (ParametersWithIV)params;
8           byte[] iv = ivParam.getIV();
9          System.arraycopy(iv, 0, IV, 0, IV.length);
10          reset();
11          cipher.init(true, ivParam.getParameters());
12        }
13  }
14
15  public int processBlock(byte[] in, int inOff, byte[] out, int outOff)
16          throws DataLengthException, IllegalStateException {
17
18         cipher.processBlock(counter, 0, counterOut, 0);
19
20
21        // XOR the counterOut with the plaintext producing the cipher text
22
23        for(int i = 0; i < counterOut.length; i++) {
24          out[outOff + i] = (byte)(counterOut[i] ^ in[inOff + i]);
25        }
26
27        int carry = 1;
28        for(int i = counter.length - 1; i >= 0; i--) {
29            int x = (counter[i] & 0xff) + carry;
30            if(x > 0xff) {
31                carry = 1;
32            }
33            else {
34                carry = 0;
35            }
36            counter[i] = (byte)x;
37        }
38
39        return counter.length;
40  }
41
42  public void reset() {
43
44        System.arraycopy(IV, 0, counter, 0, counter.length);
45        cipher.reset();
46  }
```

Figure 6-6: Implementation of the counter mode in Bouncy Castle API

In Line 11 of Figure 6-6, if the `init()` method is not initialized with a `ParametersWithIV` object, then the counter will default to a blank array of 16-bytes (the same size as the underlying block cipher if AES is used for example). Although it does not subject to a counter IV problem, there is a possibility where the key can be reused with the same counter during on-device data encryption. If the `init()` method is initialized with a `ParametersWithIV` object, the responsibility of managing the IV is shouldered onto the developer. SATSA has a similar technique where the IV is wrapped in the `IvParameterSpec` class.

BC and SATSA-CRYPTO allows an AES cipher to be initialized up to 256-bit key size. The key factory class expects a key material (such as a byte array of a string in the form of a password) to be initialized into a key. This byte array can be from 128 to 256-bit, however a password might not be of this length. Therefore the solution is to pad the password (with spaces for example) to make up for the remaining bytes. The key factory creates only a static key according to the key material. This can be a limitation when a session key used for encrypting a specific network channel is required.

SATSA-CRYPTO is the most complex API to use because the developer has to allocate the correct buffer size to hold the encrypted and decrypted text. This complexity is illustrated in Lines 23-43 and Lines 18 – 20 in Figures 6-5 and 6-7 respectively.

```
1   // Create a key from the password
2   Key key = new SecretKeySpec(passwd, 0, passwd.length, "AES");
3
4   // Initialize the cipher
5   Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
6
7   // This IV is the same used for encryption
8   IvParameterSpec iv = new IvParameterSpec(AESIv, 0, AESIv.length);
9
10  cipher.init(Cipher.DECRYPT_MODE, key, iv);
11
12  // toDecrypt is the ciphertext
13  byte[] decrypted = new byte[toDecrypt.length];
14
15  // Decrypt the cipher text
16  cipher.doFinal(toDecrypt, 0, toDecrypt.length, decrypted, 0);
17
18  // Calculate the length of the plaintext
19  int plainTextLength = ((decrypted[0] << 8)  | (decrypted[1] & 0xff));
20  byte[] finalText = new byte[plainTextLength];
21
22  // Decode the final text which results in the original plaintext
23  System.arraycopy(decrypted, 2, finalText, 0, plainTextLength);
24
25  return finalText;
```

Figure 6-7: Symmetric key decryption using Secure and Trust Service API

```
1   // The plaintext to encrypt
2   byte[] toEncrypt = "PIN Request: OK".getBytes();
3
4   // Assume the user has entered a password
5   StringBuffer pswd = new StringBuffer("de23ISS5");
6
7   // We assume we are not working over a network connection..
8   CryptoSystem cipher = CryptoSystem.getCryptoSystem(false);
9
10  // Initialize Symmetric cipher System...
11  cipher.initSymmetricKeyCipherComponent(pswd, null, null);
12
13  // Create a buffer to hold the ciphertext and do encryption
14  byte [] ciphertext = new byte[cipher.getSymmetricKeyCipherOutputSize(toEncrypt.length, true)];
15  cipher.symmetricKeyEncryption(1, toEncrypt, ciphertext);
16
17  // Create a buffer to hold the plaintext and do decryption
18  byte [] plaintext = new byte[cipher.getSymmetricKeyCipherOutputSize(ciphertext.length, false)];
19  cipher.symmetricKeyDecryption(1, ciphertext, plaintext);
```

Figure 6-8: Symmetric key encryption and decryption using Linca

The security mechanism for IV management and key generation in Linca was discussed in sections 5.2 and 5.3.1. Figure 6-8 demonstrates that applying symmetric cryptography in Linca is much simpler and more effective compared to BC and SATSA-CRYPTO. Because Linca's key generator is not used as a PRNG and the `generateKey()` method is only invoked by Linca's internal component, the risk of acquiring the key generator's internal state by an attacker is minimal (see Fundamental 3). The developer has only to manage the message sequence, which is an extra support for data authentication. The password entered by the user is susceptible to being obtained by a keylogger software (see Lines 5 and 11 in Figure 6-8). However this is the same problem with SATSA-CRYPTO when the user has to enter the PIN to access the smart card.

On-device Data Security

BC has a PBE encryption scheme that can be used to encrypt persistent data on a mobile device (see Figure 6-9). SATSA-CRYPTO can regenerate a static key using the password and the code in Figures 6-5 and 6-7 can be reused for encrypting on-device data. Linca in this case is the most complex API to use as the developer has to store the key generating and mode parameters along with the message numbers of each byte array stored (see chapter 7). These parameters have to be stored on the device so that the symmetric key cryptosystem can be reinitialized with the exact key and mode initializing parameters.

```
1   // Initialize the PBE classes and with a random 256-bit salt and a password
2   PBEParametersGenerator generator = new PKCS12ParametersGenerator(new SHA256Digest());
3   generator.init(PBEParametersGenerator.PKCS12PasswordToBytes(passwd), salt, 1024);
4
5   //Generate a 256 bit key with a 128 IV
6   ParametersWithIV key = (ParametersWithIV) generator.generateDerivedParameters(256, 128);
7
8   BufferedBlockCipher cipher = new PaddedBufferedBlockCipher(new SICBlockCipher(new
9                                                   AESEngine()));
10  cipher.init(true, key);
11
12  // Create a buffer to hold the ciphertext and do the encryption
13  byte [] result = new byte[cipher.getOutputSize(toEncrypt.length)];
14  int len = cipher.processBytes(toEncrypt, 0,  toEncrypt.length, result, 0);
15
16  cipher.doFinal(result, len);
17
18  return result;
```

Figure 6-9: PBE using Bouncy Castle API

6.1.3 Asymmetric Key Cryptography

BC has classes that can read an ASN.1 encoded X.509 certificate and private key. In this way, the key pairs can be used for encryption and decryption (see Figures 6-10 and 6-11). The certificate and private key are stored as a binary file on the mobile phone, which can be read into a byte stream object. The strength of BC is that it supports a wide variety of asymmetric algorithms. However as with symmetric ciphers, the developer would have to make decisions on which key size and encryption/encoding schemes to use (see section 2.4.5). For example, there is no explicit guidance for developers to apply an encoding function over a RSA cipher. Although Figures 6-9 and 6-10 illustrates the usage of an encoding function in Lines 24 and 30 respectively, the RSA cipher can also be used independently thus violating Fundamental 9.

Complexity is another problem. As it is shown in Figures 6-10 and 6-11, the developer has to make a substantial number of method calls to parse the correct mathematical elements from the X.509 certificate and private key files in order to construct the RSA key pair.

```
1   // Read in the certificate
2   ByteArrayInputStream bIn = new ByteArrayInputStream(readFile("RSA2048Cert.cer"));
3
4   // Define the ASN1 data constructs
5   ASN1InputStream dIn = new ASN1InputStream(bIn);
6   DERSequence  seq = (DERSequence)dIn.readObject();
7
8   // Define X509 certificate
9   X509CertificateStructure obj = new X509CertificateStructure(seq);
10  TBSCertificateStructure tcs = obj.getTBSCertificate();
11  DERBitString subId = tcs.getSubjectUniqueId();
12
13  // Get the public key elements
14  SubjectPublicKeyInfo ski = obj.getSubjectPublicKeyInfo();
15  DERObject doe = ski.getPublicKey();
16  Enumeration e = ((ASN1Sequence)doe).getObjects();
17  BigInteger mod = ((DERInteger)e.nextElement()).getPositiveValue();
18  BigInteger pubExp = ((DERInteger)e.nextElement()).getPositiveValue();
19
20  // Wrap the public key elements into the RSA key parameter type
21  RSAKeyParameters pubKey = new RSAKeyParameters(false, mod, pubExp);
22
23  // Define the OAEP encoding and wrap the RSA cipher and SHA256 digest
24  OAEPEncoding cipher = new OAEPEncoding(new RSAEngine(), new SHA256Digest());
25
26  // Do the encryption and return the ciphertext
27  cipher.init(true, pubKey);
28  return cipher.processBlock(toEncrypt, 0, toEncrypt.length);
```

Figure 6-10: Asymmetric key encryption using Bouncy Castle API

```
1   // Read in the private key from a file
2   ASN1InputStream str =  new ASN1InputStream(new
3                       ByteArrayInputStream(readFile("rsaprivatekey.ser")));
4   DERObject doj = str.readObject();
5   ASN1Sequence asq = ASN1Sequence.getInstance(doj);
6   DEREncodable dec = asq.getObjectAt(2);
7   DEROctetString oct = (DEROctetString) dec;
8   ByteArrayInputStream boct = new ByteArrayInputStream(oct.getOctets());
9   ASN1InputStream doct = new ASN1InputStream(boct);
10  DERObject finalget = doct.readObject();
11  Enumeration e = ((ASN1Sequence) finalget).getObjects();
12
13  // First get version
14  BigInteger  version = ((DERInteger)e.nextElement()).getValue();
15
16  // Now get the private key elements
17  BigInteger mod = ((DERInteger)e.nextElement()).getValue();
18  BigInteger pubExp = ((DERInteger)e.nextElement()).getValue();
19  BigInteger priExp = ((DERInteger)e.nextElement()).getValue();
20  BigInteger p1 = ((DERInteger)e.nextElement()).getValue();
21  BigInteger p2 = ((DERInteger)e.nextElement()).getValue();
22  BigInteger q1 = ((DERInteger)e.nextElement()).getValue();
23  BigInteger q2 = ((DERInteger)e.nextElement()).getValue();
24  BigInteger coeff = ((DERInteger)e.nextElement()).getValue();
25
26  // Wrap the private key elements into the RSA key parameter type
27  RSAPrivateCrtKeyParameters priKey = new RSAPrivateCrtKeyParameters(mod, pubExp, priExp, p1,
28                                                                     p2, q1, q2, coeff);
29  // Do the decryption and return the plaintext
30  OAEPEncoding cipher = new OAEPEncoding(new RSAEngine(), new SHA256Digest());
31  cipher.init(false, priKey);
32  return cipher.processBlock(toDecrypt, 0, toDecrypt.length);
```

Figure 6-11: Asymmetric key decryption using Bouncy Castle API

```
1   // Wrap the public key in the X509 key spec
2   X509EncodedKeySpec pks = new X509EncodedKeySpec(kRSAPublicKey);
3
4   // Generate the public key
5   KeyFactory kf = KeyFactory.getInstance("RSA");
6   PublicKey publicKey = kf.generatePublic(pks);
7
8   // Create the buffer to hold the ciphertext
9   byte[] ciphertext = new byte[512];
10
11  // Initialize the cipher and create the ciphertext
12  Cipher cipher = Cipher.getInstance("RSA");
13  cipher.init(Cipher.ENCRYPT_MODE, publicKey);
14  cipher.doFinal(toEncrypt, 0, toEncrypt.length, ciphertext, 0);
15
16  return ciphertext;
```

Figure 6-12: Asymmetric key encryption using Secure and Trust Service API

SATSA-CRYPTO's asymmetric key cryptography goes as far as encryption as there is no support for decryption using the private key. Prior to encryption, the developer has to encode the public key used by the `X509EncodedKeySpec` class according to the Subject's public key information defined by the X.509 standard. Unfortunately, SATSA-CRYPTO does not provide any classes to enable such an encoding; therefore the developer has to resort to other methods defined outside the scope of SATSA.

SATSA-CRYPTO supports the RSA cipher only, however it is not known if the actual implementation of the library on the mobile phone will support an encoding function in the future. According to the SATSA specification [81], the RSAES-PKCS1-V5 is the recommended encoding function, however through our study in section 2.4.3, it is recommended to use RSAES-OAEP.

```
1   // Assuming the user has supplied the password to decrypt the private key file..
2   StringBuffer keypin = new StringBuffer("jkLw23Bnme");
3
4   // We assume we are working over a network connection..
5   CryptoSystem cipher = CryptoSystem.getCryptoSystem(true);
6
7   // Initialize Asymmetric cipher System and using the same keypair for signing...
8   cipher.initAsymmetricKeyCipherComponent("file:/rsa2048cert.cer",
9                                           "file:/privatekeyenc.ser", keypin, true);
10
11  return cipher.asymmetricKeyEncryption(1, toEncrypt);

// At the receiving end, the initialization process is the same as above
..
..
n   return cipher.asymmetricKeyDecryption(1, toDecrypt);
```

Figure 6-13: Asymmetric key encryption and decryption using Linca

Linca's asymmetric key cryptographic operations were discussed in section 5.6.4. As can be see in Figure 6-13, Linca promotes a logical way on applying asymmetric key cryptography. The

developer does not have to worry about complex method calls relating to certificate management. Linca supports a minimum RSA key size of 2048-bits and can support X.503 version 3 certificates (see Fundamental 10 and 14). The RSA cipher itself is implemented with CRT to improve the decryption speed (see Fundamental 8).

Unfortunately, Linca does not support PKI in the sense that it does not verify the certificate it loads with a CA. The authenticity of the certificate is verified when the application is verified by the application manager of the mobile device (see chapter 7).

6.1.4 Digital Signature Generation and Verification

```
1   // The message to sign
2   byte [] msg = "Account 455355 transacted".getBytes();
3
4   // Initialize the necessary algorithms for PSS
5   SHA256Digest dig = new SHA256Digest();
6   RSAEngine cipher = new RSAEngine();
7
8   // Initialize the PSS algorithm
9   PSSSigner signer = new PSSSigner(cipher, dig, 64);
10
11  // Sign the message using the private key
12  signer.init(true, priKey);
13  signer.update(msg, 0, msg.length);
14
15  byte [] signature = signer.generateSignature();
16
17  // To verify the message, have the public key and message ready
18  signer.init(false, pubKey);
19  signer.update(msg, 0, msg.length);
20
21  boolean isValid = signer.verifySignature(signature);
```

Figure 6-14: Signature generation and verification using Bouncy Castle API

BC supports a number of digital signature algorithms and Figure 6-14 demonstrates the usage of the PSS signer. Unfortunately, there are patent issues surrounding the PSS algorithm, therefore it is very difficult to follow the recommendation in PKCS#1 v2.1. The key pair derivation used for digital signing shares the same complexity as it is illustrated in Figures 6-10 and 6-11.

Digital signing in SATSA-CRYPTO is handled via the `Signature` class and `CMSMessageSignatureService`. Figure 6-15 illustrates the usage of the `Signature` class, which only handles signature verification. The public key has to be encoded in the same way as described in section 6.1.3.

```
1   // Wrap the public key in the X509 key spec
2   X509EncodedKeySpec pks = new X509EncodedKeySpec(kRSAPublicKey);
3
4   // Generate the public key
5   KeyFactory kf = KeyFactory.getInstance("RSA");
6   PublicKey publicKey = kf.generatePublic(pks);
7
8   // Initialize the Signing class using the public key
9   Signature signature = Signature.getInstance("SHA1withRSA");
10  signature.initVerify(publicKey);
11  signature.update(msg, 0, msg.length);
12
13  // Have the signature ready for verification
14  return signature.verify(kSignature);
```

Figure 6-15: Signature verification using Secure and Trust Service API

Digital signing can only be done on a string text using the `CMSMessageSignatureService` class. This can be a limitation if a non-String data type is required to be signed. Unfortunately the digital signature service in SATSA-CRYPTO is limited to signature verification since signature generation is not supported.

```
//Assume the initialization follows from the code in Figure 6-13
..
..

1   // The message to sign
2   byte [] msg = "Account 455355 transacted".getBytes();
3
4   // To generate a signature
5   byte [] signature = cipher.generateSignature(msg);
6
7   // To verify a signature
8   boolean isValid = cipher.verifySignature(msg, signature);
```

Figure 6-16: Digital signature generation and verification using Linca

Linca offers an easier way to generate and verify signatures. It also supports an effective way to allow developers to manage key pairs that are different in the usage criteria. Figure 6-16 illustrates the usage of the same key pair used in asymmetric key encryption for digital signature generation and verification. The initialization method in Line 8-9 of Figure 6-13 will be initialized to a `false` if another key pair is used for signature verification. In this way, a separate key pair used for signing can be initialized in the `initSignerComponent` method (see Figure 6-17).

```
1   //explicitly assign the key pairs used for digital signature
2   cipher.initSignerComponent("file:/signingcert.cer", "file:/sgnprikey.ser", keypin2);
```

Figure 6-17: Using another key pair for digital signature

By explicitly allowing the developer to assign the usage criteria of the asymmetric keys, Fundamental 11 can be achieved more effectively instead of the ad hoc usage of the key pairs in BC and SATSA.

6.2 Qualitative Analysis

There are various ways in which a system such as Linca can be assessed and some of the concepts taken from [35, 60, 76] have been merged to suit our evaluation. These criteria are presented in section 6.2.1 and the metrics for evaluating the design is presented are section 6.2.2. The results are presented in section 6.2.3.

6.2.1 The Criteria

Dynamic extensibility: The ability of the framework to facilitate an easy inclusion of new algorithms.

Reusability: The ability of the framework to provide reusability at the application layer.

Flexibility: Designing for flexibility (also known as adaptability) means actively anticipating changes that a design may have to undergo in the future and preparing for them.

Coupling: A measure of the extent to which interdependencies exist between software modules. An important design principle is to reduce coupling.

Effective design: Linca is a software component and it should separate the concerns of applying cryptography from the application. In this way, the application becomes easier to implement. The architecture and design implementation for Linca Mobile presented in chapters 5 and 6 will be analyzed using the metrics described in section 6.2.2.

Maintainability: An important internal quality of software that measures the extent to which the software can be modified at the lowest possible cost, that is, maintenance can be made easier.

Anticipating the obsolescence: Anticipating obsolescence means planning for the evolution of the technology or environment so that the software will continue to run or can be easily changed.

Substitutability: The ability to interchange internal components of the framework by a developer without any expensive refactoring or re-engineering.

Imperturbability: This refers to the impact of change of algorithms on the system.

Comprehensibility: The degree of comprehension a developer will require on the functionality of any particular cryptographic algorithm.

Portability: Portability ensures software have the ability to run on as many platforms as possible.

Ease of testing: The ability of the framework to enable an easier testing of cryptographic algorithms.

6.2.2 Metrics for Evaluating the Design

To be able to quantitatively measure a design it has to be possible to calculate certain metrics from the application architecture. This section presents a handful of useful metrics based on [41] for evaluating the Linca's architecture and design. The motivation for using these metrics is because they operate on the package level rather than class level and Linca is best evaluated on the number of closely related classes which are coupled into one package and reused together.

Instability Metric

In a software system, packages that are hard to change are normally the ones that many other packages depend on. When many components depend on one particular package *X*, it is difficult to make changes to *X* without making changes to the other components as well. Therefore *X* is an independent package because it has no external influence to make it change. Such a package is considered to be stable. On the other hand, if a package *Y* is dependent on many components, a change in any of these components will affect *Y*. Therefore *Y* is dependent, which results in an instable package.

The instability metric describes the stability of a package in terms of dependencies that leave or enter the package. To calculate instability we need to define two terms for different kinds of dependencies:

- C_a or afferent couplings is the number of classes outside this package that depend on the classes inside this package.

- C_e or efferent couplings is the number of classes inside this package that depend on classes outside this package.

Using these metrics instability I can be calculated using the following equation:

$$I = \frac{C_e}{C_a + C_e}$$

Equation 6-1: Instability equation

This metric has the range [0, 1]. A 0 value for I indicates a maximally stable package while 1 indicates a maximally instable package.

Abstraction Metric

A stable package should also be abstract so that its stability does not prevent it from being extended. On the other hand, an instable package should be concrete since its instability allows the concrete code within it to be easily changed. Thus if a package is to be stable, it should also consist of abstract classes so that it can be extended. Stable packages that are extensible are flexible and do not overly constrain the design. The abstraction metric is a measure of the abstractness of a package by calculating the ratio of abstract classes in a package to the total number of classes in the package.

The package abstractness equation consists of:

- N_c – The number of classes in the package.
- N_a – The number of abstract classes in the package.
- A – Abstractness.

Thus given an equation:

$$A = \frac{N_a}{N_c}$$

Equation 6-2: Abstractness equation

The *A* metric ranges from 0 to 1. Zero implies that the package has no abstract classes at all. A value of 1 implies that the package contains nothing but abstract classes.

The Main Sequence

Relationship between abstractness and instability can be described using a two dimensional graph that has abstractness on the vertical axis and instability on the horizontal axis (see Figure 6-18). The packages that are maximally stable and abstract are found at the upper left (0,1), while the maximally instable and concrete are at the lower right (1,0). Not all packages can fall into one of these positions. Packages have degrees of abstraction and stability.

Since not all packages sit either at the ideal points (0,1) or (1,0), there is a locus of points on the A/I graph that infers to the areas where a package should not be (i.e zones of exclusion).

The area around (0,0) is called the *zone of pain*. A package in the zone of pain is a highly stable and concrete package. Such a package is not desirable because it is rigid. It cannot be extended because it is not abstract and it is difficult to change because of its stability. However a package can be in the zone of pain provided that they are non-volatile. An example of such a package could be a utility package where they are unlikely to be changed.

The area around (1,1) is called the *zone of uselessness*. This location is undesirable because it is maximally abstract and yet has no dependents. Such packages are useless.

Abstractness and instability should form a balance with each other in a well-designed architecture. This means that packages should be away from the zones of exclusions as far as possible. The locus of points that is maximally distant from each zone is the line that connects (1,0) and (0,1). This line is known as the main sequence (denoted by a green line in Figure 6-18).

Therefore, a well-designed package should be completely instable and concrete, completely stable and abstract, or lie somewhere very close to the main sequence.

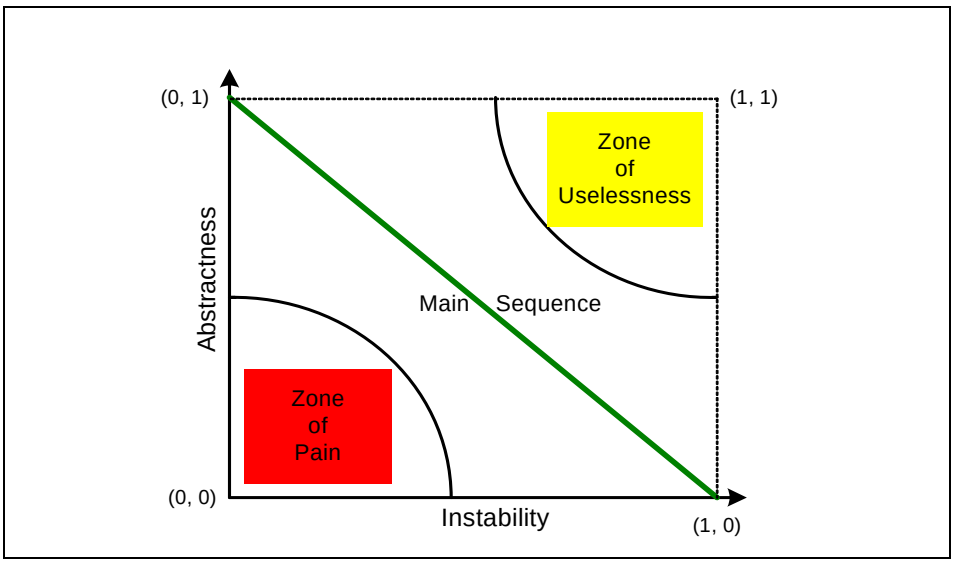


Figure 6-18: The A-I graph with zones of exclusion

If it is desirable for packages to be on or close to the main sequence, a metric can be determined to measure how far away a package is from this ideal. Therefore a normalized distance D' can be calculated as:

$$D' = |A + I - 1|$$

Equation 6-3: Normalized distance equation

D' can range from [0,1]. A D' value of 0 indicates that the package is directly on the main sequence, while a 1 indicates that the package is as far away as possible from the main sequence.

6.2.3 Analysis

Dynamic extensibility: The Linca connectivity component can incorporate multiple types of network connections. This is made possible by the generic connection interface and factory design pattern for creating different connections.

Reusability: In section 4.4, we have illustrated the dependencies of the various crypto components on each other. Such dependencies generate some form of reuse amongst certain components. For example the symmetric key generator reuses `mode` and `skcipher` to generate random keys. Because Linca itself is a framework, it can be used as a software component in any mobile enterprise applications.

Flexibility: Linca aims to be flexible enough to be tailored to the developer and system's needs – in terms of providing sound cryptographic fundamentals. The `akcrypto` package allows the developer to adapt to changes made to the asymmetric cryptosystem without affecting other packages. However, the developer has no flexibility to change the type of algorithm or key size to use in an application. This reduces the chances of poor decisions a developer might make when cryptography is applied. The `ModuleLoader` class does not support dynamic class loading though variable names due to obfuscation limitations (see section 5.5).

Coupling: Linca reduces content[15] coupling [35] by encapsulating all instance variables within the `crypto` components as *private* and only allowing the `core` component to initialize them. The `core` component also reduces coupling between the `crypto` and `io` components and application layer.

Effective design: By using the metrics presented in section 6.2.2, Linca showed a good design result because most of the packages lie close to the main sequence (see Table 6-3). The `core` package has an instability of 1 which means that it is an instable package. This is expected because the `core` package is a façade over the `crypto` and `io` package and will be dependent by them. However, the `core` package does not have any abstract or interface classes, thus allowing the concrete code within it to be easily changed. In this way, a main sequence of 0 is achieved. The `authentication` and `skcipher` packages have an instability value of 0 because they do not depend on any packages. The `math` and `util` packages have a main sequence of 1, which suggests poor quality of design. However, because they have classes within them that serve as utilities and are unlikely to change, this instability is acceptable.

Linca's design is also compared against the design of BC and SATSA and the results of the metrics are presented in Tables 6-1 and 6-2 respectively. The values in N_C, N_A, C_A, C_E, I, A and D are determined by a tool called *Structural Analysis for Java* [25]. The values in Table 6-2 are based on the SATSA RI 1.0 from Sun and it may vary according to the implementations done by different mobile phone vendors. Linca Server implementation will not be evaluated because the values in the metric will vary depending on the underlying architecture for key management.

[15] Content coupling occurs when one component surreptitiously modifies data that is internal to another component.

Package Name	N_C	N_A	C_A	C_E	I	A	D'
org.bouncycastle.crypto	26	15	452	6	0.01	0.58	0.41
org.bouncycastle.crypto.agreement	4	0	0	60	1	0	0
org.bouncycastle.crypto.digests	17	0	21	13	0.38	0	0.62
org.bouncycastle.crypto.encodings	3	0	3	46	0.94	0	0.06
org.bouncycastle.crypto.engines	27	0	8	214	0.96	0	0.04
org.bouncycastle.crypto.generators	19	0	3	201	0.99	0	0.01
org.bouncycastle.crypto.io	4	0	0	16	1	0	0
org.bouncycastle.crypto.macs	8	0	4	74	0.95	0	0.05
org.bouncycastle.crypto.modes	9	0	10	80	0.89	0	0.11
org.bouncycastle.crypto.paddings	8	1	10	33	0.77	0.13	0.10
org.bouncycastle.crypto.params	45	0	296	99	0.25	0	0.75
org.bouncycastle.crypto.signers	8	0	3	168	0.98	0	0.02
org.bouncycastle.asn1	53	4	2389	11	0.005	0.08	0.92
org.bouncycastle.asn1.cmp	4	1	3	32	0.91	0.25	0.16
org.bouncycastle.asn1.cms	26	2	11	406	0.97	0.08	0.05
org.bouncycastle.asn1.cryptopro	7	1	0	74	1	0.14	0.14
org.bouncycastle.asn1.esf	5	2	0	47	1	0.40	0.40
org.bouncycastle.asn1.ess	5	0	0	69	1	0	0
org.bouncycastle.asn1.gnu	1	1	0	2	1	1	1
org.bouncycastle.asn1.icao	3	1	0	33	1	0.33	0.33
org.bouncycastle.asn1.misc	6	1	0	45	1	0.17	0.17
org.bouncycastle.asn1.mozilla	1	0	0	12	1	0	0
org.bouncycastle.asn1.nist	2	1	0	8	1	0.5	0.5
org.bouncycastle.asn1.ocsp	17	1	0	258	1	0.06	0.06
org.bouncycastle.asn1.oiw	2	1	2	15	0.88	0.5	0.38
org.bouncycastle.asn1.pkcs	27	1	8	441	0.98	0.04	0.02
org.bouncycastle.asn1.sec	3	1	2	27	0.93	0.33	0.26
org.bouncycastle.asn1.smime	6	1	0	52	1	0.17	0.17
org.bouncycastle.asn1.teletrust	1	1	0	2	1	1	1
org.bouncycastle.asn1.tsp	5	0	0	116	1	0	0
org.bouncycastle.asn1.util	2	0	0	53	1	0	0
org.bouncycastle.asn1.x9	11	1	8	147	0.95	0.09	0.04
org.bouncycastle.asn1.x509	62	1	141	713	0.83	0.02	0.15
org.bouncycastle.asn1.x509.qualified	8	2	0	89	1	0.25	0.25
org.bouncycastle.bcpg	49	8	23	31	0.57	0.16	0.27
org.bouncycastle.bcpg.attr	1	0	1	1	0.5	0	0.5
org.bouncycastle.bcpg.sig	11	0	15	22	0.59	0	0.41
org.bouncycastle.math.ec	10	1	73	29	0.28	0.10	0.62
org.bouncycastle.util	2	0	1	0	0	0	1
org.bouncycastle.util.encoders	11	2	3	0	0	0.18	0.82
java.io	2	0	9	0	0	0	1
java.math	1	0	247	2	0.008	0	0.99
java.security	1	0	78	3	0.04	0	0.96

Table 6-1: Metrics for the Lightweight Bouncy Castle API design

Package Name	N_C	N_A	C_A	C_E	I	A	D'
javax.crypto	5	0	2	22	0.92	0	0.08
javax.crypto.spec	2	0	3	10	0.77	0	0.23
java.security	14	2	20	27	0.57	0.14	0.29
java.security.spec	5	2	7	8	0.53	0.40	0.07
javax.microedition.pki	4	1	2	17	0.89	0.25	0.14
javax.microedition.securityservice	2	0	1	10	0.91	0	0.09
java.rmi	2	1	2	8	0.80	0.5	0.30
javacard.framework	8	0	2	3	0.60	0	0.40
javacard.framework.service	1	0	0	1	1	0	0
javacard.security	1	0	0	1	1	0	0
javax.microedition.apdu	1	1	0	3	1	1	1
javax.microedition.io	19	16	3	69	0.96	0.84	0.80
javax.microedition.jcrmi	3	2	0	10	1	0.67	0.67

Table 6-2: Metrics for the SATSA RI 1.0 API design

Package Name	N_C	N_A	C_A	C_E	I	A	D'
linca.core	6	0	0	46	1	0	0
linca.crypto.akcipher	2	1	13	11	0.46	0.5	0.04
linca.crypto.akcrypto	2	1	4	10	0.71	0.5	0.21
linca.crypto.asymmetrickey	2	2	20	4	0.17	1	0.17
linca.crypto.asymmetrickey.certificate	59	5	6	31	0.84	0.08	0.08
linca.crypto.authentication	4	2	27	0	0	0.5	0.5
linca.crypto.encoder	2	1	6	10	0.63	0.5	0.13
linca.crypto.mode	2	1	16	6	0.27	0.5	0.23
linca.crypto.signature	2	1	4	18	0.82	0.5	0.32
linca.crypto.skcipher	2	1	17	0	0	0.5	0.5
linca.crypto.symmetrickey	3	1	8	17	0.68	0.3	0.02
linca.io	4	2	7	1	0.13	0.5	0.37
linca.io.http	1	0	1	2	0.67	0	0.33
linca.math	1	0	20	0	0	0	1
linca.util	3	0	7	0	0	0	1

Table 6-3: Metrics for the Linca Mobile design

BC and SATSA also showed a good design, with most of their packages lying close to the main sequence. In order to compare the designs of Linca, BC and SATSA, we took an overall average of the *D'* columns from each table. These results are presented in Table 6-4.

LIBRARY/FRAMEWORK	AVERAGE OF D'
Lightweight Bouncy Castle API	0.34
Secure and Trust Service API	0.31
Linca	0.33

Table 6-4: Comparison of D′ averages

The results from Table 6-4 showed that average of the *D'* values for all three APIs lies close to the main sequence. However, because BC and Linca contains utility packages which are unlikely to change, we can exclude them from the calculation. Therefore, if we remove the `linca.util` package from Linca's design and `bouncycastle.util` and `bouncycastle.util.encoders` packages from BC, Linca obtained the best design overall. This result is illustrated in Table 6-5.

LIBRARY/FRAMEWORK	AVERAGE OF D'
Lightweight Bouncy Castle API	0.32
Secure and Trust Service API	0.31
Linca	0.28

Table 6-5: Comparison of D' averages without taking Utility packages into consideration

Maintainability: Application code is easier to maintain once concerns for the different cryptographic systems have been separated. Separating independent security features into different components makes it possible to distribute the development work between several developers. Also, the algorithms supported within Linca conform to the most secure standards possible, thus no immediate changes to replace these algorithms is required. Maintenance of applications should become easier because new security advantages can be added without touching the existing code. Maintenance of cryptographic algorithmic code is made simpler for the same reason.

Anticipate obsolescence. Linca is an open source[16] framework which can be reused for many types of mobile application environments and deployed on any type of mobile device. Through the open source effort, there will be a community of developers finding ways to anticipate obsolescence.

Substitutability: Linca's crypto components can be easily replaced with a securer implementation without affecting the `core` component.

[16] Linca Reference Implementation is licensed under the Lesser GNU General Public Licence (LGPL) see: www.gnu.org/licenses/lgpl.html

Imperturbability: Since all cryptographic algorithmic details are encompassed within Linca, the application will not have to make any changes in order to accommodate a change of algorithm.

Comprehensibility: The developer's comprehension of the *functionality* of any particular algorithm is probably reduced by the façade pattern in the `core` package.

Portability: Linca is a framework; therefore porting it to other programming languages should not be a problem. Currently, Linca is implemented in Java and mobile phones in the future will be Java enabled. In this way, portability is not much of a concern for Linca on any mobile devices irrespective of their make or model.

Ease of testing: Components within `crypto` package can be easily tested via their interfaces before they are plugged into Linca. For example the `skcipher`, `akcipher`, `mode` and `authentication` components have interfaces that enable easy testing of these algorithms against their respective test vectors.

6.3 Summary

In this chapter, we presented an evaluation of Linca in two categories namely: secure code comparison and qualitative analysis. In the secure code comparison, we provided security critiques against BC and SATSA and illustrated the security advantages Linca has over BC and SATSA. In the qualitative analysis, Linca was analysed against several criteria and we conducted a design comparison against BC and SATSA under the effective design criteria. The conclusion was that Linca showed a better design result in terms of the relationship between the instability and abstraction metrics.

Chapter 7 - Application

"Exercise is the beste instrument in learnyng." --Robert Recorde, The Whetstone of Witte (1557)

Linca is of no use unless it can be applied practically. In this chapter we demonstrate how Linca can be used to secure data transmitted over the network and on-device data by means of two realistic applications. Currently, Linca is deployed on Mobile Information Device Profile (MIDP) enabled mobile phones. A Java application written for the MIDP phones is called a *MIDlet* and the class files as well as any binary files associated with the application is encapsulated in a `jar` file called the *MIDlet suite*. The applications written in this chapter are emulated on Sun Microsystem's Wireless Toolkit Beta 2.5 and tested on Nokia 6600, Nokia 6680, Nokia N80 and Sony Ericsson P910 handhelds. We begin this chapter by illustrating the security relationships between Linca and the platform on which it is deployed. The security mechanism in which Linca is deployed is also analysed.

7.1 Deployment

There are two ways to deploy MIDlet suites namely: peer-to-peer or remote deployment (see Figure 7-1).

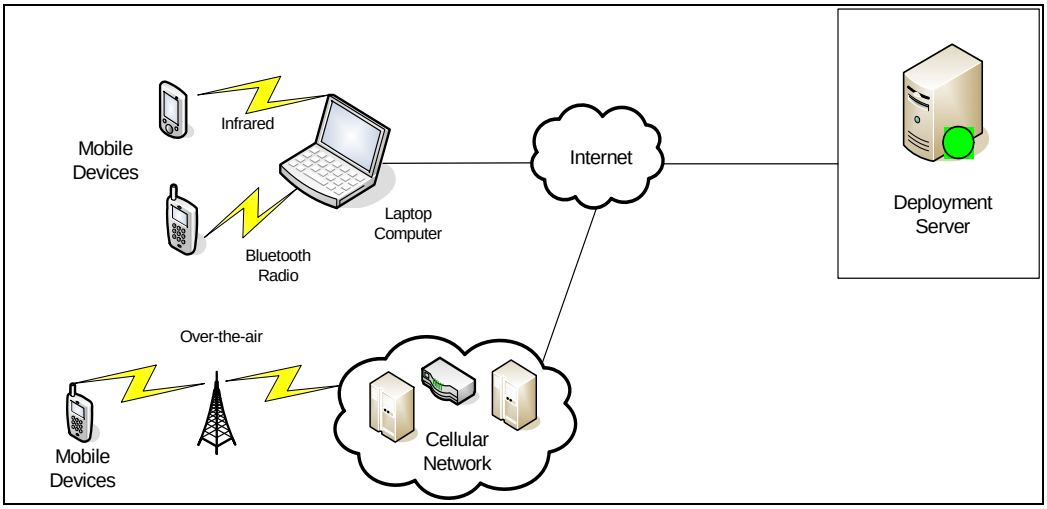


Figure 7-1: Different ways of application deployment

7.1.1 Peer-to-peer Deployment Method

A mobile application can be deployed onto a mobile device via Bluetooth, infrared, or a serial cable by connecting the mobile device with the desktop/laptop where the application was written. For example a commercial MIDlet suite can be downloaded onto a laptop first via HTTPS and then transferred onto the mobile device via Bluetooth.

7.1.2 Remote Deployment Method

Remote deployment can be made via a server hosting the MIDlet suite. This server acts as a wireless portal that allows clients to download the application, over-the-air (OTA). Sun Microsystems has drafted a documentation called *Over-the-air User Initiated Provisioning Recommended Practice for the MIDP* [78]. This document describes the best practices to deploy MIDlet suites over-the-air and the functionalities a device should provide to support this deployment.

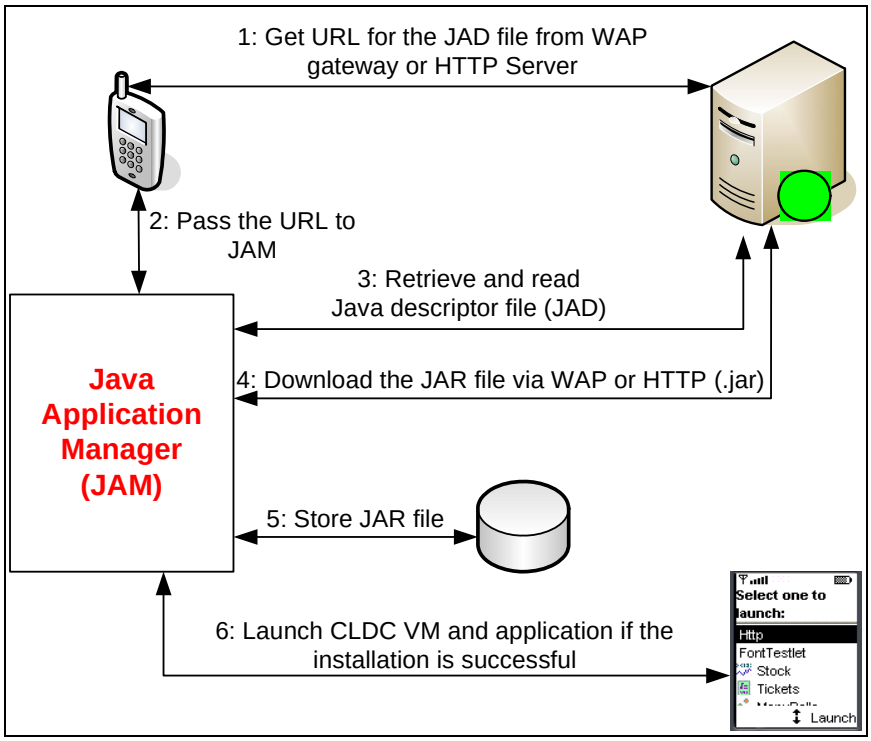


Figure 7-2: Over-the-Air provisioning of MIDlets (adopted from [39])

Devices are expected to provide mechanisms that allow users to discover MIDlet suites that can be loaded into the device. In some cases, discovery will be via the device's resident browser. In other cases, it may be a resident application written specifically to identify MIDlet suites for the user to download [39]. The OTA provisioning process is described in Figure 7-2. The Java application

descriptor (JAD) is a manifest file with attributes identifying the suite name, version, the creator of the MIDlet suite, profile description and configuration version. The JAD file is useful for OTA deployment because the application manager on the device can verify the MIDlet properties contained in JAD before it is used on the device.

Sometimes a web browser on a desktop machine can be used to download and save the MIDlet suite and then transfer it onto the target device in the same way as peer-to-peer deployment.

7.1.3 Deploying Linca

As was mentioned earlier in chapter 6, Linca is implemented separately for the mobile device and desktop server; therefore the deployment methods for each implementation will be different. The entire source code for the mobile implementation is obfuscated (see section 5.5) along with the MIDlet and is encapsulated in the MIDlet suite prior to deployment. The MIDlet suite is then distributed onto the mobile device via peer-to-peer or remote deployment methods. During remote deployment, we recommend that the MIDlet suite be downloaded onto the mobile device via secure connections such as WTLS, or HTTPS. For the deployment of the server implementation, we recommend Linca be deployed as an obfuscated `jar` file on the desktop server. The server is to be securely protected behind a firewall.

7.2 Security offered by the Platform

The security offered by the MIDP 2.0 platform supports how Linca is deployed on the mobile device.

MIDP 2.0 specification suggests a protection domain based on cryptographic signatures and certificates. The software developer, creates a signing-key pair and obtains a digital certificate from a recognized CA. The developer computes a signature of the MIDlet suite's JAR file and places that signature and the corresponding certificate in the JAD. On a MIDP 2.0 device, the application manager can verify the signature of a downloaded MIDlet suite against a root certificate. In this way the verification is done on Linca as well as the certificate and private key it uses because they are encapsulated in the MIDlet suite.

The record store system for MIDP ensures that the record generated by a Java application is only accessible by itself [9, 33]. This means that no other Java applications on the mobile device may

write or delete a record store thus providing a securer way to store the key generating parameters, mode parameter and message number during on-device data security.

7.3 Securing Network Connectivity

The most important prerequisite for securing network connectivity is to ensure the encryption keys used for the communication are not compromised. In most cases, it is the mobile device that acts as the client which requests information from a server. Linca is intended but not limited to this communication scheme. When the mobile client wants to request sensitive information from a server, therefore, either the client or the server is required to generate a symmetric key that will be used to encrypt the communication channel. The symmetric key generating mechanism in Linca was tested to be secure and fast for the mobile device implementation; therefore our recommendation is to have the mobile device to generate the symmetric key. Once the key is generated, the challenge is to let the server know of the key and most importantly, both the client and the server is required to authenticate each other.

We developed such a protocol called *SMSSec* [ref] that would enable a secure key exchange and client-server authentication. SMSSec ensures that there is:

- no transporting of secret keys and user's personal identification number (PIN) on any computing environment or intermediate station in GSM network,
- sufficient speed in encryption and decryption,
- use of existing commercially available crypto algorithms,
- availability of end-to-end encryption,
- reliability,
- no additional security protocol (for example Kerberos) or network related hardware infrastructure required for implementation.

Originally SMSSec was developed to secure SMS messaging, however the security principles applied in the protocol can be reused for cellular packet data encryption. SMSSec has a two-phase protocol with the first handshake using asymmetric cryptography which occurs only once, and a more efficient symmetric *n*th handshake which is used more dominantly (see Figure 7-3).

The first handshake is defined as follows:

M1: C $\{P_z\} \rightarrow$ S $\{P_z\}$: $E_{PKpub}[U \parallel H \parallel K \parallel Q \parallel R_c]$

M2: S $\{P_z\} \rightarrow$ C $\{P_z\}$: $E_{SK}[R_c \parallel P_j \parallel SQ]$

M3: C $\{P_z\} \rightarrow$ S $\{P_j\}$: $E_{SK}[M \parallel SQ]$

Mn: S $\{P_j\} \rightarrow$ C $\{P_z\}$: $E_{SK}[M \parallel SQ]$

The number denotations in the first handshake are defined as follows:

1a) Inputs *PIN* and *U* before *M1*

2a) Saves (*K*, *Q*) once *M1* completes

3a) Retrieves user's *PIN* to use in *H* for *M1*

4a) Saves (*K*, *Q*, *HU*) once *M1* completes

The *n*th handshake is defined as follows:

M1: C $\{P_z\} \rightarrow$ S $\{P_z\}$: $[HU \parallel E_{SK}[U \parallel H \parallel K_n \parallel Q_n \parallel R_c]]$

M2: S $\{P_z\} \rightarrow$ C $\{P_z\}$: $E_{SK\_n}[R_c \parallel P_j \parallel SQ]$

M3: C $\{P_z\} \rightarrow$ S $\{P_j\}$: $E_{SK\_n}[M \parallel SQ]$

M4: S $\{P_j\} \rightarrow$ C $\{P_z\}$: $E_{SK\_n}[M \parallel SQ]$

The number denotations in the nth Handshake are defined as follows:

1b) Inputs *PIN* and *U* before *M1*

2b) Retrieves (*K*, *Q*) to produce E_{SK} and *HU* during *M1*

3b) Saves (K_n, Q_n) once *M1* completes

4b) Retrieves *HU* and (*K*, *Q*, *PIN*) to produce E_{SK}, and *H* during *M1*

5b) Saves (K_n,Q_n, HU_n) once *M1* completes

In the first handshake, the client (*C*) forms *M1* by encrypting the following: the cellphone number (*U*), a MAC *H* consisting of HMAC_SHA-256(*U* || *PIN* || *Q*), where *Q* is given by *C* as the session identifier, secret key generating parameters (*K*), the session identifier (*Q*) and the random challenge (R_c) using the public key (E_{PKpub})of the server (*S*). The *S* validates *M1* by verifying the *H*, R_c and *Q* so that there is no replay attack on the handshake. Once *S* have validated *M1* it responds by sending *M2* by encrypting a R_c, a port number (P_j) and a sequence number (*SQ*) updated sequence to the *C* using 256-bit AES encryption in the CTR mode (E_{SK}). Recall in chapter 5 that the symmetric key can be reconstructed using the authentication data, key generating parameters and mode parameters

in Linca; therefore no key is ever exchanged over the network. The *C* receives *M2* and can authenticate the server if it can decrypt the message and verify Q and R_c. The handshake is communicated using a default port P_z. Subsequent messages after the handshake are sent using symmetric key encryption over P_j.

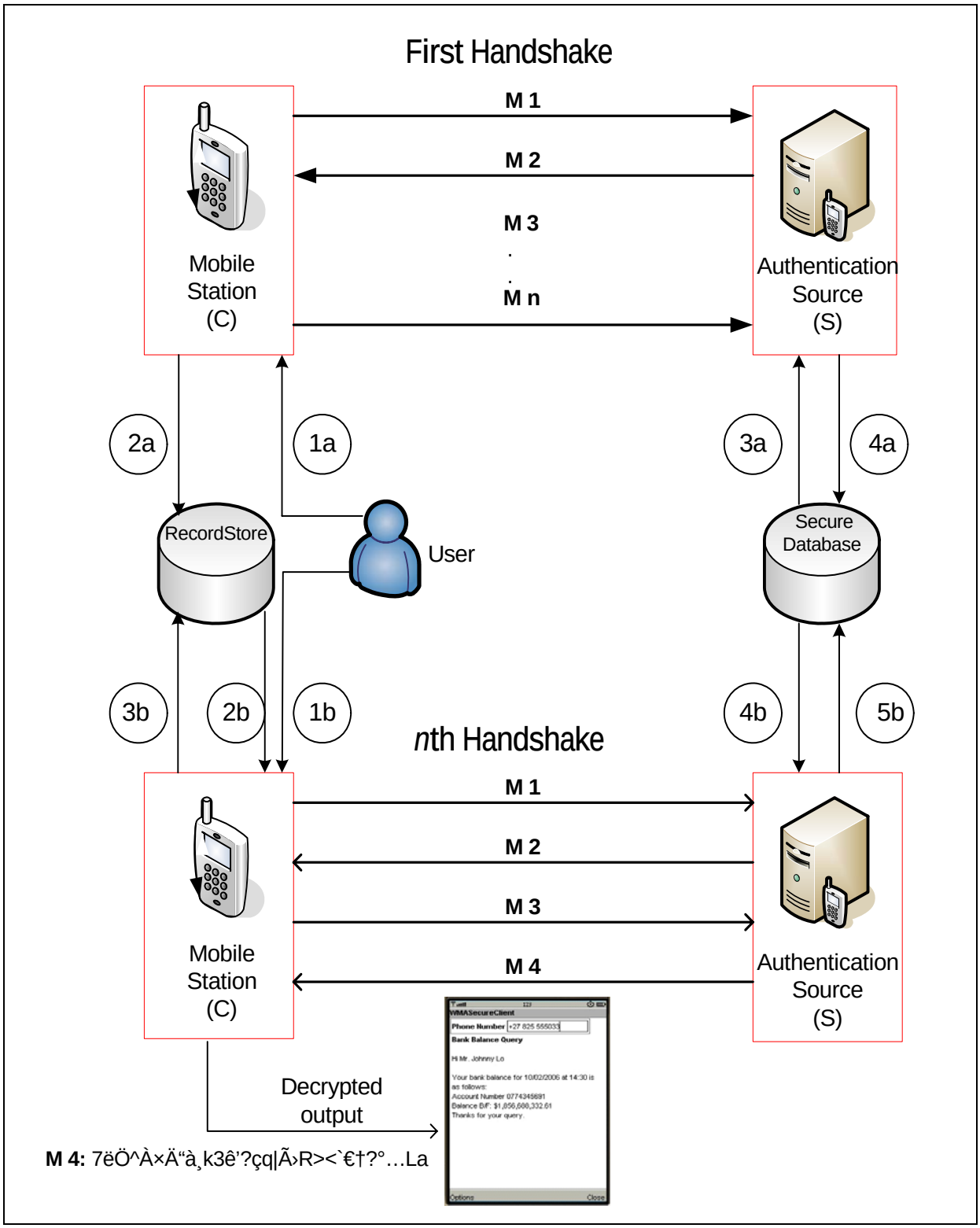


Figure 7-3: SMSSec protocol

In the *n*th handshake, the principle works in the same way as the first handshake, except the first message *M1* in the handshake is sent using the previously established symmetric key (E_{SK}) with *HU* appended in front of the encrypted elements. The *HU* is HMAC_SHA-256(*U*) with *K* as the secret input value. The *HU* value is used for establishing the origin of *M1* in the *n*th handshake. Once *M1* is validated, *M2* is sent using the freshly updated symmetric key (E_{SK_n}). The newly established key is reconstructed from the new secret key generating (K_n) parameters and is verified by the updated Q_n

The benefit of using Linca is that the developer can focus on coding the protocol without worrying about the intricacies relating to cryptography. The reader should note that we used another mobile phone as a simulated server; therefore the key loading protocol is used as a *file* instead of a *keystore*. As shown in Figures 7-4 and 7-5, the encryption and decryption of *M1* on the client and server are done in a straightforward manner without asking the developer to do any asymmetric key management.

```
1   // Initialize Linca for network encryption
2   crypto = CryptoService.getCryptoService(true);
3   ….
4   ….
5   //Formulate the message M1
6   ….
7   ….
8   //Encrypt M1
9   crypto.initAsymmetricKeyCipherComponent("file:/rsa2048cert.cer", false);
10  ciphertext = crypto.asymmetricKeyEncryption(cnt, m1);
11  ++cnt;  //update the message number
12  ….
13  ….
14  // Send the SMS
15  comm.sendSMS(ciphertext, number, port, v_listener[0]);
```

Figure 7-4: Encryption of M1 on the client during the first handshake

```
1   // Initialize Linca for network encryption
2   crypto = CryptoService.getCryptoService(true);
3   ….
4   ….
5   //Formulate the message M1
6   ….
7   cipher.initAsymmetricKeyCipherComponent("file:/rsa2048cert.cer", "file:/privatekeyenc.ser",
8                                           keypin, true);
9   byte [] M1 = crypto.asymmetricKeyDecryption(cnt_server,msg);
```

Figure 7-5: Decryption of M1 on the server during the first handshake

```
1   // Initialize symmetric key cryptosystem
2   crypto.initSymmetricKeyCipherComponents(null, null);
3   ….
4   //The symmetric key generating parameters and mode parameters are sent to the server during M1
5   ….
6   byte [] sms_enc = new byte[crypto.getSymmetricKeyCipherOutputSize(message.length, true)];
7   crypto.symmetricKeyEncryption(++cnt, message, sms_enc);
8   comm.sendSMS(sms_enc, number, Integer.toString(Pj), v_listener[0]);
9   ….
10   ….
11  //Decrypting the SMS messages
12  byte sms_dec [] = new byte[crypto.getSymmetricKeyCipherOutputSize(sms_rcv.length, false)];
13  crypto.symmetricKeyDecryption(++cnt, sms_rcv, sms_dec);
```

Figure 7-6: Subsequent SMS messages sent and received via the client

Symmetric key cryptography was used mostly throughout the protocol especially with the encryption of subsequent SMS messages (see Figure 7-6). The client computes the key generating parameters and mode parameters and both are sent in *M1*. The server reconstructs the key by retrieving the key generating and mode parameters in Line 2 of Figure 7-7.

```
1   // Initialize symmetric key cryptosystem on the server by parsing the key generating and mode parameters
2   crypto.initSymmetricKeyCipherComponents((byte []) M1DataObj[3], (byte []) M1DataObj[2]);
3   ….
4   ….
5   ….
6   byte [] sms_enc = new byte[crypto.getSymmetricKeyCipherOutputSize(message.length, true)];
7   crypto.symmetricKeyEncryption(++cnt, message, sms_enc);
8   comm.sendSMS(sms_enc, this);
9   ….
10  ….
11  //Decrypting the SMS messages
12  byte sms_dec [] = new byte[crypto.getSymmetricKeyCipherOutputSize(sms_rcv.length, false)];
13  crypto.symmetricKeyDecryption(++cnt, sms_rcv, sms_dec);
```

Figure 7-7: Subsequent SMS messages sent and received via the server

During the encryption and decryption of the SMS messages, the message number `cnt` is constantly incremented on each side so that:

- The integrity of the message sequences is kept.
- Every message sent across the network has a different ciphertext output.

7.3.1 Demonstration

In this demonstration, we emulated the server on a Nokia 6680 and the client on a Nokia N80. This is shown in Figure 7-8 where the Nokia N80 is shown on the left while the Nokia 6680 is on shown on the right. A Sony Ericsson P910 handset was also used as a server but it is not illustrated in the following illustrations.

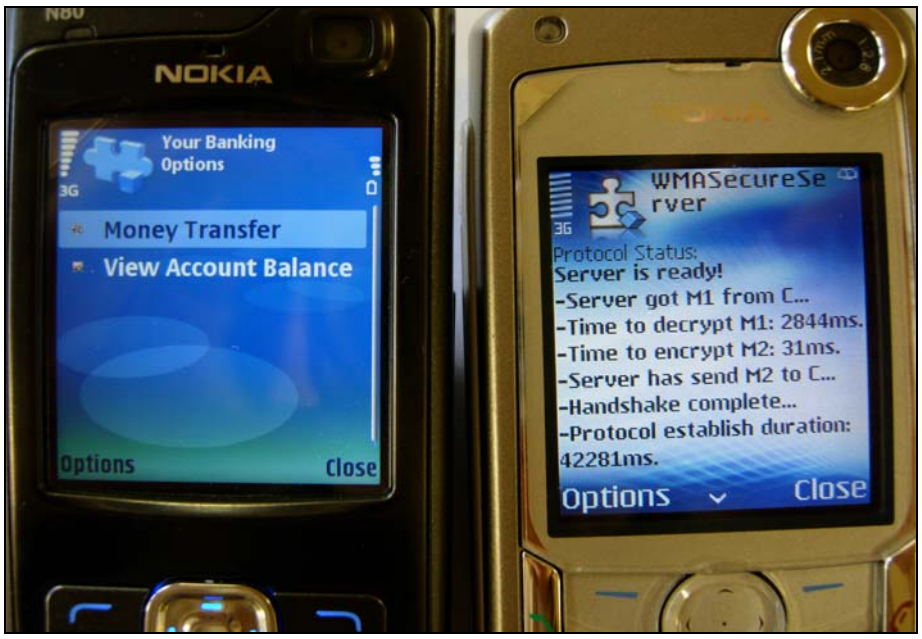
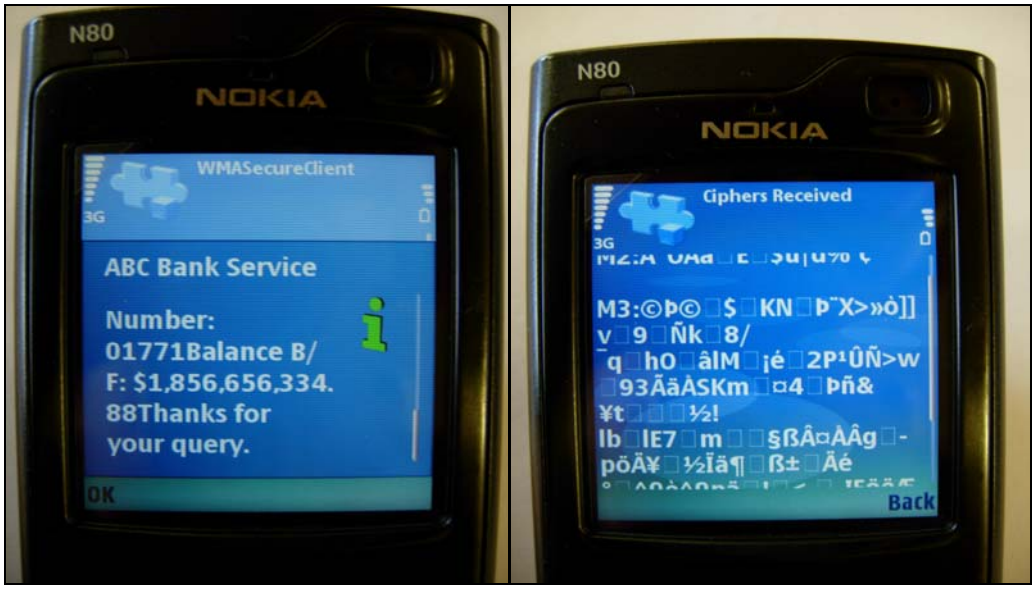


Figure 7-8: The first handshake established successfully

In Figure 7-8 the client and the server have successfully established the first handshake. The ciphertext of *M1* that the server has received from the first handshake is shown in Figure 7-9. Although the name of the application form in Figure 7-9 is called "Ciphertext Sent", it is actually displaying the ciphertext that is sent by the client that engaged the protocol.

Figure 7-9: Ciphertext of M1 received on the server

The client application is a simulated online banking application. In order to view the account balance, the client application will encrypt the content of the requested option and send it to the server. In Figure 7-10, the client queried an account balance and the decrypted bank balance is displayed in Figure 7-10 (a) while the ciphertext transmitted over the cellular network is displayed in Figure 7-10 (b).



(a) **(b)**

Figure 7-10: A successful response from the server when the account balance is queried

Upon the next handshake, the *n*th handshake will reduce the size of *M1* because *M1* will be encrypted using symmetric key encryption. The difference in the message size of *M1* cannot be shown clearly on the mobile devices due to the limitation of the screen size. Therefore the reader is encouraged to experience the *n*th handshake on an emulator.

7.4 Securing On-device Data

We demonstrate on-device data security by means of an application that stores passwords or any other secrets securely on a mobile device (see Figure 7-11). Although the record store can allow single application access, it is much safer to encrypt the on-device data.

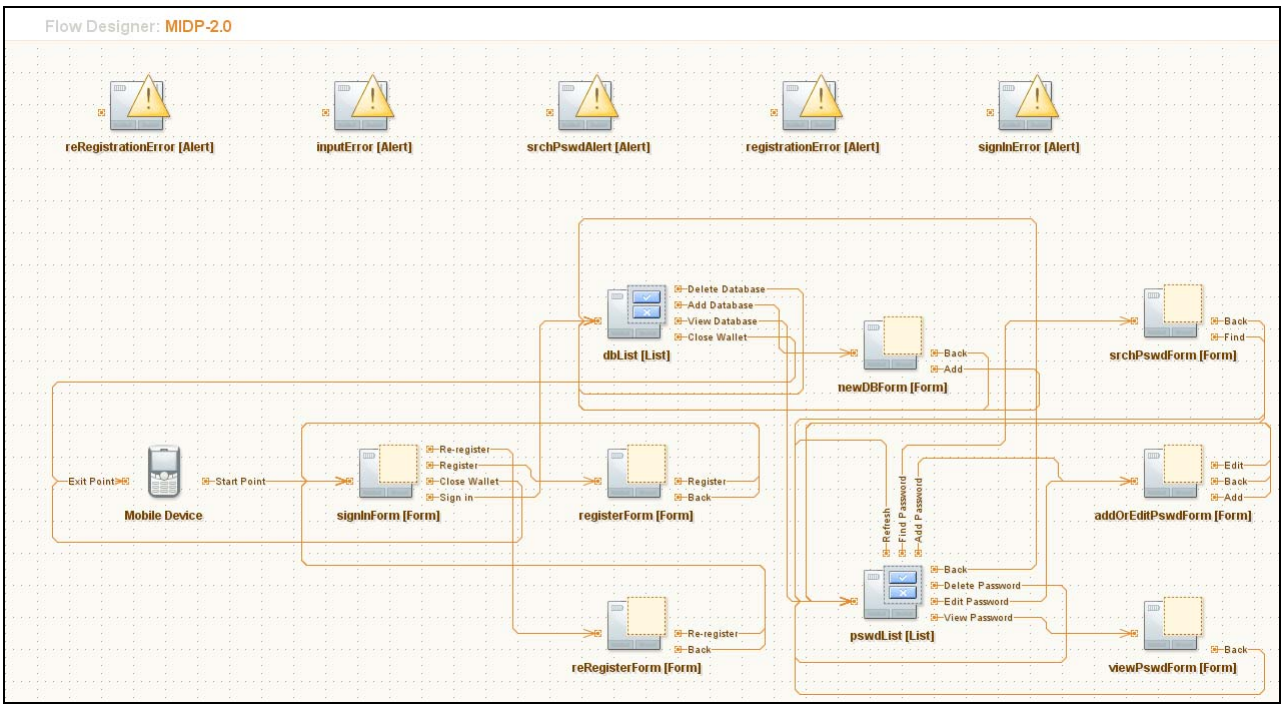


Figure 7-11: Flow design view of the PasswordWallet application

The PasswordWallet application has functionalities that allow a user to:

- Register a user name and master password (generating the symmetric key that is used for encryption).
- Re-register the user name and master password (change the current symmetric key).
- Sign-into the application using the newly created master password.
- Search, Add, Delete and Edit a password.
- Manage group-based passwords.

In order to manage the symmetric key as a PBE key in Linca, the developer has to store the key generating, mode parameters and the encrypted message number on the device. The key cannot be regenerated without the correct user authentication data that is supplied via the `usr_id` parameter in the `initSymmetricKeyCipherComponent()` method (see Figure 7-12).

When the wallet was first initialized, the registration process handles the generation of the key generating and mode parameters by calling the `initNewSymmetricKeyComponent()` method in the `SecurityManager` class which returns those parameters to be stored in the record store. The storing of the parameters is handled in the `UserManager` class which is responsible for user's sign-in and registration process.

```
1   public Object [] initNewSymmetricKeyComponent(StringBuffer credentials) {
2
3       //This method is called when it is necessary to initialize a new symmetric key component
4
5       CryptoService newService = CryptoService.getCryptoService(false);
6
7       try {
8
9           newService.initSymmetricKeyCipherComponent(credentials, null, null);
10
11           byte keyParams []= new byte[newService.getSymmetricKeyParamSize()];
12           byte modeParams[]  = new byte [newService.getModeParamSize()];
13
14           newService.getSymmetricKeyParam(keyParams);
15           newService.getModeParam(modeParams);
16
17           Object params [] = new Object[2];
18           params[0] = keyParams;
19           params[1] = modeParams;
20
21           return params;
22       }catch(CryptoException ce){}
23
24       return null;
25  }
```

Figure 7-12: A method that returns the key generating and mode parameters

```
1   public void initSecurityManager(StringBuffer credentials, byte [] keyParam, byte [] modeParam,
2                                   boolean isReRegister) {
3
4       try {
5           if(isReRegister) {
6               if(regService != null) regService = null;
7               regService = CryptoService.getCryptoService(false);
8               regService.initSymmetricKeyCipherComponent(credentials, keyParam, modeParam);
9               hasReRegistered = true;
10
11          } else {
12
13              if(hasReRegistered && mainService != null) {
14                  mainService = null;
15                  hasReRegistered = false;
16              }
17              mainService = CryptoService.getCryptoService(false);
18              mainService.initSymmetricKeyCipherComponent(credentials, keyParam, modeParam);
19              mainService.setMessageNumber(trackNum);
20
21          }
22
23      }catch(CryptoException ce) {
24
25          ce.printStackTrace();
26      }
27
28  }
```

Figure 7-13: Initializing the symmetric key cryptosystem that will be used for encryption and decryption

In Figure 7-13, the stored key generating and mode parameters are passed via `keyparam` and `modeparam` during sign-in. The reader should note that we made use of two `CryptoService` instances where one handles the sign-in, encryption and decryption (`mainService`) and the other handles the re-registration (`regService`).

The reason for this separation is to cater for the different user requests in managing the master password. During re-registration, the wallet application has to first decrypt the stored passwords using the old master password and re-encrypt them with the new master password. For an ordinary sign-in, the `mainService` instance is used and the message number is set to the last used number (Line 19 in Figure 7-13). In this way, Linca is able to continue encrypting a new password with the following message number since the last encryption. The message number is stored each time a password is saved or edited and retrieved during encryption. The message number is never reused, even when a password is deleted. The passwords are only decrypted in memory. The code for encryption and decryption can be referred in the `Container` class.

So far, we only demonstrated how encryption for on-device data is managed in Linca. The usage of the MAC function is also important. The MAC function is used for the verification during the user registration. We generated a MAC using the key and mode generating parameters (see Lines 19-30 in Figure 7-14). The password is hashed using SHA-256 within Linca to be used as the secret value for the MAC. The MAC generation method is wrapped in the `generateMAC()` method in the `SecurityManager` class.

To ensure the integrity of the MAC, the record store version is used to ensure the hash was not tampered with since the last sign-in. Therefore, the MAC value will always remain different in accordance with the last modification of the record store. During sign-in, the MAC value is verified so that the last hash was correct and that the record was not tampered with. Linca's MAC initialization function allows the developer to cater for such integrity values during the update of the MAC.

```
1   public boolean register(String userName, String password) {
2
3       if(!isRegistered()) {
4           try {
5
6               StringBuffer cr = new StringBuffer();
7               cr.append(userName);
8               cr.append(password);
9
10              //initialize a new symmetric key crypto component
11              Object params [] =  MainManager.getSecurityManager().initNewSymmetricKeyComponent(cr);
12              byte temp [] = null;
13
14              //Storing the user credentials in the following order...
15              //1: Symmetric Key Param
16              //2: Mode Param
17              //3: Password Hash Data
18
19              userRecord.addRecord(temp = (byte [])params[0], 0, temp.length);
20              userRecord.addRecord(temp = (byte [])params[1], 0, temp.length);
21
22              temp = null;
23
24              //we now know the extra value of the update...
25              Object authData [] = {Integer.toString(userRecord.getVersion() + 1)};
26
27              Object toMAC [] = ((byte [])params[0], (byte [])params[1]};
28
29              byte [] mac = MainManager.getSecurityManager().generateMAC(password.getBytes(), toMAC,
30                                                                         authData);
31              userRecord.addRecord(mac, 0, mac.length);
32
33              return true;
34             } catch(RecordStoreException rse) {}
35      }
36
37       return false;
38  }
```

Figure 7-14: Generating the MAC for user verification

7.4.1 Demonstration

We tested the Password Wallet application on a Nokia N80 (see Figure 7-15). A master password was created for signing into the application. We created three databases namely: Registered Websites, Internet Banking and Cs Department (shown in Figure 7-15 (a)) and selected to view the Cs Department's (Computer Science Department) password list. There are two fictitious passwords within the Cs Department's database (see Figure 7-15 (b)) and once the desired entry is selected, the password is decrypted in memory before it is displayed (see Figure 7-15 (c)).

(a) (b) (c)

Figure 7-15: PasswordWallet application in action

7.5 Summary

In this chapter we demonstrated how MIDP applications are deployed on mobile phones and discussed the influence of platform security on Linca. We developed two applications namely an SMSSec protocol and a Password Wallet to demonstrate how Linca can be applied to secure network communication and on-device data. The sample codes from these applications illustrate that Linca was simple to use and an effective application of cryptography was achieved. The full source code of the SMSSec protocol and PasswordWallet can be found in on the CD-ROM.

Chapter 8 - Conclusion and Future Work

"It is impossible to predict the unpredictable." – Don Cherry

In this research, we found that the current cryptographic APIs for mobile applications are quite complex to use due to a lightweight API. Due to this complexity, incorrect use of cryptography increases. To substantiate our claim, we analysed two popular APIs namely: Bouncy Castle API (BC) and Secure and Trust Service API (SATSA) for the J2ME/MIDP platform, both of which were shown to violate certain cryptographic fundamentals. We have listed these fundamentals as a part of the literature study. Aside from the complex interface methods supplied by these two APIs, we also showed that their internal implementation of cryptographic components has security vulnerabilities. These vulnerabilities include choices in the cryptographic algorithms, cryptographic algorithm initialization, key management and signature verification and generation.

In order to overcome these security limitations, we developed a framework called Linca. In this concluding chapter, we give an overview on the achievement Linca has made in applying sound cryptography practically in a mobile application.

8.1 Security Achieved

In this section we give an overview on the security improvement Linca has achieved.

8.1.1 Internal Components

We highlight some major areas within Linca where good security and efficiency are implemented:

- **Strong cryptographic algorithm support.** Due to a plethora of cryptographic algorithms available, the choice of an algorithm can be a daunting task. When implementing cryptography for mobile devices, developers sometimes sacrifice security over efficiency. Therefore poor choice of the algorithm and the key size could be made. We opted for strong encryption algorithms such as AES256, AES256_CTR and RSAES-OAEP-2048. An evaluation of the encryption and decryption speed of those algorithms was done on a

mobile device and the results were promising for encrypting/decrypting small messages. A strong MAC function is supported by HMAC_SHA-256.

- **Cryptographic algorithm initialization.** Intricate cryptographic algorithm initializations are handled within the framework so that they are hidden from the developer. These complexities include the instantiation of cryptographic algorithms, the environment (network or on-device) in which these algorithms are used, management of symmetric and asymmetric keys, counter/IV management and signature management. We utilized sound cryptographic fundamentals over these areas so that vulnerabilities during the initialization of these algorithms are minimized. Most algorithms are reused within other security mechanisms within the framework thus keeping the footprint small. In this way the developer can concentrate on implementing the logics behind the security within the application.

- **Key management.** For symmetric key management, the key generation is hidden from the developer and the key can be used to secure network communication and on-device data. For the mobile implementation the sound stream recorded from the microphone on a mobile phone was evaluated to have poor statistical random results. In order to improve the randomness on the sound stream, we evaluated several mechanisms and chose the best according to the programming effort, speed and statistical randomness. The sound stream is used as a part of the random data that is used to seed the key-generating algorithm for the mobile implementation. The output of Linca's key-generating algorithm was analysed to have a good statistical random result. For asymmetric key management on the mobile device, the key pairs are loaded from a binary file stored on the device (by default) instead of them being generated. In this way, there is less computing done on the mobile device. The private key has to be encrypted prior to the loading. For the server implementation, the keys are loaded from a reputable `keystore` by default. A proposal was made for Linca for loading asymmetric key pairs from smart cards, however this is not implemented.

- **Signature verification and generation.** An efficient, secure and non-patented signature management algorithm was implemented by reusing the same asymmetric key cipher, symmetric key cipher and hash function within Linca.

8.1.2 External API

Linca has a simple façade API which is realised by the `CryptoService` class. Through this simplicity, effective cryptography can be applied practically. We evaluated the security improvement Linca's external API has over BC and SATSA. In order to demonstrate the practicality of Linca, we implemented two realistic examples namely an end-to-end secure SMS protocol and a password storage application using Linca. We provided sample codes to illustrate that a balance between simplicity and security was achieved.

8.2 Design Objectives Achieved

We critiqued Linca against these qualitative criteria namely: dynamic extensibility, reusability, flexibility, coupling, maintainability, anticipate obsolescence, substitutability, imperturbability, comprehensibility, portability and ease of testing. In this way, the algorithms within Linca can be better managed and reused. The reusability of each cryptographic algorithm for certain operations within Linca ensures the overall *size* of the framework to be small and assist obfuscation.

Linca's design was evaluated against BC and SATSA using the relationship between abstractness and instability. The overall average result of the main sequence for Linca is slightly better compared to BC and SATSA. This is attributed by the architecture and design patterns implemented internally within Linca.

8.3 Limitations

Due to the stringent security requirement we have set for Linca, several limitations are noted. However the reader should note that these limitations are seen as the sacrifices made for a better security. These limitations are:

- **Inflexible choice in algorithm and key size.** As mentioned in chapter 5, Linca only supports a restricted set of cryptographic algorithms and key sizes. This has an impact on bending the security policies set by a company where a 128-bit Twofish is preferred and so on.

- **The API might restrict the way on security implementation.** Since Linca does not support a PRNG, it can be limiting if a protocol is required to have a random bytes to be

generated in the application. In order to retrieve random bytes, the seed value in the key generating parameter can be used.

- **Code optimization.** Linca is implemented with the best practices found in [30, 33] for optimizing control structures and array creations. However the actual implementation was not analysed using a profiling tool [53]. The profiling tool can help in monitoring: heap memory size, garbage collection statistics, thread count, thread state, creation of objects and determine the CPU time by an application. We would like to optimize Linca further using such a tool.

- **Symmetric key management for on-device data security is complicated.** In order to manage the on-device data encryption, the developer has to store the message number, key generating parameters and mode parameter. The authentication data is susceptible to being retrieved by keylogger software during the initialization of the symmetric key cryptosystem if the authentication data is entered using the keypad. However this problem is the same with entering the PIN for accessing smart cards.

8.4 Proposed Future Work

- A cryptographic library called *cryptlib* [20] focuses on the security of the internal object by assigning access control lists. We would like to investigate further the possibilities of applying the security techniques in cryptlib can be applied in Linca.

- The current specification of Linca for mobile devices has support for smart cards, however it is not implemented yet and further work is needed for testing such an implementation.

- We would like to investigate a technique for foiling attempts on recovering the authentication data via keylogger software on mobile phones.

- Elliptic curve cryptography (EEC) is a promising replacement of RSA cipher in the future. Further work needs to be done to verify the ease of integration of EEC into Linca.

- We would like to implement Linca for other platforms such as Windows CE, Symbian OS, Pocket PC, PalmOS and Embedded Linux (Qtopia).

- Future works for the SMSSec protocol is to modify the protocol for securing GPRS/3G connections. A possible improvement for the password wallet is to implement a backup facility for the passwords to be sent to a server.

- We would like to get more people involved in using and maintaining Linca. In this way, constructive feedbacks from other people can be used to improve the framework. Therefore, we are going to create a project space on SourceForge[17] for Linca.

Security is an ongoing process and it does not end by applying secure cryptography. As noted by Bruce Schneier:

"*the security of a system is as strong as its weakest link*",

and it is our hope that Linca will ensure cryptography will not be the weakest security link in a system.

[17] The URL for SourceForge is: http://sourceforge.net/index.php

References

[1] Ross Anderson, Eli Biham, and Lars Knudsen 1999. Serpent: A Candidate Block Cipher for the Advanced Encryption Standard. [Online]. Available: http://www.cl.cam.ac.uk/~rja14/serpent.html. Last accessed on: 12 July 2005.

[2] M. Bellare and P. Rogaway 1995. Optimal Asymmetric Encryption - How to Encrypt with RSA. *Advances in Cryptology - Eurocrypt '94*, vol. 950 of Lecture Notes in Computer Science, 92-111, Springer Verlag.

[3] The Legion of BouncyCastle 2005. BouncyCastle API. [Online]. Available: www.bouncycastle.org. Last accessed on: 10 November 2005.

[4] Carolynn Burwick, Don Coppersmith, Edward D'Avignon, Rosario Gennaro, Shai Halevi, Charanjit Jutla, Stephen M. Matyas Jr, Luke O'Connor, Mohammad Peyravian, David Safford, and Nevenko Zunic 1999. MARS - a candidate cipher for AES. [Online]. Available: http://www.research.ibm.com/security/mars.html. Last accessed on: 10 March 2004.

[5] CellularOnline 2006. Mobile User Statistics. [Online]. Available: http://www.cellular.co.za. Last accessed on: 10 November 2006.

[6] C.C. Center 2000. CERT Advisory CA-2001-04 Unauthentic "Microsoft Coporation Certificates". [Online]. Available: http://www.cert.org/advisories/CA-2001-04.html. Last accessed on: 10 September 2004.

[7] Nicolas T. Courtois 2005. Is AES a Secure Cipher. [Online]. Available: http://www.nicolascourtois.net/. Last accessed on: 16 August 2005.

[8] Joan Daemen and Vincent Rijmen 1999. AES Proposal: Rijndael. [Online]. Available: http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael-ammended.pdf. Last accessed on: 12 July 2004.

[9] Mourad Debbabi, Mohamed Saleh, Chamseddine Talhi, and Sami Zhioua 2005. Security Analysis of Mobile Java. In *Proceedings of the Sixteenth International Workshop on Database and Expert Systems Applications*, 231 - 235.

[10] W. Diffie and M. Hellman November 1976. New Directions in Cryptography. *IEEE Transactions on Information Theory*.

[11] T. ElGamal 1985. A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory*, vol. IT-31, n. 4, 469-472.

[12] C Ellison and B Schneier 2000. Ten Risks of PKI. *Computer Security Journal*, vol. 16, n.1, 1-7.

[13] Niels Ferguson and Bruce Schneier 2003. *Practical Cryptography.* Indianapolis, Indiana: Wiley Publishing, Inc.

[14] M. Fink 2003. *Open-source Software.* Upper-Saddle River, New Jersey: Prentice Hall.

[15] Brian Gladman 2005. Code for AES and Combined Encryption/Authentication Modes. [Online]. Available: http://fp.gladman.plus.com/cryptography_technology/index.htm. Last accessed on: 18 November 2005.

[16] GNU Crypto 2006. The GNU Crypto Project. [Online]. Available: http://www.gnu.org/software/gnu-crypto/. Last accessed on: 02 October 2006.

[17] I. Goldberg and D. Wagner January 1996. Randomness and the Netscape Browser. *Dr. Dobb's Journal*.

[18] Constatinos F. Grecas, Sotirios I. Maniatis, and Iakovos S. Venieris 2003. Introduction of the Asymmetric Cryptography in GSM, GPRS,UMTS, and Its Public Key Infrastructure Integration. *Mobile Networks and Applications*, vol. 8, 145-150.

[19] M.L. Grell 1999. Re: Encoding Methods PSS/PSS-R. Letter to IEEE P1363 working group, University of California. [Online]. Available: http://grouper.ieee.org/groups/1363/P1363/patents.html. Last accessed on: 23 May 2005.

[20] Peter Gutmann 2001. *The Design and Verification of a Cryptographic Security Architecture*, PhD Thesis, Department of Computer Science, University of Auckland, New Zealand.

[21] Peter Gutmann 2000. X.509 Style Guide. [Online]. Available: http://www.cs.auckland.ac.nz/~pgut001/pubs/x509guide.txt. Last accessed on: 2 March 2005.

[22] M. Haahr 1999. Introduction to Randomness and Random Numbers. [Online]. Available: http://www.random.org/essay.html/index2.html. Last accessed on: 15 October 2006.

[23] Hewlett-Packard Company 1997. Engineering Process Summary (Fusion 2.0).

[24] IAIK 2004. IAIK JCE-ME library. [Online]. Available: http://jce.iaik.tugraz.at/products/10_me-jce/index/php. Last accessed on: 12 June 2004.

[25] IBM 2004. Structural Analysis for Java. [Online]. Available: https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=AW-0FF&S_PKG=0FF&S_TACT=105AGX30&S_CMP=DEVX. Last accessed on: 27 November 2006.

[26] Internet Engineering Task Force 1994. "RFC 1750 - Randomness Recommendations for Security."

[27] Internet Engineering Task Force 1997. "RFC 2104 - HMAC: Keyed-Hashing for Message Authentication."

[28] Internet Engineering Task Force 2002. "RFC 3280 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile."

[29] Burt Kaliski 2003. TWIRL and RSA Key Size. [Online]. Available: http://www.rsasecurity.com/rsalabs/node.asp?id=2004. Last accessed on: 18 April 2006.

[30] Jonathan Knudsen 2003. *Wireless Java: Developing with J2ME*. New York: Apress.

[31] Donald E. Knuth 1998. *The Art of Computer Programming*. vol. 2, 3rd ed: Addison-Wesley.

[32] T. Kohno 2004. Analysis of the WinZip encryption standard. *IACR ePrint Archive, University of California at San Diego*, vol. 2004/078.

[33] Micheal Kroll and Stefan Haustein 2002. *Java 2 Micro Edition Application Development.* Indianapolis, Indiana: SAMS.

[34] Eric Lafortune 2006. Proguard Obfuscator. [Online]. Available: http://proguard.sourceforge.net. Last accessed on: 10 August 2006 2006.

[35] Timothy C. Lethbridge and Robert Laganière 2005. *Object-Oriented Software Engineering Practical Software Development using UML and Java.* 2nd ed. Glasgow: McGraw Hill.

[36] Helger Lipmaa, Phillip Rogaway, and David Wagner January 2003. Comments to NIST concerning AES Modes of Operations: CTR-Mode Encryption. [Online]. Available: http://csrc.nist.gov/CryptoToolkit/modes/workshop1/papers/lipmaa-ctr.pdf. Last accessed on: 12 September 2005.

[37] Johnny Lo and Judith Bishop 2003. Component-based Interchangable Cryptographic Architecture for Securing Wireless Connectivity in Java Applications. In *Proceedings of the 2003 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists, SAICSIT*, Johannesburg, South Africa, 301-307.

[38] Johnny Li-Chang Lo, Judith Bishop, and J.H.P. Eloff 2006. SMSSec: an end-to-end protocol for secure SMS. Technical Report.

[39] Qusay H. Mahmoud 2002. Deploying Wireless Java Applications. [Online]. Available: http://developers.sun.com/techtopics/mobility/midp/articles/deploy/. Last accessed on: 18 August 2004.

[40] G. Marsaglia 2000. The Diehard Battery of Tests of Randomness. [Online]. Available: http://www.cs.hku.hk/diehard. Last accessed on: 02 October 2006.

[41] Robert C. Martin 2002. *Agile Software Development.* 2nd ed: Addison-Wesley.

[42] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone October 1996. *Hand book of Applied Cryptography.* Boca Raton, Florida: CRC Press.

[43] Microsoft Corporation 2003. Windows CE and Pocket PC. [Online]. Available: http://www.microsoft.com/windowsmobile/pocketpc/ppc/default.mspx. Last accessed on: 10 August 2004.

[44] Microsoft Corporation 2005. Whats New in the .NET Compact Framework 2.0. [Online]. Available: http://msdn2.microsoft.com/en-us/library/aa446574.aspx. Last accessed on: 23 May 2006.

[45] V.S. Miller 1986. Use of Elliptic Curves in Cryptography. *Advances in Cryptology - Crypto 85 Proceedings, Springer Verlag*, 417-426.

[46] National Communications System 2003. SMS over SS7. *Technical Information Bulletin 03-2*, 39,41.

[47] National Security Agency Cross Organization CAPI Team 1997. "Security Service API: Cryptographic API Recommendation, Updated and Abridged Edition."

[48] National Institute of Standards and Technology 2002. Secure Hash Standard. *Federal Information Processing Standards Publication 180-2*.

[49] National Institute of Standards and Technology 1999. Data Encryption Standard. *Federal Information Processing Standards Publication*.

[50] National Institute of Standards and Technology 1991. "Digital Signature Standard - FIPS PUB 186."

[51] National Institute of Standards and Technology 2005. Cryptographic Toolkit - Encryption. [Online]. Available: http://csrc.nist.gov/CryptoToolkit/tkencryption.html. Last accessed on: 14 October 2005.

[52] National Institute of Standards and Technology 2005. "FIPS PUB 140-2 Security Requirements for Cryptographic Modules."

[53] Netbeans.org 2006. Netbeans Profiler 5.0. [Online]. Available: http://www.netbeans.org/products/profiler/. Last accessed on: 18 August 2006.

[54] Nokia Corporation 2005. Developer Platform 2.0: Known Issues. [Online]. Available: http://www.forum.nokia.com/. Last accessed on: 12 December 2005.

[55] NTRU 2004. NTRU Toolkits. [Online]. Available: http://www.ntru.com/cryptolab/algorithms.htm. Last accessed on: 10 March 2004.

[56] C. Enrique Ortiz 2005. The Security and Trust API (SATSA) for J2ME: The Security APIs. [Online]. Available: http://developers.sun.com/techtopics/mobility/apis/articles/satsa2/. Last accessed on: 05 May 2006.

[57] Palm OS 2004. PalmOne and PalmSource. [Online]. Available: http://www.palm.com/us/. Last accessed on: 18 August 2004.

[58] Phaos Technology 2005. Phaos Technology Micro Security Products. [Online]. Available: http://www.phaos.com/products/category/micro.html. Last accessed on: 28 March 2004.

[59] David Pointcheval 2002. How to Encrypt Properly with RSA. *RSA Laboratories Cryptobytes*, vol. 5, 10-19.

[60] Karen Renaud, Judith Bishop, Johnny Lo, and Basil Worrall 2005. Algon: from interchangeable distributed algorithms to interchangeable middleware. *Software Composition 2004, Electronic. Notes Theory. Computer Science*, 114:65-85.

[61] R. Rivest, A. Shamir, and L. Adleman February 1978. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *Communications of the ACM*.

[62] Ronald L. Rivest, M.J.B. Robshaw, R. Sidney, and Y.L. Yin 1999. The RC6 Block Cipher. [Online]. Available: http://www.rsasecurity.com/rsalabs/rc6. Last accessed on: 17 July 2005.

[63] RSA Laboratories 2000. Why is cryptography export-controlled? Available online in Crypto FAQ. [Online]. Available: http://www.rsasecurity.com/rsalabs/node.asp?id=2330. Last accessed on: 11 September 2005.

[64] RSA Laboratories 1999. PKCS#5 v2.0: Password-Based Cryptography Standards, 25 March 1999. [Online]. Available: ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf. Last accessed on: 22 August 2005.

[65] RSA Laboratories 1999. PKCS#5 v2.0: Password-Based Cryptography Standards. *PCKS Standards*.

[66] RSA Laboratories 2002. PKCS#1 v2.1: RSA Cryptography Standard. *PCKS Standards*.

[67] RSA Laboratories 2005. The New RSA Factoring Challenge. [Online]. Available: http://www.rsasecurity.com/rsalabs/node.asp?id=2092. Last accessed on: 14 October 2005.

[68] RSA Laboratories 2006. What is PKI? [Online]. Available: http://www.rsasecurity.com/rsalabs/node.asp?id=2268. Last accessed on: 9 August 2006.

[69] James Rumbaugh, Ivar Jacobson, and Grady Booch 1999. *The Unified Modeling Language Reference Manual.* Addison Wesley Longman, Inc.

[70] Bruce Schneier 1996. *Applied Cryptography.* 2nd ed. New York: John Wiley & Sons, Inc.

[71] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson 1998. Twofish: A 128-bit Block Cipher. [Online]. Available: http://www.schneier.com/paper-twofish-paper.pdf. Last accessed on: 20 July 2005.

[72] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson 1999. Performance Comparison of the AES Submissions. In *Proceedings of Second AES Candidate Conference*, NIST, 15-34.

[73] V. Shoup 2001. OAEP Reconsidered. *Advances in Cryptology - Crypto 2001*, vol. 2139 of Lecture Notes in Computer Science, 239-259, Springer Verlag.

[74] Kent Inge Fagerland Simonsen 2005. *J2ME Security*, Master of Science Thesis, Department of Informatics, University of Bergen, Norway.

[75] William Stallings 2000. *Network Security Essentials, application and standards.* New Jersey: Prentice Hall.

[76] Jari Sukanen 2004. *Extension framework for Symbian OS application*, Master of Science Thesis, Department of Computer Science, University of Helsinki, Finland.

[77] Sun Microsystems 2000. Security Code Guidelines. [Online]. Available: http://java.sun.com/j2se/sdk/1.2/docs/guide/security/seccodeguide.html. Last accessed on: 12 April 2005.

[78] Sun Microsystems 2001. Over The Air User Initiated Provisioning Recommended Practice for the Mobile Information Device Profile. [Online]. Available: http://java.sun.com/products/midp/OTAProvisioning-1.0.pdf. Last accessed on: 23 Apirl 2004.

[79] Sun Microsystems 2002. Java Cryptography Architecture Reference. [Online]. Available: http://java.sun.com/products/jdk1.2/docs/guide/security/cryptospec.html. Last accessed on.

[80] Sun Microsystems 2002. JSR 118: MIDP 2.0 Specification.

[81] Sun Microsystems 2004. Security and Trust Service API. [Online]. Available: http://java.sun.com/products/satsa/. Last accessed on: 12 September 2005.

[82] Sun Microsystems 2006. Java Card Development Kit version 2.2.2. [Online]. Available: http://java.sun.com/products/javacard/. Last accessed on: 10 April 2006.

[83] Symbian 2004. Symbian OS v8.1a documentation and example source code. [Online]. Available: http://www.symbian.com/developer/techlib/v8.1adocs/index.asp. Last accessed on: 21 November 2004.

[84] Trolltech Inc. 2004. Qtopia overview. [Online]. Available: http://www.trolltech.com/products/qtopia/. Last accessed on: 18 August 2004.

[85] United States Bureau of Industry and Security 2005. Export Control Basics. [Online]. Available: http://www.bis.doc.gov/licensing/index.htm. Last accessed on: 22 June 2005.

[86] S.A. Vanstone 2003. Next generation security for wireless: elliptic curve cryptography. *Computers and Security*, vol. 22, 12-44.

[87] U. Varshney, R.J. Vetter, and R. Kalakota October 2000. Mobile Commerce: A new Frontier. *IEEE Computer*, vol. 10, 32-38.

[88] Russel Dean Vines 2002. *Wireless Security Essentials.* New York: Wiley Publishing.

[89] John Walker 1998. ENT Tool. [Online]. Available: http://www.fourmilab.ch. Last accessed on: 20 September 2006.

[90] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu 2005. Finding Collisions in the Full SHA-1. In *Crypto 2005*, Santa Barbara, California, USA, to appear.

[91] Michael J. Wiener 1990. Cryptanalysis of short RSA secret exponents. In *IEEE Transactions of Information Theory*, vol. 36, 553-558.

[92] Wikipedia 2005. History of Cryptography. [Online]. Available: http://en.wikipedia.org/wiki/History_of_cryptography. Last accessed on: 07 July 2005.

[93] Micheal Juntao Yuan 2004. *Enterprise J2ME - Developing Mobile Java Applications.* Upper Saddle River, New Jersey: Prentice Hall.

Appendix A - Acronyms

3G	Third Generation Network
AES	Advanced Encryption Standard (Rijndael)
AES<size>	Advanced Encryption Standard with the key size <size>
AES<size>_CTR	AES with key size <size> combined with the counter mode
API	Application programming interface
BC	Bouncy Castle API
CLDC	Connected Limited Device Configuration
CMS	Cryptographic Message Syntax
CTR	Counter mode
DoS	Denial of Service
ECC	Elliptic Curve Cryptography
FIPS	Federal Information Processing Standards
GPRS	General Packet Radio Service
GSM	Global System for Mobile communications
HMAC_SHA-256	HMAC with SHA256 as the underlying hash function
IETF	Internet Engineering Task Force
IV	Initialization Vector
JAD	Java Application Descriptor
JAM	Java Application Manager
J2ME	Java 2 Micro Edition
J2SE	Java 2 Standard Edition
Linca Mobile	Implementation of Linca for the mobile device
Linca Server	Implementation of Linca for the server
LCC	Linca Connectivity Component
MAC	Message Authentication Code
MIDP	Mobile Information Device Profile
NIST	National Inistitute of Standards and Technology
NSA	National Security Agency
OAEP	Optimal Asymmetric Encryption Padding
OTA	Over The Air
PBE	Password-based Encryption
PKI	Public Key Infrastructure
RI	Reference Implementation
RSAES	Encryption scheme using the RSA cryptosystem
RSAES-OAEP	RSA encryption/decryption using OAEP encoding scheme
SATSA	Secure and Trust Service API
SHA-<size>	Secure Hash Algorithm – with the output size as <size>
SIM	Subscriber Information Module
SMS	Small Messaging Service
WMA	Wireless Messaging API
WTLS	Wireless Transport Layer Security