University of New Orleans
ScholarWorks@UNO

University of New Orleans Theses and Dissertations | Dissertations and Theses

12-17-2004

Three Factor Authentication Using Java Ring and Biometrics

Jyothi Chitiprolu
University of New Orleans

Follow this and additional works at: https://scholarworks.uno.edu/td

Recommended Citation

Chitiprolu, Jyothi, "Three Factor Authentication Using Java Ring and Biometrics" (2004). *University of New Orleans Theses and Dissertations*. 187.
https://scholarworks.uno.edu/td/187

This Thesis is protected by copyright and/or related rights. It has been brought to you by ScholarWorks@UNO with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis has been accepted for inclusion in University of New Orleans Theses and Dissertations by an authorized administrator of ScholarWorks@UNO. For more information, please contact scholarworks@uno.edu.

THREE FACTOR AUTHENTICATION USING JAVA RING AND BIOMETRICS

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
The Department of Computer Science

by

Jyothi Chitiprolu

B.E. Madras University, 2001

December, 2004

ACKNOWLEDGMENTS

I would like to thank my husband Sravan, for his love, support and encouragement. He is truly my best friend and partner. This thesis would not have been possible with out his help and patience. I would like to thank my mother and father for their constant love and support and for being the best parents any one could wish for. I would like to thank my sister and brother for their abundant love, affection and humor.

Very special thanks to Dr. Golden Richard III, for supervising my thesis, for all his help, support and patience and for being an excellent teacher and advisor.

I would like to thank Dr. Shengru Tu for his help through my years at UNO.

I would like to thank all my friends for being there for me, for their help and for making this a great experience.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Computer security is a growing field in the IT industry. One of the important aspects of the computer security is authentication. Using passwords (something you know) is one of the most common ways of authentications. But passwords have proven to provide weak level of security as they can be easily compromised. Some other ways of authenticating a user are using physical tokens, (something you possess) and biometrics, (something you are). Using any one of these techniques to secure a system always has its own set of threats. One way to make sure a system is secure is to use multiple factors to authenticate. One of the ways to use multiple factors is to use all the three factors of authentication, something you possess, something you are and something you know. This thesis discusses about different ways of authentication and implements a system using three factor authentication. It takes many security aspects of the system into consideration while implementing it, to make it secure.

Chapter 1

INTRODUCTION

As the computer technology is prevailing, the security related problems with computers are also increasing. The purpose of computer security is to protect an organization's valuable resources, such as information, hardware, and software. To know if a person is right user, we need to authenticate the person. The goal of authentication is to identify a person directly or indirectly.

This thesis implements one of the best ways of authenticating a person taking many issues into consideration. The structure of this thesis is organized as follows.

Chapter 2 provides security background and different kinds of authentications. It introduces different ways of authentications and also discusses how multiple factors can be used for authentication.

Chapter 3 introduces Java Ring, and iButton. It discusses different functions the iButton provides and also focuses on how secure an iButton is. It details the Java Card technology and its functionality in an iButton. It discusses about smart card technology and open card framework. It also explains as to how the communication takes place between a Java Card and an Open Card.

Chapter 4 describes the different biometrics and how fingerprints are better than other biometrics. It discusses the fingerprint API used in the implementation.

Chapter 5 discusses the implementation of the system using three-factor authentication.

Chapter 6 concludes by summarizing the security aspects of the system implementation.

Chapter 7 discusses the future work that can be done with the system.

Chapter 2

COMPUTER SECURITY

Computer security has become a prevalent concern from the dawn of the Internet and before. Network breaches and e-commerce fraud are increasing rapidly, as reported in the Computer Security Institute -2001 Computer Crime and Security Survey [7]. Today 85**%** of large corporations and government agencies acknowledge network security breaches. Of these organizations, 64**%** acknowledge financial losses that run into millions of dollars due to network breaches and e-commerce fraud [7].

Security is about well-being (integrity) and about protecting property or interests from intrusion, stealing or wire-tapping (privacy) [1]. The purpose of computer security is to protect an organization's valuable resources, such as information, hardware, and software. Examples include remote access to computer accounts, access to web sites, and bank account access at automated teller machines. Through the selection and application of appropriate safeguards, security helps the organization's mission by protecting its physical and financial resources, reputation, legal position, employees, and other tangible and intangible assets. To grant access to a few, we need to know whom we can trust and we need to verify the credentials (authenticate) of those who come near us.

Security is thus based on the following independent issues [1]:

- **Privacy** - the ability to keep things private/confidential

- **Trust** - do we trust data from an individual or a host? Could they be used against us?

- **Authenticity** - are security credentials in order? Are we talking to whom we think we are talking to, privately or not?

- **Integrity** - has the system been compromised/altered already?

Information security is a necessary underpinning for further advances in electronic business. Technologies such as session encryption, firewalls, virtual private networks, wireless LANs, and digital certificates have all emerged as pieces of the solution. While each is designed to enhance some aspect of information security — whether by restricting access to or preventing the interception of private data — none of them alone is designed to address the fundamental security issue that underlies the most damaging information crimes such as "is the person who is attempting to access protected files or resources an authentic user or an impostor?" To know this we need to authenticate the user.

2.1. Authentication:

The goal of authentication is to identify a user either directly or indirectly. There are many possible ways of authentication available. Some of them are passwords, host-based authentication, physical tokens and biometrics.

2.1.1. Passwords:

Using password is one of the ways of restricting access to documents where the server administrator needs to be able to control access on an individual basis. This is called user authentication and requires a user name and password before being allowed to access a document. Setting up a User authentication takes two steps, first creating a file name containing the usernames and passwords. Secondly, telling the server what resources are to be protected and which users are allowed (after entering a valid password) to access them.

2.1.2. Host-Based Authentication:

Host based authentication is situation where authentication takes place based on the host information, rather than the more usual method of prompting for a password. This is very convenient if a non-interactive process is trying to authenticate with a remote machine. SSH relies on such authentications and uses public and private key pairs to establish a secure connection.

2.1.3. Physical Tokens:

Physical Token includes physical devices that are used to compute the credentials presented to the verifier as well as software files that must be possessed by the claimant in order to compute the credentials. Examples of physical devices include smart cards, magnetic stripe cards and one-time password generators. Software files will typically contain secret or private keys that are used to compute credentials; however physical devices may contain these keys as well. These physical devices are sometimes called authentication tokens or dongles.

2.1.4. Biometrics:

Biometrics includes something inherent to a person. In this category, human physical characteristics such as fingerprints, facial features, heat signature of head, retinal scans, handwriting or voice are used to produce the credentials. There are many biometric devices available. An example of biometric is the car Lexus 430s model 2004 which uses a fingerprint recognition system to identify its owner. The owner's fingerprint is scanned when he opens the door handle which uses fingerprint recognition technology to authenticate [3].

No matter what kind of authentication we use, it is impossible to have a completely secure system. But we can make it as hard as possible for the person trying to breach.

Today, the most common form of authentication is password control. In general, technologies for authenticating a potential user of an information system are organized into three identification factors: something you know, something you have, and something you are. An example of something you know is a password or a personal identification number (PIN), something you have is a physical token such as a smart card and something you are is a biometric such as a fingerprint. An application can use either one of these authentication techniques or a combination of two or more of these techniques. However, each of these factors is vulnerable to attack if used alone or in pairs. Highly secure systems can use multiple factors to secure their systems. We will next discuss about one factor, two factor and three factor authentications.

2.2. One Factor Authentication:

One factor authentication relies on one of the above three factors for authentication. Using a password is one of the most common one factor authentication used. Unfortunately, passwords can be easily misapplied and provide a weak level of security. One reason is that users tend to pick simple passwords that are easy to remember. Some of the ways passwords can be compromised are as follows.

- If a **dictionary word** is used as a password, it is a fairly quick and easy task for a computer program to try every English word and guess the password. Policies for ensuring secure passwords result in greater inconvenience for users, in turn causing users to write down the passwords or use passwords that can be easily memorized. In addition, typical users use the same password for multiple accounts, further degrading security.

- **Keystroke Monitoring** is monitoring and storing every keystroke a person makes on keyboard. Using special software, passwords are easily lifted, leading to a potential security compromise. In more extreme situations, a monitor's emissions can be read and deciphered, revealing everything displayed on the screen [8].

- **Password cracking tools** such as brute force that allow an attacker to automate the process of guessing user passwords are readily available for download from the Internet, making it relatively easy to crack the average password. This type of security breach is a result of repeated login attempts with different key combinations or words.

- Furthermore, many successful attacks are accomplished using passwords obtained from **social engineering** (an attacker's use of clever manipulation to trick trusting users into divulging password information), a problem that even the best of corporate password policies find difficult to address[6]**.**

- Using **Man-in-the-Middle Attack**, a computer is set up as an interface between a client computer and the server that handles authentication. The computer in the middle accepts the client's password as if it were the server and logs in to the server using the client's identity.

- **Network Monitoring,** also known as sniffing, occurs when a computer on a network looks for message streams that contain words such as "password" or "login." This is especially common in Ethernet networks where every computer on the network can easily read any network traffic. Streams containing passwords can be stored and used to gain unauthorized access [8].

Password security mechanisms can be strengthened further through the use of "one-time passwords." One-time passwords can be implemented through either software or hardware. But one One-time passwords have their own set of complications. User error is a common problem with password generators because users must manually enter each password during the authentication process. This can be cumbersome when repeated many times and can increase the likelihood of repeated errors. Session-based authentication is vulnerable to 'session hijacking' because the end-user is able to leave the computer unattended while the authenticated session is still active. In addition back-end management of password generation environments is time consuming and costly as databases and servers must be retooled to accommodate the changing password requirements [8].

2.3. Two factor Authentication:

When two factors are used to authenticate, it is called two factor authentication. The two factors involved may be any two of three methods: passwords, physical tokens or biometrics. Two factor authentications are better than one factor authentication because, if passwords are the only ones used, there is a probability for the password to be compromised. Passwords with physical tokens such as a smart card can be used for two factor authentication. This type of authentication is resistant to single-factor attacks including keystroke monitoring, social engineering, man-in-the-middle attacks, network monitoring, and password cracking. One example in daily use is the Bank application where the user is prompted to enter his card after which he is prompted for a pin.

Many applications use a smart card with a password. To authenticate to an information system or network, the user will insert his/her smart card into a hardware reader connected to a workstation or laptop computer. The processor on the smart card will encrypt a text string with the user's private key and the

authentication server decrypts using the public key. In this approach, the user's private key never has to be communicated outside of the smart card and never leaves the smart card's circuitry. This helps preserve the integrity of the private key.

Systems using this form of two-factor authentication are vulnerable to attacks through theft of the hardware token coupled with the use of password compromise techniques mentioned above.

An example of a product which uses two factor authentication and goes beyond a simple static password is DataCard. Datacard's Two-Factor Authentication is designed to curb network and e-commerce abuse [7].

2.4. Three factor Authentication:

Three factor authentication systems relies on three factors for authentication: something you know such as password, something you possesses such as a physical token, something you are, a biometric characteristics. Using a combination of all three makes the security system stronger. Biometrics is one of the most reliable and most widely used forms of authentication these days.

Having three factors makes it difficult to cheat a system, thus strengthening the security. There are many companies and software products that use three factor authentication. Some companies and software products in today's market which use three factor authentication are given below along with the factors each use for three factor authentication.

Fortress Technologies, the market leader in securing wireless LANs, uses three-factor authentication, a pioneering approach to multi-tiered network authentication for wireless enterprises [4]. The three-factor approach, unique to

Fortress AirFortress wireless security gateway, has three layer of authentication, Network Authentication, Device Authentication and User Authentication.

Dekart Logon, a software program designed to add the strong authentication and convenience to the standard Windows logon procedure allows to access Windows driven computers and domains in an easy, fast and secure way by using different types of hardware keys. The login and password of the user are entered automatically once the hardware key is connected to the computer. Users gain the flexibility to select from different smart cards, hardware tokens, as well as USB flash drives and other types of removable media to provide fast and convenient two- or three-factor authentication within their Windows environment [5].

Lock-Out 2000 Biometric Authentication Edition is a combination of Biometric Middleware and Authentication Middleware. The core module of Lock-Out 2000 Biometric Authentication Middleware is designed to engage three-factor's of authentication access. The security administrator of the workstation assigns a unique wearable Java-powered iButton to the user and then assigns a PIN number for (two-factor) authentication, similar to any bank ATM machine. The administration utility tool then acts as the Biometrics enrollment station, whereby the users look straight into a camera and is enrolled in less than 20 seconds with 11 default pictures. This authentication assures a 90-100% verification results.

TRIO VAULT™ combines 3-factor user authentication, a single-sign-on solution, and access management into a single, integrated Palm OS® application that interfaces seamlessly with the existing network security infrastructure and eliminates the need for authentication and single-sign-on servers. TRIO VAULT™ requires users to authenticate themselves to a PDA using all three factors of authentication. The PDA then authenticates the user to the service user

wants. Because the PDA manages the creation and use of passwords for each account, users no longer have a need to remember passwords for individual accounts [6].

This Thesis implements an application with Three factor authentication using passwords, a Java Ring (a physical token) and fingerprint recognition (biometric). The next chapter gives an introduction to the Java Ring and discusses the security issues of a Java Ring and the communication in a Java Ring.

Chapter 3

JAVA RING

3.1. Introduction to JavaRing:

A Java Ring is a wearable finger ring that contains a small microprocessor. The Java Ring is an extremely secure Java-powered electronic token with a continuously running, unalterable real time clock. The rugged packaging of the Java Ring makes it suitable for many applications. The jewel of the Java Ring is the Java iButton.

Figure 3.1 Java Ring

The Java Rings made their appearance at 1998 Java One Conference.

The iButton is a one-million transistor; single-chip trusted microcomputer with a powerful Java virtual machine (JVM) housed in a rugged and secure stainless-steel case [9].

Figure 3.2: iButton

When coming to secure internet transactions, the two most fundamental problems with internet transactions involving sensitive information are authentication and secure transmission of the data. By eavesdropping, someone can gain information about a person and steal his identity. The iButton provides for secure end-to-end internet transactions–including granting conditional access to Web pages, signing documents, encrypting sensitive files, securing email and conducting financial transactions safely - even if the client computer, software and communication links are not trustworthy. When PC software and hardware are hacked, information remains safe in the physically secure iButton chip. The ibutton connects to computers with a Blue Dot receptor [10].

Figure 3.3: Blue Dot Receptor

These are some of the functions that can be done by simply pressing the Blue Dot with the iButton:

- Granting access privileges to sensitive information on a conditionally accessed Web page using PKI challenges/response authentication.

- Signing documents so the recipient can be certain of their origin. For example, you can write and sign an expense report. Or you can author a

newspaper story, sign it at your vacation home and email it to the publisher.

- Encrypt and decrypt messages, securing email for the intended eyes only.

- Conduct hassle-free monetary transactions–print your own electronic postage stamps or prints, write, and sign your own electronic checks [10].

3.2. iButton:

The iButton is a self-powered computer chip with networking serial number housed in a 16mm stainless steel can.

Figure 3.4: Self-powered computer chip in an iButton

The iButton form factor has a computer chip with a unique way of communication by touch contact of the button to a variety of read/write devices. iButton makes many capabilities which are limited to a stationary or hard-wired computer, portable and universally available. Among these capabilities are user-accessible memory, timekeeping, temperature measurement or logging, and encryption computation.

It has the ability to perform large integer modular exponentiations at high speed which is central to RSA encryption, Diffie-Hellman key exchange, Digital

Signature Standard (FIPS 186), and many other modern cryptographic operations.

Each iButton has a unique 8-byte serial number and guarantees that no two serial numbers are the same. Each iButton can be easily used in a network with its serial number as an address for Internet connection.

As mentioned earlier, an iButton communicates with a processor by a simple touch to a 1-Wire interface called a Blue Dot Receptor. The iButton is ideal for applications where information needs to travel with a person or object. Some iButtons are memory devices that can hold files and subdirectories and can be read and written like small floppy disks. There are iButtons with password-protected capabilities for a file for security applications where the iButtons counts the number of times the files have been rewritten for securing financial transactions, point-of-sale transactions, remote access authorization, data logging (including time and temperature), maintenance and quality control.

Java Ring with the iButton can be carried around as an accessory for many reasons. Passwords are difficult to keep as a secret. They can be stolen. Short passwords are easy to guess where as long passwords are difficult to remember and tend to be written down. There are many password breaking tools available such as brute force, scanning word lists and using patterns. Many network applications transmit passwords clear over the network.

Java Ring with the iButton can overcome the deficiencies of the secret passwords. It can be used to store the secret passwords and private keys needed to conduct a transaction. Using the iButton, the keystrokes can be eliminated with a quick, intentional press of the Blue Dot.

The receptor has an adapter that connects to the computer's serial, parallel, or USB port. Communication is established when an iButton is touched to the Blue Dot receptor. The iButton draws the power it requires to operate from the connection. When not in contact with a receptor, the state of the Java virtual machine and memory is maintained with lithium backup power.

Java iButton is Java Card 2.0 compliant. Java iButton, can be used to write applets that can be compiled with the standard tools available from Sun Microsystems. These applets can be loaded into the Java iButton, and run on demand to support a wide variety of financial applications. The Java Card 2.0 specification provides the opportunity to implement a useful version of the JVM and runtime environment with the limited resources available to a small processor**.** Java Card and its related topics are discussed in chapter 3.

3.2.1. How secure is iButton:

The National Institute of Standards (NIST) and the Communications security Establishment (CSE) have validated a version of the crypto iButton for protection of sensitive, unclassified information. FIPS 140-2 validation assures government agencies that the products provide a trusted, physically secure module to properly protect secure information [10].

The stainless steel case of the device provides clear visual evidence of tampering thus providing extraordinary security. The monolithic chip includes up to 134K of SRAM that is specially designed so that it will rapidly erase its contents as a tamper response to an intrusion [10]. Rapid erasing of the SRAM memory is known as zeroization. When an iButton detects any intrusion, it erases its private keys leading to zeroization. Any attempt to uncover the private keys within the SRAM made by an attacker are thwarted because the attacker has to both

penetrate the iButton's barriers and read its contents in less than the time it takes to erase its private keys.

There are specific intrusions that result in zeroization. Opening the case of the iButton, removing the chips metallurgically bonded substrate barricade, micro-probing the chip or subjecting the chip to temperature extremes leads to zeroization.

In addition to above, the sole I/O pin is designed in such a way that if an excess voltage is encountered, the I/O pin fuses and renders the chip inoperable.

The U.S. Postal Service's (USPS) Information Based Indicia Program Postal Security Device Specification is intended to permit printing of valid U.S. postage on any PC. This required a combination of two areas of expertise, cryptographic security and high resistance to attack by hackers.

With it zeroization capability and the private key, crypto iButton is one of the least counterfeitable devices. It would destroy itself rather than reveal its secret when tampered. The iButton in the Java Ring is Java Card 2.0 compliant. Java Card is a type of smart card. Next chapter explains what smart cards and Java Cards are and how they work and a closer look at writing applets in Java Card.

The iButton provides different pins such as User PIN, Admin PIN and Master PIN that can be set on the ibutton to control the operations on the iButton.

3.3. Java Card Technology

The iButton is Java Card 2.0 compliant. A Java Card is a type of smart card that is enabled to work with Java Card Technology. To understand how a Java Card works and communicates, we need to know how a smart card works and its communication model. The next section gives an introduction to smart cards and

discusses how smart cards communicate. The Java Card section discusses about Java Card, how Java Card communicates and how the applets run in a Java Card.

3.3.1. Smart Card:

Smart cards are small computing devices that act as tokens to enable services that require security. A smart card is a type of chip card, embedded with a computer chip that store and transact data between users. This data in a smart card is associated with either value or information or both and is stored and processed within the card's chip which is either a memory or microprocessor.

A smart card resembles a credit card in size and shape, but inside it is an embedded 8-bit microprocessor. There are two basic kinds of smart cards: An intelligent smart card contains a microprocessor and offers read, write, and calculating capability, like a small microcomputer. A memory card, on the other hand, does not have a microprocessor and is meant only for information storage. A memory card uses security logic to control the access of memory [12].

In all, there are five types of smart cards:

1. memory cards
2. processor cards
3. electronic purse cards
4. security cards
5. JavaCard

Here processor card is the intelligent smart card.

A smart card can communicate by inserting it into a Card Acceptance Device (CAD), which may be connected to another computer. The Card Acceptance Device can be a terminal, reader, or interface device. They all provide the same basic functions such as supplying the card with power and establishing a data-carrying connection.

Smart Card Communication model:

The Communication takes place in a smart card by inserting Smart Cards into a CAD which is connected to some computer, where the applications reside. These applications are known as host applications. The host applications communicate by sending commands to the applets in the smart cards. These commands are known as Command APDUs (Application Protocol Data Unit). APDU contains either a command or a response message. In this card model, the master-slave model is used whereby a smart card always plays the passive role. In other words, a smart card always waits for a command APDU from a terminal. It then executes the action specified in the APDU and replies to the terminal with a response APDU. Command APDUs and response APDUs are exchanged alternatively between a card and a terminal. A detail look at APDU is given in the Java Card section of the chapter.

Smart Card Applications:

Smart Cards are used in many applications these days. Smart cards greatly improve the convenience and security of any transaction. They provide tamper-proof storage of user and account identity. Smart cards also provide vital components of system security for the exchange of data throughout virtually any type of network. They protect against a full range of security threats, from careless storage of user passwords to sophisticated system hacks. Multifunction cards can also serve as network system access and store value and other data.

Smart Cards are used for many applications. Smart cards can be used with a smart-card reader attachment to a personal computer to authenticate a user. Web browsers can use smart card technology to supplement Secure Sockets Layer (SSL) for improved security of Internet transactions. Smart-card readers can also be found in mobile phones and vending machines.

The most common smart card applications are:

- Credit cards
- Electronic cash
- Loyalty systems (like frequent flyer points)
- Banking
- Satellite TV
- Government identification [14]

3.3.2. OpenCard Framework :

Using a smart card requires an interface for the user to be able to read the card and communicate with it using an application. These interfaces are implemented by OpenCard framework.

OpenCard is an open standard that provides inter-operability of smart card applications across network computers, POS terminals, desktops, laptops, set tops, and PDA's. OpenCard can provide pure Java smart card applications. Smart card applications often are not 100% pure because they communicate with an external device or use libraries on the client.

OpenCard provides a framework by defining interfaces that must be implemented. Applications using smart cards can read and communicate by implementing the interfaces defined by OpenCard framework. Once these interfaces are implemented, other services in the upper layers of the API can be used.

OpenCard Framework architecture:

The architecture of the OpenCard Framework is made up of the CardTerminal, the CardAgent, the Agents and/or applications that interact with these components. OpenCard consists of four Java packages with the prefix opencard:

1. application
2. io
3. agent
4. terminal

The figure below gives an overview of the OpenCard Framework architecture.

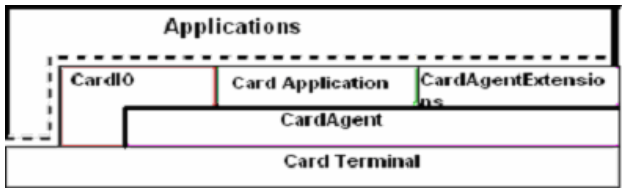

Figure 3.5: OpenCard Framework Architecture

The packages opencard.application and opencard.io provide the high-level API used by the application developer. Classes in opencard.agent and opencard.terminal packages provide the services needed by the high-level API.

The opencard.agent package abstracts the functionality of the smart card through the CardAgent. The opencard.terminal package contains classes to represent the card-terminal hardware, to interact with the user, and to manage card-terminal resources. A card terminal abstracts the device that is used in a computer system to communicate with a smart card.

3.3.3. Introduction to JavaCard

JavaCard was introduced by Schlumberger and submitted as a standard by JavaSoft [15]. Java Card is a smart card with the potential to set the overall smart card standard, and is comprised of standard classes and APIs that let Java applets run directly on a standard ISO 7816 compliant card [15]. Java Cards enable secure and chip-independent execution of different applications.

A Java Card means a smart card that is enabled to work with Java Card Technology. Java Card Technology allows applets written in the Java language to be executed on smart cards. A Java card is a smart card that is able to execute Java byte code, similar to the way Java-enabled browsers can execute. But standard Java with all of its libraries is far too big to fit on a smart card. A solution to this problem is a stripped-down flavor of Java. Java Card is a special, stripped-down version of Java that runs on a smartcard itself. In whole, Java Card Technology provides JCRE (Java Card Runtime Environment) together with other classes and APIs for developers to create applets to be executed on smart cards. It is based on a subset of the Java API plus some special-purpose card commands. Unlike smartcard products which have only one application per card, Java Card allows smart cards to have multiple applications on them. The minimum system requirement is 16 kilobytes of read-only memory (ROM), 8 kilobytes of EEPROM, and 256 bytes of random access memory (RAM).

The system architecture on the Java Card is illustrated in the following figure.

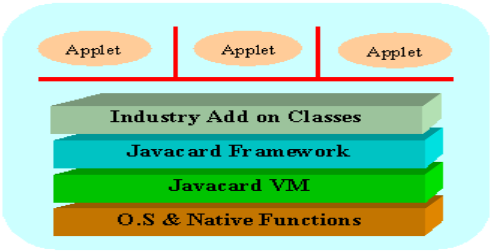

Figure 3.6: Java Card System Architecture

As shown in the figure, the Java Card VM is built on top of a specific integrated circuit (IC) and native operating system implementation. The JVM layer hides the manufacturer's proprietary technology with a common language and system interface. The Java Card framework defines a set of Application Programming Interface (API) classes for developing Java Card applications and for providing system services to those applications. Add-on libraries to provide a service or to refine the security and system model are supplied by specific industry or business supplies. Java Card applications are called applets. Multiple applets can reside on one card. Each applet is identified uniquely by its AID (application identifier), as defined in ISO 7816 [19].

The Java Card virtual machine separates applications from the underlying hardware and operating system. The Java Card platform's standardized API provides a uniform interface to disparate smart cards. This unique approach uses the widely-understood benefits of object-oriented programming to enable security at both the application and platform level [22].

Due to limited memory resources and computing power, the Java Card specifically, does not support:

- Dynamic class loading

- Security manager
- Threads and synchronization
- Object cloning
- Finalization
- Large primitive data types (float, double, long) and char data type [17]

3.3.4. Java Card and OpenCard:

An OpenCard Framework is Java in the computer or terminal talking to a smartcard. Java applications running on a PC can use OpenCard to access Java Card smart cards and standard smart cards. Java applets (also known as cardlets) can be written and run on Java Card which is compliant with the Java Card standard. OpenCard is the ideal host-side application framework for accessing Java Card. Any smart card to access Java Card needs a card service which supports the interfaces of Java Card applet [13].

3.3.5. Java Card Applets:

When a Java Card is inserted, the Card Acceptance Device (CAD) accepts the Java Card and selects an applet which sends a series of commands to execute. Each applet in a Java Card is identified and selected by its unique Application Identifier (AID). Commands are formatted and transmitted between the application and the applets using Application Protocol Data Units (APDU). Applets reply to each APDU command as status words. Applets can optionally reply to an APDU with other data. The communication between the applet and the application are discussed in detail in the next part.

Briefly, applet designing requires:

- Specifying the working functionality of the applet
- Requesting and assigning AIDs to both the applet and the packages containing the applet class
- Designing the class structure of the program and
- Defining the interface between the applet and the terminal application.

Interface between an Applet in Java Card and Its Terminal Application:

The APDU is like an interface between an applet and the application hosted on the CAD. All the communication between an applet and the application hosted on the Cad is carried by the APDU.

APDU (Application Protocol Data Unit):

- APDU commands are always a set of pairs. Each pair contains a Command APDU and a Response APDU. A Command APDU specifies a command sent by the application through a CAD, and response APDU specifies the result executed by the applet.
- The terminal application sends a command APDU through the CAD. The JCRE receives the command and either selects a new applet or passes the command to the currently selected applet, which processes the command and returns a response APDU to the terminal application. Command APDU and response APDU are exchanged alternately between a card and a CAD.
- APDU Format

Command APDU						
Mandatory header				Optional header		
CLA	INS	P1	P2	Lc	Data Field	Le

CLA (1 byte): Class of instruction -- indicates the structure and format for a category of command and response APDUs.

INS (1 byte): Instruction code: specifies the instruction of the command.

P1 (1 byte) and P2 (1 byte): Instruction parameter.

Lc (1 byte): Number of bytes present in the data field of the command.

Data field (bytes equal to the value of Lc): Data in the form of sequence of bytes.

Le (1 byte): Maximum of bytes expected in the data field of response command.

Response APDU

Optional Body	Mandatory trailer	
Data field	SW1	SW2

Data field (variable length): A sequence of bytes received in the data field of the response.

SW1 (1 byte) and SW2 (1 byte): Status Words -- denote the processing state in the card.

Figure 3.7: Command and Response APDU Structure

CLA - The CLA field is meant to be used as control data. Normally, each applet has one CLA. A normal use of the CLA field is to insure that the host is talking to the correct applet. For example, normally, the first thing that occurs in the process method is to check whether the CLA in the commandAPDU just received matches the CLA of that applet. If not, it should return with an error. Example:

```
final static private byte THIS_APPLET_CLA = (byte)0x80;
...
public void process(APDU apdu)
{
   byte[] buffer = apdu.getBuffer();
   //Check for a valid CLA.
   if(buffer[ISO.OFFSET_CLA] != THIS_APPLET_CLA)
   {
      ISOException.throwIt(ISO.SW_CLA_NOT_SUPPORTED);
   }
   ...
```

INS - The INS field is meant to be used to tell the applet what instruction the host wishes to be performed. The particular value of the instruction bytes do not matter, as long as the applet and host both know what numbers each instruction corresponds to. For example, both a host and its corresponding applet might have the following declarations:

```
// BEGIN INSTRUCTION DECLARATIONS

public static final byte BASICS_CLA = (byte)0x80;
public static final byte BASICS_INS_STORE_NUMBER = (byte)0;
public static final byte BASICS_INS_GET_NUMBER = (byte)1;

// END INSTRUCTION DECLARATIONS
```

The process method would perform the appropriate instruction by doing a switch on the INS field of the apdu, calling the appropriate method:

```
//Call the appropriate dispatch method for the given INS.
switch (buffer[ISO.OFFSET_INS])
{
   case BASICS_INS_STORE_NUMBER:
      store_numberDispatch(apdu, buffer[ISO.OFFSET_P1],
                           buffer[ISO.OFFSET_P2]);
   break;
```

```
        case BASICS_INS_GET_NUMBER:
           get_numberDispatch(apdu, buffer[ISO.OFFSET_P1],
                              buffer[ISO.OFFSET_P2]);
        break;

        default:
           ISOException.throwIt(ISO.SW_INS_NOT_SUPPORTED);
     }
```

P1 & P2 - The P1 and P2 fields are normally used as additional control data. For example, if you had an instruction to sort a byte array on the iButton with the option to sort forwards or backwards. You may not want to break that up into separate instructions. Instead, you could pass the same instruction, and set a value of P1 or P2 to indicate how the array should be sorted.

DATA - This field contains the data (in the form of a byte array) sent. An example of how to access this data follows:

```
public void process(APDU apdu)
{
   ...
   byte[] buffer = apdu.getBuffer();
   apduData = new byte[buffer[ISO.OFFSET_LC] & 0x0FF];
   short apduDataOffset = 0;
   //Read in the entire APDU.
   short bytesRead = apdu.setIncomingAndReceive();
   //Loop until all bytes have been read.
   while (bytesRead > 0)
   {
      Util.arrayCopyNonAtomic(buffer, ISO.OFFSET_CDATA, apduData,
      apduDataOffset, bytesRead);
      apduDataOffset += bytesRead;
      bytesRead = apdu.receiveBytes(ISO.OFFSET_CDATA);
   }
   /**********************************************
    *  The byte array apduData now contains the
    *  data sent from the host to the applet.
    **********************************************/
   ...
```

Functionality inside a Java Card:

Inside a Java Card, JCRE (Java Card Runtime Environment) refers to the Java Card virtual machine and the classes in the Java Card Framework. JCRE assigns the unique AID to each applet within a Java Card. After an applet is correctly loaded into the card's persistent memory and linked with the Java Card Framework and other libraries on the card, JCRE calls the applet's install method as the last step in the applet installation process. A public static method, install, must be implemented by an applet class to create an instance of the applet and register it with JCRE.

An applet on the card remains inactive until it is explicitly selected. The terminal sends a select APDU command to JCRE. JCRE suspends the currently selected applet and invokes the applet's deselect method to perform any necessary cleanup. JCRE then marks the applet whose AID is specified in the select APDU command as the currently selected applet and calls the newly selected applet's select method. The select method prepares the applet to accept APDU commands. JCRE dispatches the subsequent APDU commands to the currently selected applet until it receives the next select APDU command.

3.4. Programming in iButton :

Programming for the iButton requires writing both a host application and an iButton applet. These two are completely separate Java programs that will communicate with each other but be executed on two different machines.

The host application resides on a personal computer or embedded system. Its function is to send control instructions and data to the iButton applet, collect and interpret the response data received, and output the results in some form to the user. A detailed explanation of the structure of a host application is given in the Host Application Structure part of this section.

The applet is downloaded and run on the iButton itself. Once installed and selected to run, it waits to process host instructions. A detailed explanation of the structure of an iButton applet is given in the iButton Applet Structure part of this section.

3.4.1. Host Application Structure:

The host application is responsible for controlling the iButton applet. It allows a user to interact with the applet, sending command instructions and displaying output.

The host code, with full security access, has complete control of the iButton. It can retrieve and erase an iButton's contents, download an applet, or select a particular applet already installed on the button to run.

Host Interface

A host application must implement opencard.core.event.CTListener. This allows the host application to receive events when an iButton is inserted or removed. A host that implements CTListener must implement the methods cardInserted and cardRemoved. A CardTerminalEvent is the parameter to these methods, and can be used to obtain a SlotChannel, which is used to send APDU's to the iButton.

Host and Applet Communication

As mentioned before, because iButton is Java Card 2.0 compliant, the host and applet in iButton communicate using APDUs. The host sends CommandAPDUs to the iButton, each containing an instruction and any data which needs to be sent. The iButton processes the instructions and sends a ResponseAPDU back to the host that contains any data to be returned plus a status word that indicates whether or not the instruction completed successfully. The data in the command and the response APDU is sent in the form of a byte array. A successful

execution, which means no errors or exceptions, occurred in processing the instruction is indicated with a status word of 0x9000 [21].

The host application must know the class and instruction bytes of each iButton applet it expects to control. These bytes are passed in the CommandAPDU header and will tell the applet what action to perform. The class byte is generally used as identification for the applet. Normally, most applets have one class byte (usually named CLASSNAME_CLA) that it references each time a CommandAPDU arrives from the host application. If the class byte sent from the host doesn't match the class byte of the applet, it throws an ISO.SW_CLA_NOT_SUPPORTED exception. In concert with the instruction byte, it can be used to act as additional control data.

Since a host application can't make a remote function call on the iButton, it has to send commands to indicate what functions to call or what actions to perform. The instruction byte of the CommandAPDU carries this information. Suppose an applet has designated the byte 0x04 to perform 'exclusive or' on two hardcoded integers, the host knows it wants the iButton to perform this action, it would send a CommandAPDU with applet's class byte and an instruction byte of 0x04.

The structure of a Host Application is shown below: [21]

Minimal Host Application, OpenCard API

```
import opencard.core.event.*;
import opencard.core.service.*;
import opencard.core.terminal.*;
import opencard.opt.applet.*;
import java.util.*;
import java.io.*;
public class ocf_Host implements CTListener
```

```
{
   public static final byte OCF_CLA = (byte)0x80;
   public static final byte OCF_INS_EXECUTE = (byte)0;
   /**
    * Sets up the listener for iButton inserted and iButton
    * removed events.
    *
   * @param appletPath  the path to the applet that should
    *                    be loaded into the iButtons.
    * @param appletName  the name of the applet that should
    *                    be loaded into the iButtons.
    */
   public ocf_Host()
   {
      //************************************
      //* Add any initialization code here. *
      //************************************
      try       {
         opencard.core.service.SmartCard.start();
         CardTerminalRegistry reg = CardTerminalRegistry.getRegistry();
         reg.addCTListener(this);
         reg.createEventsForPresentCards(this);
      }      catch(Exception e)
      {
         System.out.println("Caught an exception: "+e.toString());
         System.exit(0);
      }
   }
   /**
    * Called when an iButton gets inserted.
    *
    * @param event  the insertion event.
    */
   public void cardInserted(CardTerminalEvent ctevent)
   {
      System.out.println("Card has been inserted");
      SlotChannel sc = null;
      try       {
         /************************************************
          * Note that a SmartCard object and a SlotChannel  *
          * object cannot both exist at the same time.  One *
          * must close so you can open the other!          *
```

```
         **************************************************/
         CardTerminal ct = ctevent.getSlot().getCardTerminal();
         int slotid = ctevent.getSlot().getSlotID();
         Object lock = new Object();
         sc = ct.openSlotChannel(slotid, lock);
         //***************************************
         //* Insert any code to be done when      *
         //* an iButton is inserted here,         *
         //* using the SlotChannel object 'sc'    *
         //***************************************
         executeDispatch(sc);
      }      catch(CardTerminalException cte)
      {
         System.err.println("ERROR:    CardTerminalException   occurred   while
communicating with iButton.");
         cte.printStackTrace();
      }      catch(IOException ioe)
      {         System.out.println("IO Exception");
         ioe.printStackTrace();
      }      catch(Exception e)
      {

        //Exceptions that occur in iButtonInserted events
         //will be drained in the OpenCard internals if we
         //don't catch them here.
         System.err.println("Exception in cardInserted:");
         e.printStackTrace();
      }
      finally      {
         //we must ALWAYS close the slot channel!!!
         try
         {    if (sc!=null)
            sc.close();
         }          catch(CardTerminalException cte)
        {
           // drain
         }
      }
   }
   /**
```

iButton Applet Structure:

The iButton applet runs on the iButton itself. It receives and executes instructions from the host. After an applet has been downloaded to the iButton and selected, it waits for Command APDU's to be sent from the host. When an APDU is received, the applet's process method is called to handle the command. The process method should perform the correct function for the instruction contained in the APDU and will automatically return a Response APDU. This response APDU contains any data the applet writes out, and a status word indicating the success or failure of executing the instruction.

Data Types

The iButton applet has the following data types available: int, short, byte, boolean, and one dimensional arrays. There is no String data type available on the iButton. String data should be saved into a byte array in order to be used in an iButton applet.

Structure:

An applet written to run on the Java ring follows a very simple structure. The iButton applet must extend javacard.framework.Applet and must override the constructor, the install method, and the process method. The constructor must first make a call to the register() function, which registers this applet with the JCRE (Java Card Runtime Environment). The process of writing an applet is very similar to applet programming in that there are several methods that you have to override. These methods are:

- deselect -- another applet, or possibly this one, is about to be selected
- install -- installs the applet
- process -- incoming APDUs arrive here

- register -- register applet

- select -- called when a select command is received

The static install method should create a new instance of this applet. The process method should perform the appropriate function based on the instruction passed in the APDU. The process method will be called when the applet is selected. So it should check to see if selection is the reason it is being called. (If the applet is being selected, then the CLA will be 0x00, and the INS will be 0xA4.)

The workings of the process() with the APDU is shown in the figure below. The APDU commands are sent from the host (client) application as shown in the figure below.

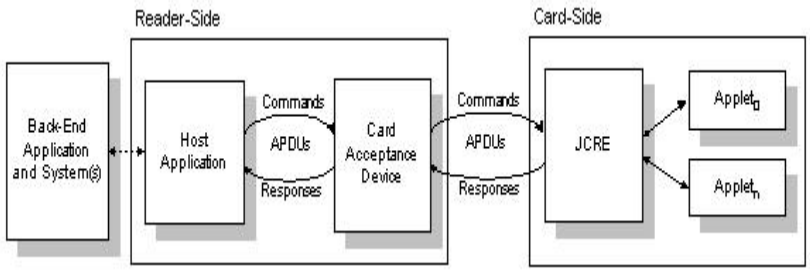

Figure 3.8: Workings of the APDU from the reader side to the card side

The basic structure of an iButton applet is as shown below

```
import javacard.framework.*;
public class Basics_Applet extends Applet
{
   ...
  public Basics_Applet()
   {
      //Register this applet with the JCRE
      register();
```

```
    //*************************************
    //* Add any initialization code here. *
    //*************************************
}
public static void install(APDU apdu)
{
    new Basics_Applet();
}
public void process(APDU apdu)
{
    byte[] buffer = apdu.getBuffer();

    //Determine if the applet is being selected.

    if((buffer[ISO.OFFSET_CLA] == SELECT_CLA) &&
    (buffer[ISO.OFFSET_INS] == SELECT_INS))
    {
        //***********************************
        //* Add any code to be executed on *
        //* applet selection here. *
        //***********************************
        return;
    }
    //Check for a valid CLA.
    if(buffer[ISO.OFFSET_CLA] != BASICS_CLA)
    {
        ISOException.throwIt(ISO.SW_CLA_NOT_SUPPORTED);
    }
    else        {
        //Call the appropriate dispatch method for the given INS.
        switch (buffer[ISO.OFFSET_INS])
        {
            case BASICS_INS_STORE_NUMBER:
                store_numberDispatch(apdu, buffer[ISO.OFFSET_P1],
                                     buffer[ISO.OFFSET_P2]);
            break;

            case BASICS_INS_GET_NUMBER:
                get_numberDispatch(apdu, buffer[ISO.OFFSET_P1],
                                   buffer[ISO.OFFSET_P2]);
            break;
```

```
            default:
               ISOException.throwIt(ISO.SW_INS_NOT_SUPPORTED);
         }
      }
   }
   ...
```

Java developers must override at least the install and process methods of the super class, Applet. In simple applets such as Business Card discussed next, install simply constructs a new instance. It is the job of the constructor to register the applet with the Java Card runtime environment (JCRE).

The following code segment demonstrates how to override the install method. [9] The install method is normally used to set up the applet environment. Simple applets may be ready to run after calling install. More complicated applets may require additional initialization sequences. The code in the process method is not given below.

```
class BusinessCard extends Applet {
  public BusinessCard() {
    // Register our applet with the JCRE
    register();
  }
  public static void install(APDU apdu) {
    new BusinessCard();
  }
  public void process(APDU apdu) throws ISOException {
    .
    .
    .
  }
  .
  .
  .
}
```

In order to communicate with the host, the card terminal, the applet must implement the process method. This method is invoked on the selected applet

whenever a command APDU is received from the host. Following is a segment of BusinessCard's implementation of the process method. Its structure is typical of Java Card applets.

```
public void process(APDU apdu) throws ISOException {
  byte[] buffer = apdu.getBuffer();
  // process selects separately
  .
  .
  .
  if (buffer[ISO.OFFSET_CLA] != BC_CLA) {
    // Don't know what to do with this instruction
    throw new ISOException(ISO.SW_CLA_NOT_SUPPORTED);
  }
  else {
    switch (buffer[ISO.OFFSET_INS]) {
    // Store new business card data
    case BC_INS_STORE:
      businessCardStore(apdu);
      break;
    // Send business card data to the host
    case BC_INS_RETRIEVE:
      businessCardRetrieve(apdu);
      break;
    // Don't know what to do with this instruction
    default:
      throw new ISOException(ISO.SW_INS_NOT_SUPPORTED);
    }
  }
}
```

As mentioned before, the main purpose of the process method is to invoke the correct dispatch method. If process has trouble understanding the header, it will throw an instance of ISOException with the appropriate status code.

3.5. Java Card Applet Example:

The following example is an electronic wallet application, which stores electronic cash. The wallet handles read_balance, deposit, and debits APDU commands. Access to the wallet is authenticated by an owner PIN [17].

The example is formatted in two columns: The left column contains Java code with Java style comments; the right column provides further explanation of the code that it lines up with on the left side.

package bank.purse;	Java Card supports package and identifier name convention as in standard Java
import javacard.framework.*; import javacardx.framework.*;	
public class Wallet extends Applet { /* constants declaration */	An applet is an instance of a class which extends from javacard.framework.Applet

// code of CLA byte in the command APDU header
 	
final static byte Wallet_CLA =(byte)0xB0;
	CLA identifies the application

// codes of INS byte in the command APDU header
final static byte Deposit = (byte) 0x10;
final static byte Debit = (byte) 0x20;
	INS specifies the

final static byte Balance = (byte) 0x30; final static byte Validate = (byte) 0x40;	application instructions

// maximum number of incorrect tries before the PIN is blocked
final static byte PinTryLimit =(byte)0x03;
// maximum size PIN
final static byte MaxPinSize =(byte)0x04;
	PIN object parameters

// status word (SW1-SW2) to signal that the balance becomes negative; final static short SW_NEGATIVE_BALANCE = (short) 0x6910;	Applet specific static word
/* instance variables declaration */ OwnerPIN pin; byte balance; byte buffer[]; // APDU buffer	
private Wallet() { // It is good programming practice to allocate	private constructor -- an instance of class Wallet is instantiated by its install

// all the memory that an applet needs during its
// lifetime inside the constructor
pin = new OwnerPIN(PinTryLimit, MaxPinSize);
balance = 0;
register();
} // end of the constructor
	method. Applet registers itself with JCRE by calling register method, which is defined in class Applet. Now the applet is visible to the outside world

public static void install(APDU apdu){
// create a Wallet applet instance
new Wallet();
} // end of
install method
	Method install is invoked by JCRE as the last step in the applet installation process

public boolean select() {
// reset validation flag in the PIN object to false
pin.reset();
// returns true to JCRE to indicate that the applet
// is ready to accept incoming APDUs.
	This method is called by JCRE to inform that this applet has been selected. It performs necessary initialization which is required to process the following APDU

return true; }// end of select method	messages.

public void process(APDU apdu) {
 	
// APDU object carries a byte array (buffer) to
// transfer incoming and outgoing APDU header
// and data bytes between card and CAD
buffer = apdu.getBuffer();
	After the applet is successfully selected, JCRE dispatches incoming APDUs to this method.
	
	APDU object is owned and maintained by JCRE. It encapsulates details of the underlying transmission protocol (T0 or T1 as specified in ISO 7816-3) by providing a common interface.

// verify that if the applet can accept this
// APDU message
if (buffer[ISO.OFFSET_CLA] !== Wallet_CLA)
ISOException.throwIt
(ISO.SW_CLA_NOT_SUPPORTED);
	When an error occurs, the applet may decide to terminate the process and throw an exception containing status word (SW1 SW2) to indicate the processing state of the card.
	
	An exception that is not

	caught by an applet is caught by JCRE.

switch (buffer[ISO.OFFSET_INS]) {
case Balance: getBalance(apdu); return;
case Debit:
 	
debit(apdu); return;
case Deposit: deposit(apdu);return;
case Validate: validate(apdu);return
 	
default:
 	
ISOException.throwIt
(ISO.SW_INS_NOT_SUPPORTED);
 	
}
 	
} // end of process method
	The main function of process method is to perform an action as specified in APDU
	
	and returns an appropriate response to the terminal.
	
	INS byte specifies the type of action needs to be performed

private void deposit(APDU apdu) {
 	
// access authentication
if (! pin.isValidated())
ISOException.throwIt (ISO.SW_PIN_REQUIRED);
	The parameter APDU object contains a data field, which specifies the amount to be added onto the balance.

// Lc byte denotes the number of bytes in the data
 	
// field of the comamnd APDU
byte numBytes = (byte) (buffer[ISO.OFFSET_LC]);
 	
// indicate that this APDU has incoming data and
// receive data starting from the offset
// ISO.OFFSET_CDATA
byte byteRead = (byte)(apdu.setIncomingAndReceive());
 	
// it is an error if the number of data bytes read does
 	
not
// match the number in Lc byte
if (byteRead != 1)
ISOException.throwIt(ISO.SW_WRONG_LENGTh);
 	
// increase the balance by the amount specified in the
// data field of the command APDU.
balance = (byte)
(balance + buffer[ISO.OFFSET_CDATA]);
 	
// return successfully
return;
 	
} // end of deposit method
	Upon receiving the APDU object from JCRE, the first 5 bytes (CLA, INS, P1, P2, Lc/Le) are available in the APDU buffer. Their offsets in the APDU buffer are specified in the class ISO. Because the data field is optional, the applet needs to explicitly inform JCRE to retrieve additional data bytes.
	
	The communication between card and CAD is exchanged between command APDU and response APDU pair. In the deposit case, the response APDU contains no data field. JCRE would take the status word 0x9000 (normal processing) to form the correct response APDU. Applet developers do not need to concern the details of constructing the proper

	response APDU. When JCRE catches an Exception, which signals an error during processing the command, JCRE would use the status word contained in the Exception to construct the response APDU.

private void debit(APDU apdu) {
 	
// access authentication
if (! pin.isValidated())
ISOException.throwIt(ISO.SW_PIN_REQUIRED);
 	
byte numBytes = (byte)(buffer[ISO.OFFSET_LC]);
byte byteRead = (byte)(apdu.setIncomingAndReceive());
 	
if (byteRead != 1)
ISOException.throwIt(ISO.SW_WRONG_LENGTH);
 	
// balance can not be negative
if ((balance - buffer[ISO.OFFSET_CDATA]) < 0)
ISOException.throwIt(SW_NEGATIVE_BALANCE);
 	
balance = (byte)
	In debit method, The APDU object contains a data field, which specifies the amount to be decrement from the balance

(balance - buffer[ISO.OFFSET_CDATA]); } // end of debit method	

private void getBalance(APDU apdu) {
 	
// access authentication
if (! pin.isValidated())
ISOException.throwIt(ISO.SW_PIN_REQUIRED);
 	
// inform system that the applet has finished processing
// the command and the system should now prepare to
// construct a response APDU which contains data field
apdu.setOutgoing();
 	
// indicate the number of bytes in the data field
apdu.setOutgoingLength((byte)1);
 	
// move the data into the APDU buffer starting at offset 0
buffer[0] = balance;
 	
// send 1 byte of data at offset 0 in the APDU buffer
apdu.sendBytes((short)0, (short)1);
 	
} // end of getBalance method
	getBalance returns the Wallet's balance in the data field of the response APDU.
	
	Because the data field in response APDU is optional, the applet needs to explicitly inform JCRE of the additional data. JCRE uses the data array in the APDU object buffer and the proper status word to construct a complete response APDU.

private void validate(APDU apdu) {
 	
// retrieve the PIN data which requires to be valid ated
// the user interface data is stored in the data field of the
 	
APDU
byte byteRead = (byte)(apdu.setIncomingAndReceive());
 	
// validate user interface and set the validation flag in the
 	
user interface
// object to be true if the validation succeeds.
// if user interface validation fails, PinException would be
 	
// thrown from pin.check() method.
pin.check(buffer, ISO.OFFSET_CDATA, byteRead);
 	
} // end of validate method
 	
} // end of class Wallet
	PIN is a method commonly used in smart cards to protect data from unauthorized access
	
	A PIN records the number of unsuccessful tries since the last correct PIN verification. The card would be blocked, if the number of unsuccessful tries exceeds the maximum number of allowed tries defined in the PIN.
	
	After the applet is successfully selected, PIN needs to be validated first, before any other instruction can be performed on the applet

Chapter 4

FINGERPRINT AUTHENTICATOR

4.1. Introduction to Fingerprint Authenticator:

Biometrics is defined in the security industry as a measurable physical characteristic or personal behavioral trait used to recognize the identity or verify the claimed identity of a person and biometric identification is the use of computers to confirm the identity of a user [24].

Unlike other ways of authentication such as passwords – something a person knows, security device – something a person possess, biometrics deals with something a person is. While a password and a security device can be stolen, a biometric cannot be stolen and is always with you. Biometrics have proven to be an effective solution for high-security access control, ensuring that only authorized individuals can access protected or secure data. Biometric systems require controlled and accurate enrollment processes, careful monitoring of security settings to ensure that the risk of unauthorized entry is low and well-designed interfaces to ensure rapid acquisition and matching.

There are many types of biometrics available such as fingerprint recognition, voice recognition, face recognition, retina, iris and DNA. Given below are some of the biometrics used for authentication purposes [24].

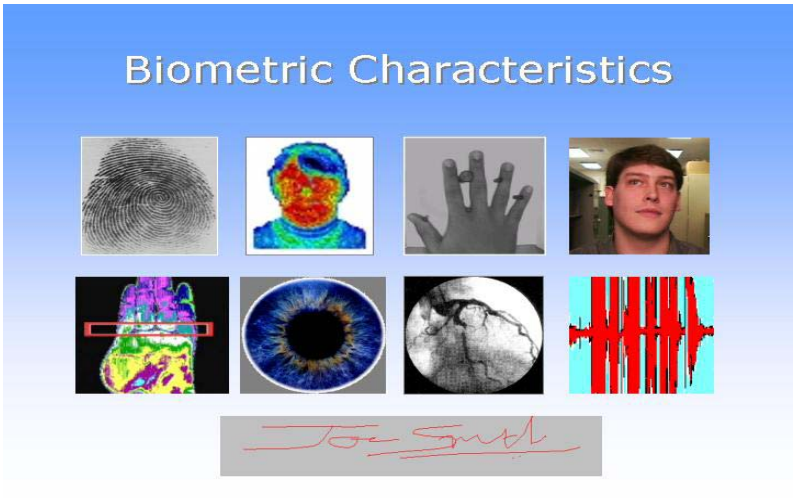

Figure 4.1: Different Biometrics

The first image is a finger print, second a spatial thermogram where an infrared image of the face is obtained by the heat emitted from the face. The third image shows the hand geometry, fourth face recognition, fifth heat emitted by the hand, sixth iris, seventh retina scan where a light is used to scan the retina, eight speech recognition and ninth signature.

The next image shows how most of the biometrics can have complexities involved and cannot be accurate all the time. The first image shows how complex face recognition system works as it has to consider different aspects such as taking images of the person from different angles as well take the different expressions of the person into consideration. The second image shows the handwriting of a person at different times and different conditions (e.g., when the person is ill or when a person is drunk). In this case, it reveals more information about the person than required. Using DNA for authentication also has the same drawback of revealing more information about the person. The next image shows how a person is disguised in different ways and can fool a face recognition system. Hence, face recognition also has its set of complexities.

Figure 4.2: Face Recognition and Handwriting [24]

As the complexity involved with fingerprint is not as much as with other kinds of biometric authentication, fingerprint authentication is considered a better way of authentication. Since different fingers have different ridges and characteristics, these minute details help to identify a person and do not reveal other information about the person than required. These minute details are permanent for each person. Even identical twins have different fingerprints. Fingerprints have a long history and are considered to be unique. Fingerprint impressions in clay tablets over 2000 years old have been seen in archaeological materials from both China and the Middle East. Fingerprinting received a scientific basis through work performed in the 19th and 20th century by a wide variety of researchers and institutions [28].

The problem with fingerprints is that the fingerprint sensor can be fooled. Here are some of the cases where fingerprint sensors were fooled.

- Dr. Matsumoto used procedures to create gelatin 'gummy fingers' that possessed the same fingerprint geometry and minutiae as a live finger.

This can be done by lifting a latent print from a sensor or spoofing with an easily crafted gummy finger made of a home made gelatin mold [25]. Fingerprints are lifted from objects such as a coffee cup and a gummy finger is created and touched up using a microscope.

- Reactivating a latent fingerprint by placing a water-filled plastic bag on the sensor or brushing graphite powder on the sensor and applying pressure to an adhesive film on top of the powder are some of the other ways of fooling a fingerprint sensor [26].

- Latent prints placed on a transparency by simply pressing on the surface. The prints are increased in clarity and contrast by using a black latent print powder. After brushing away the excess powder, the latent fingerprint is lifted using scotch tape. The tape and print is then placed on to the sensor with the sticky side down and pressure is applied to activate the sensor. [26].

- A Milpitas, California-based company claims to have addressed some fingerprint fooling methods with a technology that relies on a combination of a new algorithm and monitoring of physical changes to the optical sensor reading the print [25].

- Factors such as calluses, dryness, moisture or the affects of aging can affect the image capturing of the fingerprint sensor.

There are many fingerprint authentication products available in the market. The fingerprint sensor used in this thesis application is an Authentec AES 4000.

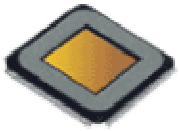

Figure 4.3: The Entrepad Family Authentec AES4000

This fingerprint uses Trueprint technology.

4.1.1. TruePrint Technology:

TruePrint based fingerprint sensors are small components that can be easily designed into almost any electronic device. The user simply places his finger flat on the sensor surface to activate the system. The sensor generates an image of the pattern in the finger skin that touches it.

TruePrint technology sensors can capture images from beneath the surface of the skin where the ridge-and-valley pattern suffers less damage from day-to-day living. It is this technology that does not let the fingerprint sensor be fooled by a gummy finger or any of the sensor fooling methods mentioned before. By looking below the surface layer of the skin, TruePrint Technology reads the real fingerprint, producing an unaffected, undistorted image, thus avoiding the limitations of previous techniques where the upper layer of the finger is scanned. Unlike prior approaches, skin surface conditions such as calluses, dryness, moisture or the effects of aging do not limit the image capturing ability. Contaminants such as ink, paint or glue have little or no effect as it is the second layer that is scanned.

4.1.2. Security concern with storing fingerprints:

The reality is that once personal information such as a fingerprint has been provided to an external medium the individual no longer has the capacity to

control who will be able to peruse or access it. To date, this lack of control has been subject to the limitations of regulation and of the technology itself.

The authentication system in this project provides a solution. It allows individuals to control the access themselves - thus rendering individuals no longer impotent to the vulnerability of computers, databases and software or to accidents, malfunction or intrusion. This project aims to hand the individual back control of their identity. This is done by storing the fingerprint in the iButton of the Java Ring which an individual carries with him. This is discussed in the implementation part in chapter 5.

4.2. Authentec API:

The fingerprint system used in this thesis provides the developers of the AuthenTec fingerprint biometrics with AuthenTec Windows Fingerprint System (AWFS) API library. In further sections of the thesis AuthenTec Windows Fingerprint System is referenced as AWFS.

The next part of this chapter goes through the fundamentals of AWFS API and discusses all the terms and functions needed in the thesis.

The AWFS API supports a comprehensive set of functions for single application using a single sensor. It provides functions for fingerprint database management to store user and fingerprint data and algorithms for extracting template data from fingerprint images and for matching fingerprint template data.

Below are some of the definitions and terminology used through the project.

Binary Large Object (BLOB): In the context of the AWFS, a BLOB is an application generated array of bytes. An application can specify the size of (in bytes) and save, retrieve or delete a BLOB of information data for a user. When

an enrollment takes place, the template is stored in BLOB. The AWFS allows an application to store arbitrary information (in addition to templates) for a user in the form of a BLOB in the AWFS database.

Enrollment: The process by which the reference template for a user is constructed and stored in a database is called Enrollment. This is done by collecting one or more images from the fingerprint sensor, extracting salient feature data and combining these results to make a template.

Identification: A match operation in which a fingerprint is compared to templates in an AWFS database to determine the identity, i.e., name and finger number, associated with the fingerprint.

Image Item: An image item is an opaque structure that is used by the AWFS to pass image data to and receive image data from an application. An image item contains pixel data, sizing data and other AWFS proprietary data.

Match Template: A template that is created during an authentication operation from one or more images acquired from the sensor is called a Match template. During authentication, match templates are compared to reference templates to determine if they represent the same fingerprint.

Reference Template: A template created during the enrollment process is called a Reference Template.

Validation: A special case of identification where the template of a user has to be specified is called Validation. A fingerprint is matched against only those templates in the AWFS database for a specified user to determine if there is a match.

Verification: A match operation in which a finger print is matched against an application-supplied list of templates. An application will use a verification operation when it maintains reference templates in its own database.

Sensor Interface:

The AWFS software can support from 1 to 36 physical sensors attached to a single system. This application uses a single sensor

```
AT_RESULT_CODE ATOpenSensor (
 TCHAR*   pszStrSensorName,
  int16   AccessMode )
```

An application must call *ATOpenSensor()* to open an AWFS sensor prior to performing any sensor access functions. The first parameter, *pszStrSensorName*, in the function is the name of the sensor to open. In the most common usage case, a system with a single unnamed sensor, an application will pass NULL for the sensor name. If *pszStrSensorName* is non-NULL, it must be the name assigned to the sensor by running the ATSensorWizzard.

Database Services:

A proprietary database is provided by AWFS to store user data and templates extracted from fingerprint images. AWFS database stores the templates during the enrollment process or by database update procedures. The AWFS database can be managed and queried using the functions provided by the AWFS API. Information stored in AWFS proprietary databases is fully encrypted to maintain security. An application can use its own template storage database instead of AWFS database. In such case, the application must present fingerprint templates to the AWFS system at the beginning of a matching operation.

Some of the features of the AWFS database are as follows:

Shareable and Exclusive Use Databases: An application using an AWFS database can open it with either shared or exclusive access. When more than one application need the same user data and need to access the same database, shared access database is used. A database can be opened with exclusive access when an application wants to preclude use of the database by other applications or processes.

Support for Binary Large Objects: The AWFS database provides functions to save and retrieve application-defined binary data for a user. The AWFS is not concerned with the data content of this user information and deals with this user data as a Binary Large Object (BLOB). A BLOB has no structure that can be interpreted by the AWFS and is known only by its size in bytes [30].

Deletion of Stored Data: The AWFS provides functions to delete specified templates for a user, to delete all data for a user and to delete all data for all users.

Database Query Support: The AWFS provides functions to get a count of the number of users in the database, to obtain a list of all users enrolled in the database and to obtain a list of fingers enrolled for a given user.

Some of the functions of AWFS database are creating a database, opening a database, closing a database. An application using an AWFS database must open it before performing any database read, write or delete functions. An error will be returned to the application if it fails to open a database prior to calling an AWFS API function. An application calls *ATOpenDatabase()* to create a new database or to open an existing database.

AT_RESULT_CODE ATOpenDatabase(

```
TCHAR* pszStrDatabaseFile,
int32       iAccessMode,
uint32      iMemorySize,
void*       pAuthenRec,
uint32      iSizeAuthenRec)
```

The first parameter, *pszStrDatabaseFile*, defines the pathname and filename of the file to be used for persistent database storage. A new database will be created if the specified filename does not exist. If the filename exists, the specified database is loaded for use.

The second parameter, *iAccessMode,* specifies the database access mode as either *AT_DATABASE_ACCESS_SHARE* or *AT_DATABASE_ACCESS_EXCLUSIVE*. The access mode is established by the first application or process that opens a named database. The established access mode for a named database remains in effect until the last application or process that has opened the database closes it. In shared access, an application can receive notification when the database is modified by another application. A database size can optionally be specified in the *ATOpenDatabase()* call.

The third parameter, *iMemorySize*, which defines the database size is used to apprise the AWFS of the amount of memory to allocate for the database. This is useful in case where there are a large number of users and fingers to be enrolled. It is more efficient to initially allocate a large block of memory than to reallocate memory on-the-fly as users are added [30]. The *ATGetEstimatedDatabaseSize()* API function provides an estimated database size based on the number of users, templates, BLOBS, etc. it will maintain. If an application specifies zero for the memory size parameter in the *ATOpenDatabase()* call, the AWFS will use default sizing values to allocate database memory buffers. The AWFS will automatically increase the size of its database memory buffers as needed when new data are added.

When a database is created, an optional application-defined certificate of authenticity is assigned to a database by specifying the certificate in the *ATOpenDatabase()* function. A certificate is specified if *pAuthenRec* is non-NULL and *iSizeAuthenRec* is non-zero. This certificate must be supplied in all subsequent *ATOpenDatabase()* calls for that database, regardless of whether the database is being opened by the creating application or by another application. A database that is not created with a certificate of authenticity will fail to open if a certificate is supplied when a subsequent attempt to open it is made.

ATCloseDatabase() should be called to close an open database. This function is called once the database is no longer required prior to termination of the application.

The AWFS maintains the database in memory and automatically saves the database to disk at the end of any operation that modifies the content of the database. For example, the database is saved after enrollment, the writing of a user BLOB and insertion or deletion of data [30].

The AWFS provides a suite of extraction and matching algorithms. These algorithms are used within AWFS enrollment and matching (identification, validation and verification) functions. An application can be designed to use AWFS algorithms or it can use its own proprietary algorithms. In the latter case, AWFS is used for image acquisition only.

Transactions:

Enrollment, authentication and image acquisition are some of the high-level biometric operations. These operations involve reading and processing a series of images from the AuthenTec sensor. There might be a possibility for any other application to block the operations from another application. To prevent

blocking for the duration of the operation, the API for these operations is transaction oriented. Each transaction is initiated with an "ATBeginX" function, where "X" stands for the operation being started such as Enroll, AcquireImage, Verify, Validate, Identify.

An application can receive events during a transaction by one of two methods: the application can receive messages synchronously by polling for new event messages or it can receive messages asynchronously by registering a callback function in the "ATBeginX" transaction function [30]. If the latter method is used the callback function is invoked whenever a new event message is available.

Focus:

When applications share the same sensor, at any given time only one application can receive images from the sensor. A loss of focus message ends the current transaction. All transaction message handlers should check for an AT_API_LOST_SENSOR_FOCUS message. The application that lost the sensor focus should not attempt to begin a new transaction until the window it is displaying gets the focus from Windows.

Transaction Timeout:

A timeout message message type AT_API_TIMEOUT is received by an application, , during a transaction if the AWFS is expecting to detect a finger on the sensor and no finger is detected after an extended period of time. A timeout message ends the current transaction. All transaction message handlers should check for an AT_API_TIMEOUT message.

Transaction Event Messages:

The following event messages can be sent to the transaction event handler during an open transaction. The messages along with what they specify are given below.

Information Messages:

• **AT_API_NEW_DISPLAY_IMAGE** - A new image is available. This is used to update the real-time image display window.

• **AT_API_FINGER_DETECTED** - A finger placement has been detected.

• **AT_API_FINGER_REMOVED** - A finger removal has been detected.

• **AT_API_DATABASE_CHANGE** - The opened shared database has been modified by another application. There is usually no action required of the application unless it is displaying database information, for example, a list of users.

• **AT_API_NO_CORE** - The current finger placement does not contain a core (the center of the fingerprint pattern). This is usually caused by a poor finger placement. A good finger placement has the core in the center of the sensor.

Prompt messages:

• **AT_API_LIFT_AND_REPLACE** - The current transaction requires the user to lift his finger from the sensor and then place the same finger on the sensor.

• **AT_API_PLACE_FINGER** - The current transaction requires the user to place a finger on the sensor.

Transaction data ready messages:

• **AT_API_ACQUIRE_DATA_RDY** - The current acquire image transaction has the final data ready. The application should call *ATEndAcquireImage()* to receive the transaction results.

• **AT_API_ENROLL_DATA_RDY** - The current enroll transaction has the final data ready. The application should call *ATEndEnroll()* to receive the transaction results.

• **AT_API_VALIDATE_DATA_RDY** - The current validation transaction has the final data ready. The application should call *ATEndValidateID()* to receive the transaction results.

• **AT_API_VERIFY_DATA_RDY** - The current verification transaction has the final data ready. The application should call *ATEndVerify()* to receive the transaction results.

• **AT_API_IDENTIFY_DATA_RDY** - The current transaction has the final data ready. The application should call *ATEndIdentify()* to receive the transaction results.

Transaction termination messages:

• **AT_API_TIMEOUT** – The AWFS cancelled the current transaction due to a timeout. The AWFS was unable to obtain a good image.

• **AT_API_LOST_SENSOR_FOCUS** – Another application sharing the same sensor has initiated a transaction. The current transaction for this application is terminated [30].

Cancelling a Transaction:

An application can cancel a transaction by calling *ATAbortTransaction()*.

Building an AWFS System:

Building an AWFS system require the include files and the libraries as given below:

Header Files The required header files are as follows:

GenTypes.h - Type definitions.

ACAPIDef.h - Message defines, structure definitions, error codes and enumeration value **ATStdAPITypes.h** – AT structure definitions.

ATInterface.h - API functions header file

Link Files An application should link to dynamic library *ATSC51.lib* and load the *ATSC51.dll* at runtime

Initialization An application must first initialize the AWFS system before calling any other API functions. In C, *ATInit()* is called to initialize the system. See the following code sample:

```
if ( AT_OK != ATInit())
  return -1;

if ( AT_OK != ATOpenSensor(NULL,
AT_SENSOR_OPEN_MODE_SHARED) )
{
MessageBox(NULL, "Failed to open a Fingerprint Sensor..\nExiting...",
"System Error!", MB_OK);
return -1;
}
// Close the sensor
ATCloseSensor() ;
// Close the system
```

ATClose();

API Functions

Initialization

ATInit()

As mentioned above ATInit() function Initializes the AuthenTec system API. An application calls this function during initialization. Calling this function requires a corresponding call to *ATClose()* prior to application shutdown.

AT_RESULT_CODE ATInit()
Parameters
None
Returns
AT_OK Initialization successful.

ATCreate()

This function initializes the AuthenTec system API. An application calls this function during initialization. Calling this function requires a corresponding call to *ATClose()* prior to application shutdown.

AT_RESULT_CODE ATCreate()
Parameters
None
Returns
AT_OK Initialization successful.

ATClose()

This function Closes the AuthenTec sub-system that was initialized previously by calling the *ATInit()* or *ATCreate()* function.

AT_RESULT_CODE ATClose()
Parameters
None
Returns
AT_OK Termination successful.

Convenience API:

There are Transaction Begin/End functions such as ATBeginEnroll, ATEndEnroll, ATBeginValidate, ATEndValidate. The AWFS API refers to these functions in ATSC51.lib and ATAuthenticateLib.lib. A user application can be developed using these functions but there is no user interface support included in this API. The AWFS includes additional "convenience" support called convenience API to rapidly develop an AWFS application. This support includes user interfaces for performing the various biometric operations such as enrollment, identification, etc using functions such as ATEnroll, ATIdentify. These functions in the convenience API make use of the transaction Begin/End functions required to do the operations. The source code and header files for the convenience functions can be compiled and included into the application which invokes a desired biometric operation by calling a single high-level convenience function. Convenience functions display user interface items, such as text prompts and fingerprint images necessary to perform the requested biometric operation. The convenience source code makes calls into the AWFS API, and obtains feedback from the main AWFS API while carrying out the biometric operation. An application is blocked while a convenience function is in process. Only the final result of the operation is returned by the convenience function. An application developer can alter the appearance or behavior of the user interface by modifying the supplied source code. The various biometric operations, and the high-level API calls for the Convenience API are described below.

High-Level Convenience Functions

ATConvenienceAPIInit() Initializes the Convenience API components.

ATEnroll() Performs an enrollment of a finger. The resulting reference template can be placed into the AWFS proprietary database or exported from the function upon successful completion. Various windows and message boxes will automatically guide the user through the enrollment process.

ATValidateFingers() Determines which, if any, external fingerprint template passed into the function matches the finger being placed on the sensor. Various windows and message boxes will automatically guide the user through the process.

ATValidateID() Determines whether a template for the specified user ID stored in the AWFS database matches the finger being placed on the sensor. Various windows and message boxes will automatically guide the user through the process.

ATIdentify()

Determines which, if any, template stored in the AWFS database matches the finger being placed on the sensor. Various windows and message boxes will automatically guide the user through the process.

Chapter 5

IMPLEMENTATION

Implementation is one of the important aspects when considering a security application. The Authentication system implemented in this thesis uses three factor authentication to give access to many applications with out the need for the user to enter his or her user id and password for each of the applications. Any application can use this component to provide three factor authentication. There might be applications that already use password to authenticate the users. These applications can further augment their security by using this three factor authentication system.

To use this system the user has to go through two phases Enroll phase and Authenticate phase. During the Enroll phase, the user is enrolled to the Authentication system using Java Ring, PIN and fingerprint. During enroll process a reference template of the user fingerprint is created. The user initially has to choose a user id and give his information such as first name, last name, phone no. etc. which are stored in the database. This database in this system is used to keep a record of the users using the authentication system and the user related information.

During the Authenticate phase, the user gives his Java Ring, pin and the fingerprint. The system checks to see if the ring belongs to that particular user, then the system checks if the pin is valid and then the system matches the users fingerprint with the reference template obtained during the enroll process. Only if all the three factors are validated, the user is authenticated. Once the user is authenticated , he is given access to the application.

Biometrics has proven to provide good authentication. But when using biometrics like a fingerprint, the question that arises is as to where to store the fingerprint. Fingerprint is vulnerable and if the fingerprint template is not stored in a secure place, it is possible that the fingerprint template can be tampered.

When considering a basic application providing logical access to PC/Network logon using biometrics, storing the fingerprint at the local computer or server might be enough as the fingerprint represent the digital identity of the person in the local environment and not on the internet. If the scope is PKI based applications (such as VPN, secure email etc) where a smart card is used for credential storage, the fingerprint template should be stored in the smart card.

It is important to choose the appropriate level of security for a system. There are different ways to implement biometrics. Two important aspects of biometric systems are

- Storing (on a server, in the PC, in a smart card)
- Matching (on a server, in the PC, in a smart card)

The card in our case is the Java Card in the Java Ring, which is a type of smart card. Depending on how these parts are combined, the security implications of the system are different.

The table below shows combinations of where a fingerprint can be stored, and where it can be matched. Some of these combinations are highly unlikely to ever exist in a commercial product and are therefore not discussed and marked with an X.

	Store on server	**Store on PC**	**Store on smart card**
Match on server	**a**	**X**	**b**
Match on PC	**X**	**c**	**d**
Match on smart card	**X**	**X**	**e**

Figure 4.4: Biometric data Stored/Matched [31]

(a) Match on server / Store on server

In this case, during the enroll process a reference template is created and stored on the server in a database. During the verification process, the user's fingerprint template (or match template) is sent to the server and the user's template is matched with the reference template at the server and a result is sent back to the user.

Matching on a server means matching the template in a protected environment. Using this system, the administrator can monitor the security and detect attempted attacks on the system. Hence the administrator has full control of the

fingerprint database. The storage on the servers means that also the template is protected from tampering, at least from the outside.

The drawbacks of using this system are that, it violates personal integrity. Getting users to store their fingerprint templates in a server out of their control may be hard; this requires that the party running the server is trusted. One security problem is the transfer of the template from the capturing device to the server. This requires a secure internet session or an intelligent way to solve the problem with cryptography. This solution also requires that a new infrastructure is built, which makes the solution difficult to deploy in large scale.

(b) Match on server / store on card

In this case, the reference template obtained during enrollment, remains with the user on a smart card. During the verification process, the user's fingerprint is sent to the server along with the reference template in the smart card. A matching of the user's fingerprint template with the reference template takes place at the server and a result of the matching is sent back to the user.

Using this kind of a system, the problem with storing ones fingerprint template on a server out of control is solved.

This solution has drawbacks both with regards to security and due to the fact that a new infrastructure has to be built. The problem with servers - the transfer of information across an untrusted network is augmented; now both the template and the input image must be transferred. In this case some kind of strong encryption should be applied to secure the transfer. This might require a new infrastructure to be built.

(c) Match on PC / Store on PC

This is a common combination where the templates are stored on the user's hard drive. This is also where the matching takes place. The advantage of using this system is that the user has got control of his/hers own templates.

Since the PC is not a secure device there is an immediate threat that secrets such as templates or passwords may be stolen or tampered with. Mobility may be a problem; the user can only log on to the computer where the template is stored. This solution is not even scalable on a local network.

(d) Match in PC / store on smart card

In this case, the reference template is stored on the smart card and during the verification process; the user's template is matched with the reference template on the PC.

This solution eliminates some of the problems with Match on PC/Store on PC. The advantage using this kind of a system is that the user can carry his or her own template. When a smart card is used it is often access to the protected area on the card that is critical. Access is granted if the correct PIN is sent to the card. The PIN is matched on the card. In this system, both the template and the PIN have to be transferred to the PC from the card, if the input image matches the template the PIN is sent back to the smart card to gain access. The template is not available for hacking at all time since it is stored on a card. The user can use the fingerprint and the smart card for accessing multiple devices.

The drawback of using this kind of a system is that the templates are exposed during verification process. The critical information (the template and the secret e.g. PIN) is sent to the PC from the card when matching. This means that both the template and the secret can be tampered with or stolen. This solution cannot be used for secure network transactions.

(e) Match on card / Store on card

In this case the reference template is stored on the card and during the verification process, the user's template is sent to the card and a matching takes place with the reference template and a result is sent back to the user. Using this kind of a solution the sensitive data (the template) never leaves the card. There is also no secret to steal since a successful match enables the use of certificates on the card without the need of stored PINs or passwords. Even in the unlikely event that a card is tampered with; only limited damage is done since only that specific users' credentials are hacked. An attack on multiple users means that the attacker must get hold of all users' cards. This method is normally seen as the most secure way of biometrically securing computers, networks and digital information in general.

The advantages of using this kind of a system are as follow:

- The smart card is made personal; it cannot be accessed without the appropriate biometric authentication
- The templates are never exposed to a non-tamper proof environment
- The user carries his/hers own templates
- The solution works with a PKI (digital signatures, authentication over networks, encryption) without the need of new infrastructure.

From the cases discussed, we know that security wise, match on card/ store on card is one of the best ways of implementing a system. The Java Ring used in this thesis implementation, does not support the fingerprint matching API. A product named Precise BioMatch provides the fingerprint matching API. Using this API

we can match the template inside the card (Java Ring). But the API precise BioMatch provides is Java Card 2.1.2 compatible where as the iButton in the Java Ring is 2.0 compatible. So, it is not possible to match the fingerprint template inside the Java Card. Due to this reason, we are going to store the reference template in the card and match the template with the user's fingerprint outside the card. So we are going to use the match on server/ store on card scenario discussed earlier.

Java Ring used in this project has 6K memory. It uses most of its memory for loading the applet in the card. Since the memory left after loading the applet is not sufficient to store a fingerprint template, we have segmented the fingerprint template into two. One of the segments is stored in the Java Ring and the other segment is stored on the database. Java Ring and iButton are used synonymously through out the discussion of the implementation.

The fingerprint API, AT API used in this project is in C and C++. To integrate the native methods in AT API with the other part of the system (which is implemented in Java), Java Native Interface (JNI) interface is used.

The two phases, the Enroll phase and the Authenticate phase as mentioned earlier are implemented as two systems, Enroll and Authenticate systems respectively. In the next section we will discuss about Enroll system and Authenticate system. For each of the system we discuss the high level design, system use case, collaboration diagrams and the features of each system as to how they are implemented. The implementation of the application is shown in UML notation

5.1. Enroll system:

Enrollment is a process where a user is initially enrolled into the system using, fingerprint, Java Ring and a pin. The figure below gives a high level design of the system.

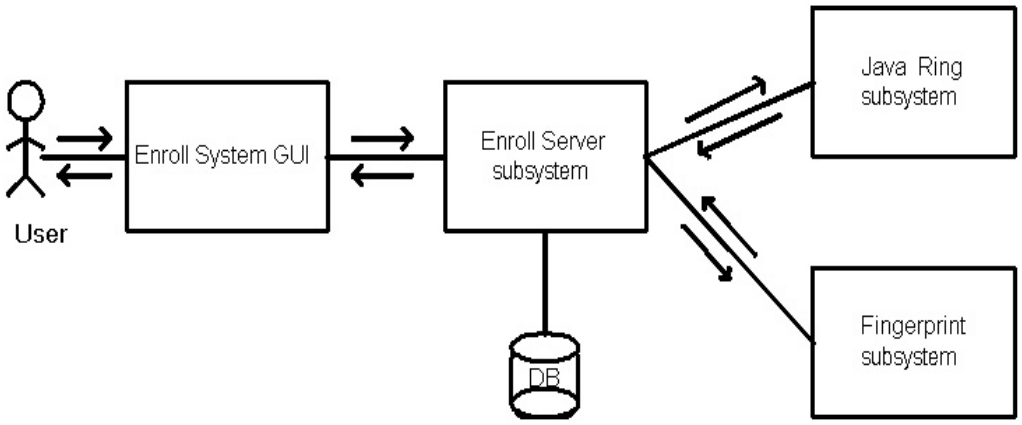

Figure 4.5: High level design of Enroll system

There are three subsystems involved in the enroll process, Java Ring subsystem, Fingerprint subsystem and Enroll server subsystem. The user sends an enroll request to the Enroll system GUI which is forwarded to the enroll server. The Enroll server sends requests to the Fingerprint subsystem to get the fingerprint template. The Fingerprint subsystem creates a fingerprint template for the user and sends the template as a response. The Enroll server next sends requests to store user information, the user application list, the user fingerprint template and the user pin to the Java Ring subsystem. The Java Ring subsystem stores the information of the user. The Enroll server sends requests to the database to store user Information, the user applications and the other part of the fingerprint template segment. The database stores all the required information of the user. Once all the information is stored, the iButton is locked. Locking the iButton is the last step in the enroll process. This is done my setting a flag.

5.1.1. Enroll User Use Case :

Use Case name: Enroll User

Summary: Customer is enrolled into the system to get access to application using a single sign on.

Actor: Authentication system customer

Precondition: System is idle with application Enroll and Authenticate option buttons on the screen. Administrator is available with the user for enrolling the user.

Description:

1. The customer clicks on the Enroll button.
2. The system displays a user information form with the fields for user name, phone number and address.
3. The user fills the form by entering his user name, phone number and address
4. If the user phone number is valid, the system prompts the user to choose a User ID.
5. If the User ID is not already enrolled for this user or any other user the system enrolls the user with the user id.
6. The system prompts the user to place his finger on the fingerprint sensor.
7. The user places his finger on the fingerprint sensor and follows the instructions for enrolling the fingerprint.

8. The system enrolls the user fingerprint.

9. The system prompts the user to insert his Java Ring in the blue dot receptor.

10. The user inserts his Java Ring in the blue dot receptor.

11. The system checks to see if the Java Ring is already enrolled.

12. If the Java Ring is not already enrolled, the system prompts the user to enter a User PIN (for the Java Ring).

13. The user enters a User PIN.

14. The system enrolls the Java Ring with the User PIN.

15. The system locks the Java Ring to Enrolled mode.

16. The system displays a message saying that the user is enrolled.

Alternatives:

- If the phone number is invalid and has less than or more than 10 digits, the system re-prompts the user to enter a valid phone number

- If the user chosen User ID is already enrolled for another user, the user prompts the user to choose a different user ID.

- If the user chosen User ID is already enrolled for this user, the system prompts if the user wants to re-enroll

- If the user selects the re-enroll option, the User ID is set as not enrolled and the user is enrolled again following steps from 1 to 16.

- If the system does not detect a fingerprint on the sensor for a certain amount of time, the system displays a system out of time message.

- If the system does not detect the user Java Ring in the blue dot receptor, the system displays a Java Ring not found error message.

- If the Java Ring is already enrolled, the system displays a message saying the Ring is already enrolled with an option to re-enroll.

- If the user selects a re-enroll option the system re-enrolls the Java Ring with the user chosen new User PIN

Postcondition: Customer has been enrolled.

5.1.2. Collaboration Diagram:

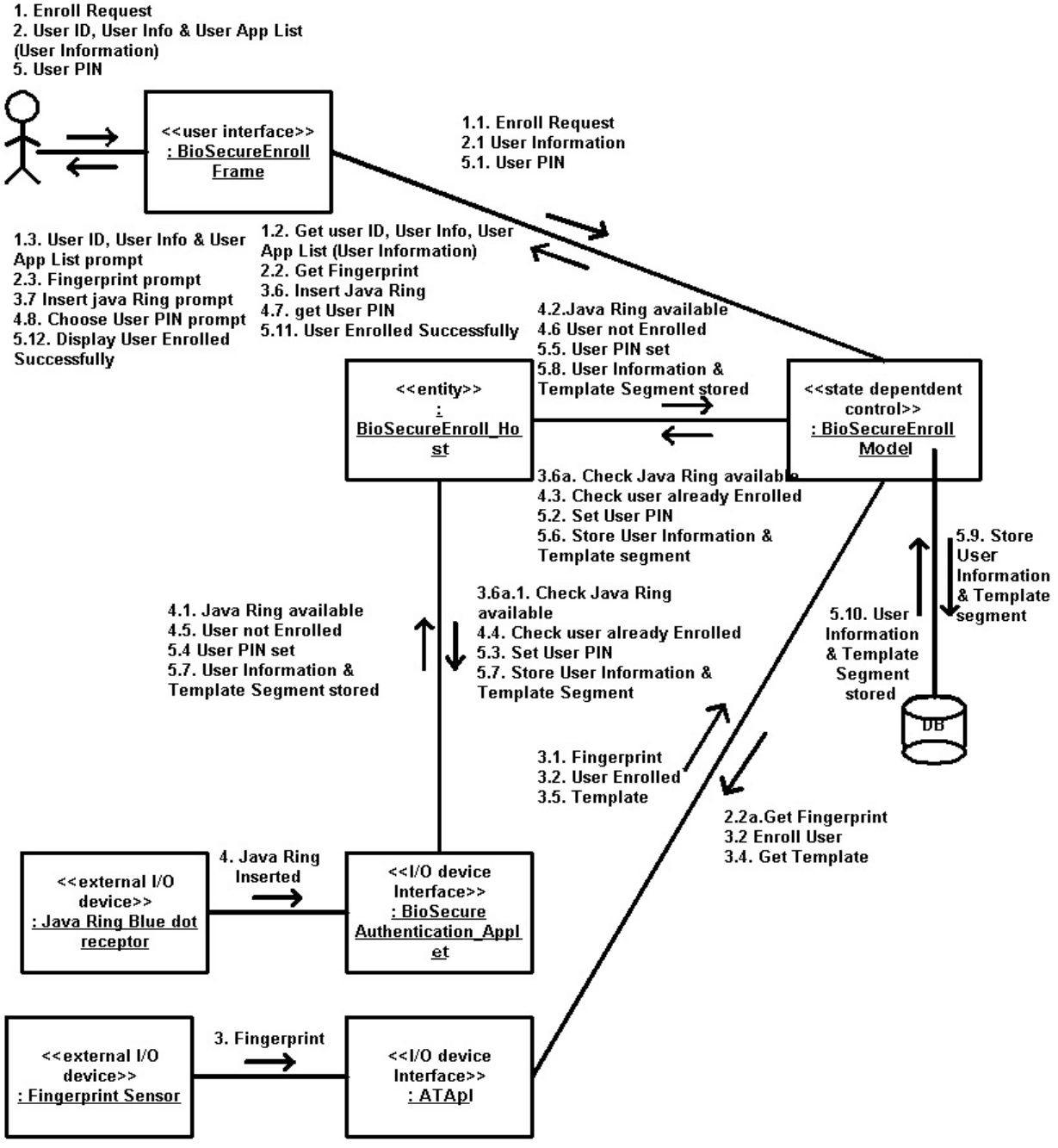

Figure 4.6: Collaboration diagram for Enroll User use case

5.1.3. Features of the Enroll system:

- The Enroll system enroll the user using the Fingerprint, Java Ring and a User PIN

- This system stores the application list that the user has to access to in the iButton.

- The Enroll system uses Obsever-Observable design pattern extending java.util.Observable to notify the user of the events about the Java Ring. The Observers of this class can be notified of the card inserted and card removed events and also notify of messages (like Exceptions etc on cardInserted actions). The Java Card has a listener called CTListener to observe for any events such as a Java ring inserted or Java Ring removed event. The Enroll application observes these events and notifies the model which further notifies the GUI which notifies the user about the update events. The host observes the applet using CTListener for any events and if any events occur, the host notifies the model which further notifies the GUI of the events and updates. This GUI displays an appropriate message to the user such as the iButton is inserted or iButton is removed.

- The Enroll system uses Model View Controller (MVC) design pattern. This design pattern clearly defines the boundaries between the user interface and business logic. Using this design pattern gives a good separation of modules and makes it clear and easy to understand. Model in this application deals with the data that need to be displayed and the operations that can be applied to transform the objects. The controller deals with updating a particular parameter in the model which has to be

displayed by the view. This is done in the action performed methods. The view deals with the GUI (presentation of the information to the user).

- The Enroll system provides, enroll and re-enroll features. A user can re-enroll with the same Java Ring again with out having to change his user id. The system uses the same user id to re-enroll the user.

- The system is self aware of the mode and presents the GUI accordingly. The system keeps track of the present mode and goes to the next mode according to action taken in this mode. For example, consider the case where the system is in 'insert' mode, where the user information is inserted into the database and Java Ring. The system finds that the ring is already enrolled. In such a case, the system asks the user if he wants to re-enroll. If the user chooses to re-enroll, the mode is automatically set to 'modify', where only the information that has been modified is updated in the database and the ring.

- The system uniquely handles exceptions and validations and displays the appropriate message to the user.

- The system locks the iButton as a last step in the enroll process. The user is considered as enrolled only if the iButton is locked. Locking the iButton, locks all the administrative functions after initial setup. iButton once locked cannot be accessed by the user thus preventing any changes to be made by the user.

- The application makes use of check pin function, which makes use of the Java Card Owner PIN API and iButton clock. This function checks to see if the PIN passed from the host (user PIN) matches the applet's

internal PIN. A host application has 5 tries to get the correct PIN. If an incorrect PIN is supplied 5 times in a row, the PIN is blocked and cannot be used again (even with the correct PIN value) for a duration of 30 minutes. After this time period has elapsed, the host may once again attempt to send a correct PIN.

- The application provides 3 attempts for entering a correct PIN. If the user does not enter a correct PIN in 3 attempts, he is not authenticated and hence not given access to the applications

- The system provides the additional feature of giving administrator access to the Ring after locking the iButton. The administrator can access the iButton using an administrator PIN. This can be useful in cases where a user happens to forget his or her user PIN. Another example is where a user wants to re-enroll or change his information in the card; he can do so with the help of the administrator who has the administrator key.

- The user fingerprint is uploaded into the applet in batches 128 bytes. The fingerprint segment cannot be sent in a single apdu due to constraints of Java One'98 release of iButton which limits the array index parameter to maximum number of bytes that can be sent. So, the fingerprint is sent iteratively in batches of 128 bytes.

5.2. Authenticate system: Once a user is enrolled, the user can access the applications he wants using the authentication process. The figure below gives a high level design of the authentication process.

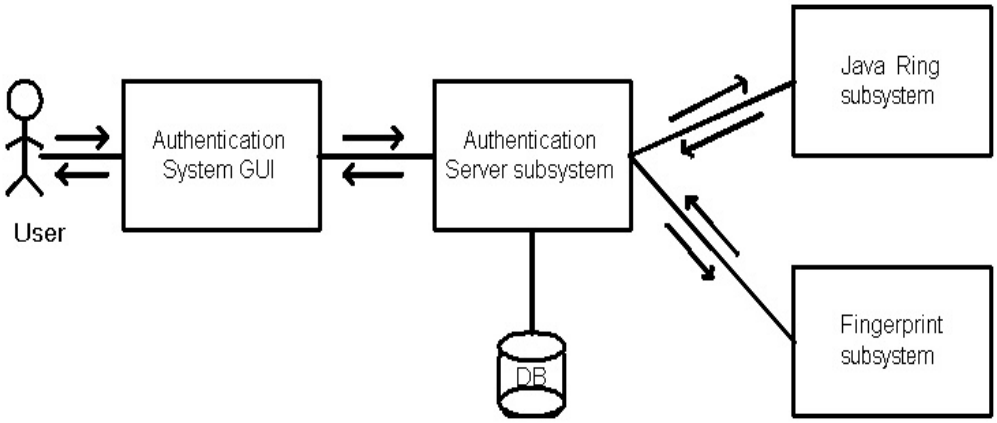

Figure 4.7: High level design of Authenticate system

There are three subsystems involved in the authentication process. They are Java Ring subsystem, Fingerprint subsystem and the Authentication server subsystem. The user sends a request for authentication to the Authentication system GUI which is forwarded to the Authentication server subsystem. The Authentication server subsystem first checks to see if the iButton is locked. Only if the ibutton is locked, the user is considered enrolled. If the ibutton is not locked, the user is considered as not enrolled and hence not authenticated and not given access tot eh application. During the authentication process, the Authentication server subsystems request the Java Ring subsystem for information such as user pin, user information, user applications and fingerprint templates, and the Java Ring sends the corresponding information. The fingerprint template segment obtained from the Java Ring is the first segment of the fingerprint template. The Authentication server subsystem gets the second segment of the fingerprint template from the database. The Authentication server subsystem then sends a

request to Fingerprint subsystem to validate the user template. The Fingerprint subsystem validates the user reference template against the user fingerprint and sends a response to the Authentication server subsystem. If the fingerprint is validated, the user is authenticated and given access to his applications.

5.2.1. Authenticate User Use Case:

Use Case name: Authenticate User

Summary: Customer is given access to the user applications using a single sign on.

Actor: Authentication system customer

Precondition: System is idle with application Enroll and Authenticate option buttons on the screen.

Description:

1. The customer clicks on the Authenticate button.
2. The system prompts the user to enter his User ID.
3. If the User ID is already enrolled, the system prompts the user to insert his Java Ring in the blue dot receptor.
4. The user inserts his Java Ring in the blue dot receptor.
5. The system checks to see if the Java Ring is already enrolled.
6. If the Java Ring is already enrolled, the system prompts the user to enter his User PIN.

7. The user enters his User PIN.

8. If the user entered User PIN is valid, the system prompts the user to place his finger on the fingerprint sensor.

9. The user places his fingerprint on the fingerprint sensor.

10. The System validates the fingerprint with the template.

11. If the fingerprint is validated, the system displays a User Authenticated message along with the list of applications accessible to the user.

12. The user selects the application he wants to access.

Alternatives:

- If the User ID is not already enrolled, the system displays an invalid User ID message.

- If the system does not detect the Java Ring in the blue dot receptor, the system displays a Java Ring not found error message.

- If the Java Ring is not already enrolled, the System displays a Java Ring not enrolled message

- If the system does not detect a fingerprint on the sensor for a certain amount of time, the system displays a system out of time message.

- If the user entered User PIN is invalid, the system re-prompt the user for a User PIN

- If the User PIN is invalid for three times, the system exits.

Postcondition: The user is authenticated using single sign on and given access to all of the user applications.

5.2.2. Collaboration Diagrams:

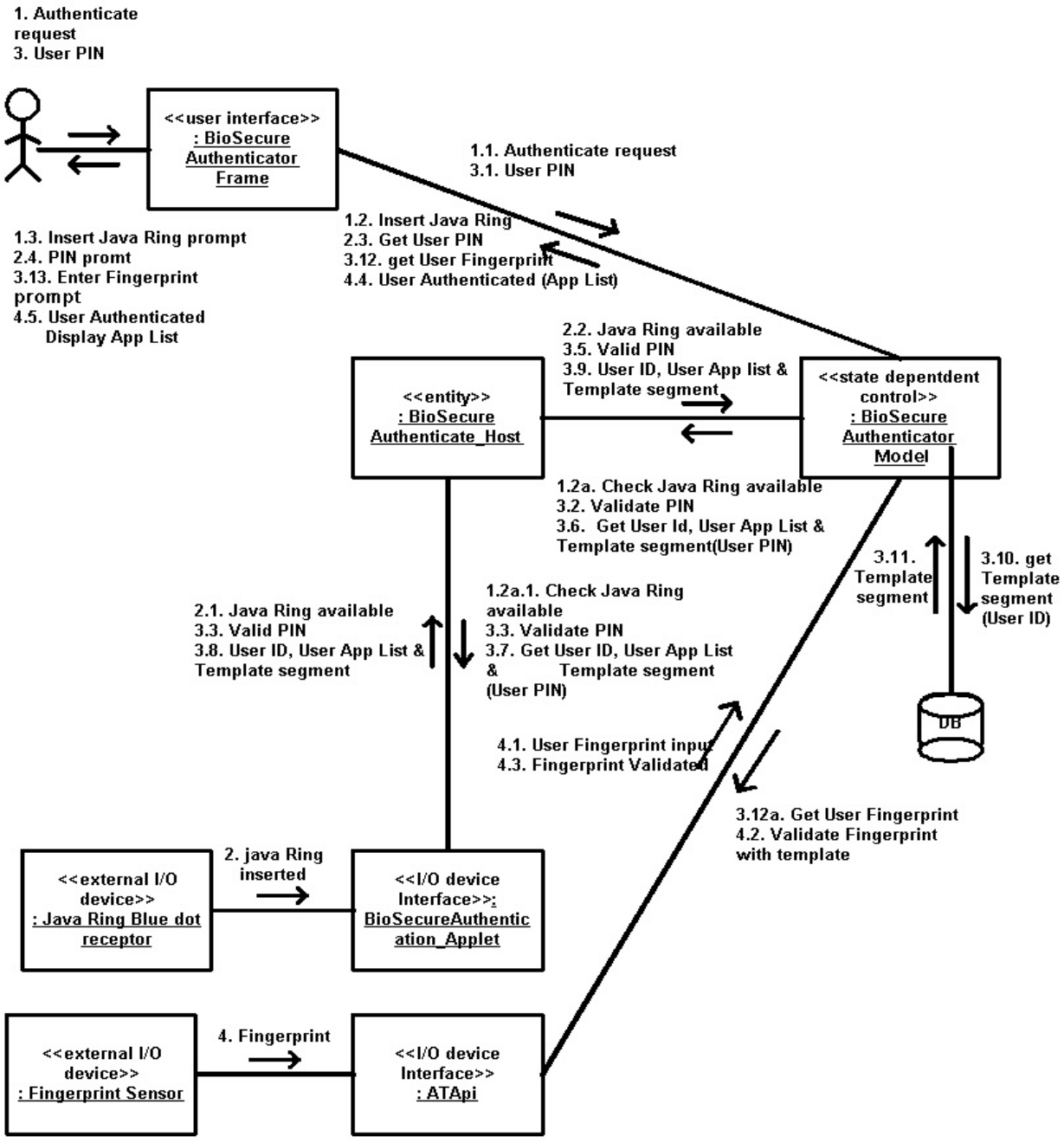

Figure 4.8: Collaboration diagram for Authenticate User use case

5.2.3. Features of Authenticate system:

- The Authenticate system stores the fingerprint template on the card and matches the template on the host.

- The Enroll system uses Obsever-Observable design pattern extending java.util.Observable to notify the user of the events about the Java Ring. The Observers of this class can be notified of the card inserted and card removed events and also notify of messages (like Exceptions etc on cardInserted actions). The Java Card has a listener called CTListener to observe for any events such as a Java ring inserted or Java Ring removed event. The Enroll application observes these events and notifies the model which further notifies the GUI which notifies the user about the update events. The host observes the applet using CTListener for any events and if any events occur, the host notifies the model which further notifies the GUI of the events and updates. This GUI displays an appropriate message to the user such as the iButton is inserted or iButton is removed.

- The Authenticate system uses Model View Controller (MVC) design pattern which clearly defines the boundaries between the user interface and business logic. Model in this application deals with the data that need to be displayed and the operations that can be applied to transform the objects. The controller deals with updating a particular parameter in the model which has to be displayed by the view. The view deals with the GUI (presentation of the information to the user).

- The Authentication system has a timeout feature. If the system does not detect a Java Ring or a fingerprint on the sensor with in a specified time,

the system exits with a time out message and the authentication of the user is failed.

- The system provides 3 attempts for entering a correct PIN. If the user does not enter a correct PIN in 3 attempts, he is not authenticated and hence not given access to the applications.

- The application makes use of check pin function, which makes use of the Java Card Owner PIN API and iButton clock. This function checks to see if the PIN passed from the host (user PIN) matches the applet's internal PIN. A host application has 5 tries to get the correct PIN. If an incorrect PIN is supplied 5 times in a row, the PIN is blocked and cannot be used again (even with the correct PIN value) for a duration of 30 minutes. After this time period has elapsed, the host may once again attempt to send a correct PIN.

- The Authentication system has a simplified usage and can be plugged into any application by instantiating an Authenticate object and calling Authenticate and user name as parameters. This feature makes it ideal to be used as an API. An application can use the Authenticate application by passing the user id.

- When an application calls Authenticate object, the application and the GUI for the Authenticate system run in two different threads. Calling an instance of Authenticate immediately starts GUI. Since application and the GUI run in two different threads, the thread for the calling application is stopped and the GUI thread is started. Once the Authentication system authenticates the user, the control is given to the application thread. The Authentication system returns the user id if the

user is authenticated. If the user is not authenticated, it throws an exception. The calling program gives access to the user, depending on what the Authenticate system returns. If the Authentication system returns user id, the user is authenticated and hence can access the application.

- The Authenticate system retrieves the fingerprint from the iButton in batches of 128 bytes.

- The system uniquely handles exceptions and validations and displays the appropriate message to the user.

5.3. System Initialization:

When the Authentication system is initialized, it performs the following functions:

- Sets the Database Connection

- Checks if fingerprint sensor is available

- Checks if the blue dot receptor is available

- Instantiates a Java Ring host

- Initializes the ATApi (fingerprint API).

If system fails to perform any one of the above initialization functions, it gives a fatal exception message and shuts down.

5.4. iButton Implementation:

The BioSecureAuthenticate_Applet is the applet in the iButton. There are two hosts, enroll host and authenticate host. In this application, enroll host is the BioSecureEnroll_Host and authenticate host is the BioSecureAuthenticate_Host. When an iButton is inserted, the enroll host and the authenticate host check to see if the applet is loaded in the iButton. The enroll host loads an applet, if an applet not already loaded in the iButton during the enroll process. The authenticate host gives an error message if an applet is not already loaded during the authentication process.

During the enroll process, the host communicates with the applet using a set of commands. The figure below shows the commands sent from the host to the applet during enroll process.

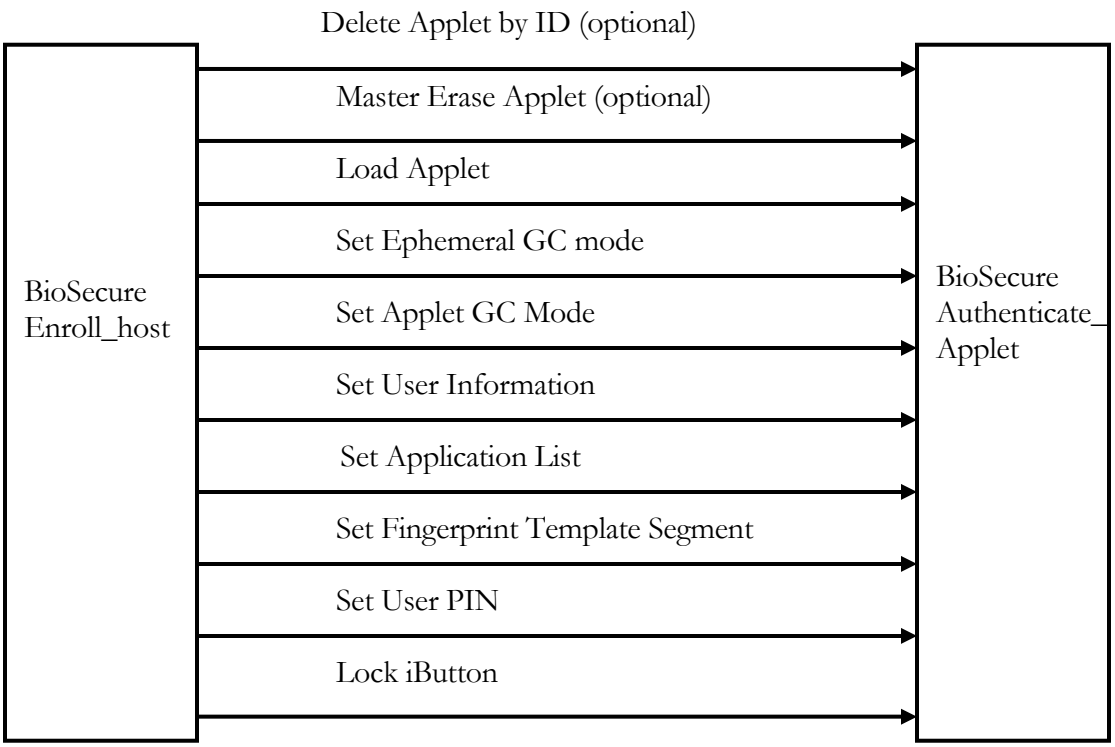

Figure 4.9: Communication between host and applet during Enroll process

If the host finds that an applet is already loaded in the iButton with the given user information, it asks the user if he or she wants to re-enroll. If the user accepts to re-enroll, BioSecureEnroll_host sends a **delete applet by ID** request to the BioSecureAuthenticate_Applet to delete the already existing applet in the iButton.

Master Erase Applet is an optional command and is sent only in cases where there are no other applets (used by other applications) co-existing with the BioSecureAuthenticate_Applet in the iButton. Master Erase Applet option erases all the applets installed on the iButton, frees all memory created by these applets,

and reset all configuration options back to default values. (i.e., set AppletGC to off, etc.).

Load applet command is sent to the applet when the host does not detect a user information applet in the iButton. The host sends the Load applet command after sending a Delete Applet by ID command or a Master Erase command to load an applet.

Setting the **Ephemeral GC mode**, the ephemeral collector in the applet recovers data that was referenced for a short period and been out of scope (local variables, objects whose references are never stored in reference fields, etc).

Setting the **Applet GC mode**, applet collector recovers data that was referenced by the fields of an applet. These references are instantiated and then the references are lost either by setting the field to null or by instantiating another block of data.

Set **User Information** sets the user given information - user id, user name, phone number and address in the applet.

Set **User Application Access List** sets the user application list along with their corresponding user name and passwords for each application in the applet.

Set **Fingerprint Template Segment** sets the fingerprint template segment for the user in the applet.

Set **User PIN** sets the user pin for the user in the iButton. The user can access the iButton using this user pin when the user wants to get access to his applications.

The last part of the enrolling processing is to **lock the iButton**. This is done after setting the user pin. Locking the iButton is to let the application know that the user has been enrolled. If the user quits before finishing the enroll process, in such a case, iButton is not locked and hence the application knows the user is not enrolled. The communication between the BioSecureAuthenticate_Host and BioSecureAuthenticate_Applet is as shown below.

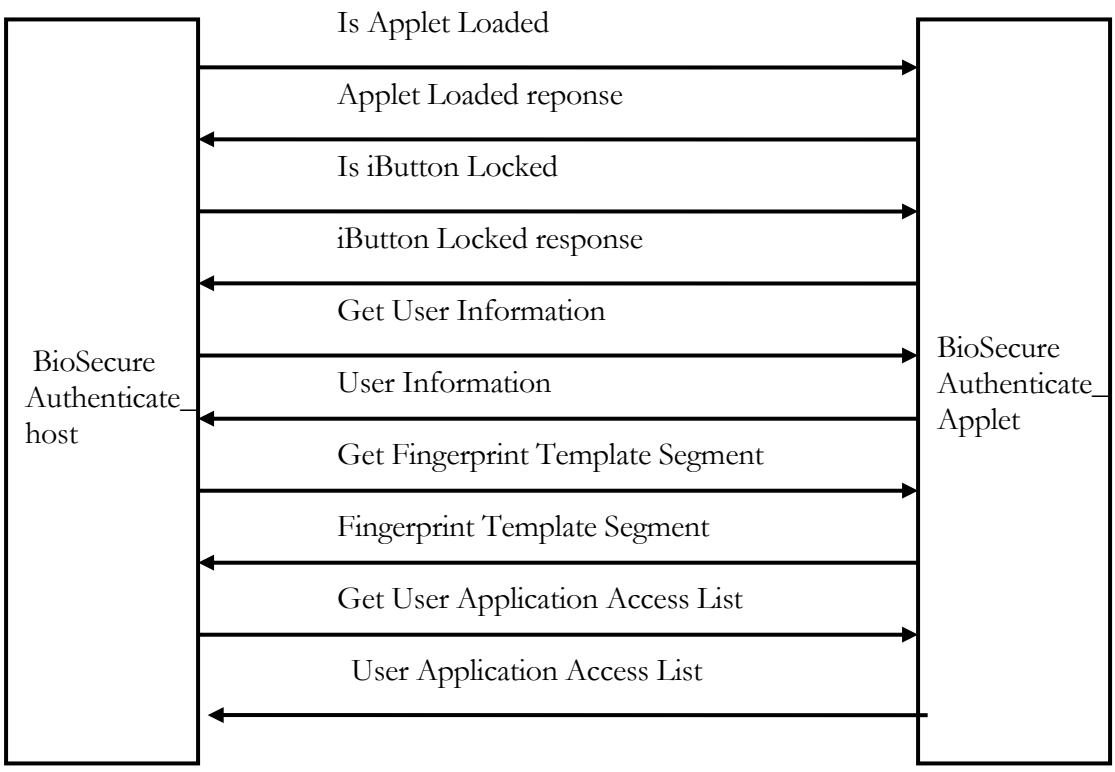

Figure 4.10: Communication between host and applet during the Authentication process

During the Authentication process, the applet checks if the applet is loaded using the **Is Applet Loaded** command. If the applet is already loaded, applet sends a response indicating that the applet is loaded. If the applet is not loaded, it sends a

response indicating that the applet is not loaded. In cases where the applet is not loaded, system throws an exception giving an error message stating that the user is not enrolled.

The applet next checks if the iButton is locked using **is iButton locked** command. If the host receives an iButton locked response, it proceeds to the other commands. If the host receives an iButton unlocked response, it sends an error message stating that the user is not enrolled.

Once the host makes sure that the applet is loaded and the iButton is locked, the host gets the user information from the applet using the **Get User Information** command. The user pin is send as a parameter to check if the user is the right user. The applet returns the user id, phone number and address as the response to this command.

Get Fingerprint Template Segment command is send with the user pin as a parameter to get the user template segment as a response from the applet.

Get User Application access List command is send with the user pin as a parameter to get the user application list as a response.

As mentioned in chapter 5, the host and the applet communicate using APDU commands and responses. A command APDU has CLA (the applet name), INS (method being called), p1 & p2 (additional control data), DATA (data to be sent) and Le (length of the data) as its parameters.

The applet sets the CLA of the applet as shown below

public static final byte BIOSECURITYENROLL_CLA = (byte)0x80;

Each method in the applet is given a number as shown below to be represented as 1 byte.

```
  // Main Functionality
  public static final byte BIOSECURITYENROLL_INS_SETUSERINFO =
(byte)0;
  public static final byte BIOSECURITYENROLL_INS_GETUSERINFO =
(byte)1;
……..
……..
```

The install method in the applet is overridden by calling the constructor as follows.

```
public static void install(APDU apdu)
  {
    new BioSecureAuthenticate_Applet();
  }
```

The first thing the process method does is to check whether CLA in the command APDU received matches the CLA of the applet.

```
    //Determine if the applet is being selected.
    if((buffer[ISO.OFFSET_CLA] == SELECT_CLA) &&
      (buffer[ISO.OFFSET_INS] == SELECT_INS))
    {
      //**********************************
      //* Add any code to be executed on *
      //* applet selection here.         *
      //**********************************

      return;
     }
```

The process method would perform the appropriate instruction by doing a switch on the INS field of the apdu, calling the appropriate method.

```
switch (buffer[ISO.OFFSET_INS])
        {
           case BIOSECURITYENROLL_INS_SETUSERINFO:
                  setUserInfo_Dispatch(apdu, buffer[ISO.OFFSET_P1],
                  buffer[ISO.OFFSET_P2]);
                  break;

           case BIOSECURITYENROLL_INS_GETUSERINFO:
                  getUserInfo_Dispatch(apdu, buffer[ISO.OFFSET_P1],
                  buffer[ISO.OFFSET_P2]);
                  break;

                  …..

           default:
                   ISOException.throwIt(ISO.SW_CLA_NOT_SUPPORTED);
        }
```

In the host application the sendAPDU method

sendAPDU(int CLA, int INS, int P1, int P2, byte[] Data, int Le) is overridden with the

sendAPDU(int CLA, int INS, int P1, int P2, byte[] Data, SlotChannel sc) method. The SlotChannel sc here is the connection through which the apdu is be sent.

All the methods in the host application use the sendAPDU method with slot channel as one of its parameters. After executing all the methods, the host runs the finalize method which closes all the open slot channels.

The next part of this section discusses about the CLA, INS, p1 & p2, Data and Le values for each of the methods in the BioSecureEnroll_Host and BioSecureAuthenticate_Host.

BioSecureEnroll_Host methods:

CLA value is the same for all methods since we are sending the apdu to the same applet. The length Le is different for each method and is equal to the length of the data. For any method, the data is always appended with the length of the user, user pin and the length of the actual data to be sent.

CLA: 0x80

INS: (byte)0 (BIOSECURITYENROLL_INS_SETUSERINFO)
p1: 0x00 & p2: 0x00
Data: byte array with user first name, last name and phone number.
p1: 0x01 & p2: 0x00
Data: byte array with user address.

INS: (byte)2 (BIOSECURITYENROLL_INS_SETFINGERPRINTSEGMENT)
p1: 0x00 & p2: 0x00
Data: Since the whole template segment cannot be send in a single apdu, the template segment is divided and send in blocks of 128 byte, which is the APDU packet length. When p1 and p2 have a value of 0x0, it indicates that the block being sent is the first block of the template segment.
p1: 0x01 & p2: 0x00
Data: when p1 has a value of 0x01, it indicates that the block being sent is a second, third, last or any other block. It indicates that it is not a first block.

INS: (byte)7 (BIOSECURITYENROLL_INS_SETUSERPIN)
p1: 0x00 & p2: 0x00
Data: user pin as a byte array.

INS: (byte)5 (BIOSECURITYENROLL_INS_SETAPPLIST)

p1: 0x00 & p2: 0x00
Data: user application list as a byte array.

INS: (byte)9 (BIOSECURITYENROLL_INS_LOCKBUTTON)
p1: 0x00 & p2: 0x00
Data: user pin that has to be locked as a byte array.

INS: (byte)1 (BIOSECURITYENROLL_INS_GETUSERID)
p1: 0x00 & p2: 0x00
Data: user pin as a byte array.

The response apdu for this method would have the user ID as its Data and the status words.

BioSecureAuthenticate_Host methods:

CLA: 0x80
INS: (byte)1 (BIOSECURITYENROLL_INS_GETUSERINFO)
p1: 0x00 & p2: 0x00
Data: user pin as a byte array. If p1 and p2 are 0x00, it indicates that the applet must send the user first name, last name and phone number as a response.
Response APDU:
Data: user first name, last name and phone number
p1: 0x01 & p2: 0x00
Data: user pin as a byte array. If p1 has a value of 0x01 and p2 0x00, it indicates that the applet must send the user address as a response.
Response APDU:
Data: user address

INS: (byte)3 (BIOSECURITYENROLL_INS_GETFINGERPRINTTEMPLATESIZE)
p1: 0x00 & p2: 0x00
Data: When p1 and p2 have a value of 0x0, it indicates that the applet must send the first block of the fingerprint segment.
Response APDU:
Data: first block of the template segment
p1: 0x01 & p2: 0x00
Data: when p1 has a value of 0x01, and p2 0x00, it indicates the applet to send the next fingerprint segment block (which is not the first block). The data also has the start index and end index of the

segment that is sent as the block. Using these two indices, a block of the fingerprint is made and sent.

Response APDU:

Data: It has the second, third, last or any other block other than the first block of the template segment

INS: (byte)1 (BIOSECURITYENROLL_INS_GETAPPLIST)

p1: 0x00 & p2: 0x00

Data: user pin as a byte array

Response APDU:

Data: the user application list

All the response apdu's have a status word along with the reponse data to indicate the status of the operation. For example a status word of 0x9000 indicates a successful execution with out errors.

5.5. Fingerprint Implementation:

The application first initializes the fingerprint API during the initialization of the enroll process. When fingerprint API is initialized, the fingerprint database is initialized. AT API is used to initialize, enroll, identify and validate a user. During an enroll process, a fingerprint template is created for a user. This template is passed to the application controller which then segments the template into two and stores one of the segments in the Java Ring and the other in the database. During the authentication process, the application controller combines the two template segments and passes the complete template to the AT API for validation. The AT API then obtains a user fingerprint and matches it against the application fingerprint template. If a match is found, the user is validated, if not the user is not validated. If a user is not validated, he or she is not authenticated and hence not given access to the applications. If a user is validate he or she is given access to the applications.

The AT API for the fingerprint is in C and C++. The native methods are called from the Java implementation part of the application by creating dynamic libraries (dll).

The following are the native functions used by the Java implementation part of the application.

N_ATInitialize(String pDatabaseFileName, int pHwnd)

This method calls the native function which initializes the fingerprint database. The database is usually stored as a file. This file name along with a window handle is sent. The window handle is used to display the fingerprint while enrolling, identifying or validating the user. The native function in C calls the function ATInit() to initialize the database as shown below.

```
lDatabaseFilename = (TCHAR*)pEnv->GetStringUTFChars(pDatabaseFileName,
                                                    &isCopy);
// Initialize the AT control DLL.
ATInit();
// Open up a database for this application to use
iFLResult = ATOpenDatabase( lDatabaseFilename,
                            AT_DATABASE_ACCESS_SHARE,
                            100000, NULL, 0);
return iFLResult;
```

N_ATIdentify()

This method calls the native function to identify a fingerprint template which returns the user id (String) as a return value. Identification is process where, the application matches the user fingerprint against the enrolled fingerprint templates in the database. This native function calls a function ATIdentify() with user id as its parameter as shown below.

```
iFLResult = ATIdentify( (PTCHAR)lUserId );
return pEnv->NewStringUTF(lUserId);
```

N_ATEnroll(String pUserId)

This method calls the native function to enroll a user. User id is given as an input parameter to the native method and an integer representing the result code is returned. This integer can be decoded and an appropriate message is given to the user. The native function calls the function ATEnroll(). The native function first allocates some memory for the fingerprint template before calling the ATEnroll() as shown below.

```
lUserId = (PTCHAR)pEnv->GetStringUTFChars(pUserId, &isCopy);
if ((iFLResult = ATEnroll(NULL, 0, NULL, 0, &iEnrollMaxStructSize )) ==
    AT_OK)
{
    iFLResult = ATEnroll( lUserId, 0,  m_pTemplateStorage,
                          iEnrollMaxStructSize, &iEnrollResultStructSize );
    if ( iFLResult != AT_OK )
    {
        m_pTemplateStorage = NULL;
    }
}
pEnv->ReleaseStringUTFChars(pUserId, (char*)lUserId);
return iFLResult;
```

getATTemplate(String pUserId)

This method calls the native function to get a template for a particular user id. The template is returned as a byte array. The native function gets a pointer to the enrolled template and returns the template as shown below.

```
PTCHAR lUserId;
jboolean isCopy;
AT_RESULT_CODE iFLResult;
jbyteArray jb;
lUserId = (PTCHAR)pEnv->GetStringUTFChars(pUserId, &isCopy);
jb = pEnv->NewByteArray(iEnrollResultStructSize);
pEnv->SetByteArrayRegion(jb, 0, iEnrollResultStructSize, (jbyte*)
m_pTemplateStorage);
pEnv->ReleaseStringUTFChars(pUserId, (char*)lUserId);
return (jb);
```

N_ATValidate(String pUserId)

This method calls the native function which validates a fingerprint against the templates available in the database file. The native function calls the function ATValidateID() , which takes in a user id as an input as shown below.

```
lUserId = (PTCHAR)pEnv->GetStringUTFChars(pUserId, &isCopy);
iFLResult = ATValidateID( lUserId );
pEnv->ReleaseStringUTFChars(pUserId, (char*)lUserId);
return iFLResult;
```

N_ATValidateFinger(byte[] pTemplate)

This method calls the native function to validate a fingerprint against another specific template passed to it as an input byte array. The function returns the result code which is an integer. The user fingerprint is obtained from the user from the window handle. It uses the ATValidateFingers function to validate the fingerprint against the fingerprint template as shown below.

```
lTemplate = pEnv->GetByteArrayElements(pTemplate, 0);
if ( pTemplate == NULL ) {
    iFLResult = AT_BAD_POINTER;
} else {
```

```
    iFLResult = ATValidateFingers( (void *)lTemplate );
}
pEnv->ReleaseByteArrayElements(pTemplate, lTemplate, 0);
return iFLResult;
```

Chapter 6

CONCLUSION

The Authenticate system uses three-factor authentication to authenticate a user. This makes the system more secure than one factor or two factor authentications. For this thesis, we developed the three-factor authentication system with Java Ring, Biometrics and a pin. We were able to demonstrate its usage by securing some applications including the Enroll application. Using the Authentication system for the Enroll application, only the authenticated users are given access to the Enroll application. This Authentication system has many security features.

It uses Java Ring with an iButton which is Java Card 2.0 compliant. Java Card is a kind of smart card. Smart cards have always been considered very secure way of storing information. Java Ring with the iButton can overcome the deficiencies of the secret passwords. In order to gain access to the iButton, the user has to know the pin. The iButton's zeroization capability erases the fingerprint template than reveal it to anyone.

The National Institute of Standards (NIST) and the Communications security Establishment (CSE) have validated a version of the crypto iButton for protection of sensitive, unclassified information.

The fingerprint sensor used in this application uses TruePrint technology. Using this technology makes it difficult for an imposter to fool the sensor with techniques such as gummy finger, thus making it less vulnerable. One of the most important issues of storing the fingerprint is solved in this application. The fingerprint template is stored in the Java Ring. The advantages of using this system are that the user can carry his or her own template (stored in the smart

card) and the user might use the fingerprint/smart card for accessing multiple devices. It allows individuals to control the access themselves - thus rendering individuals no longer impotent to the vulnerability of computers, databases and software or to accidents, malfunction or intrusion.

With this three factor authentication, even if a hacker gets the pin, he cannot gain access to the application as he has to go through the process of fingerprint validation.

The iButton is locked once enrolled, not allowing any one to see or change the information in the Java Ring. If the Ring is stolen, the hacker cannot access it because he has to unlock the iButton to access any of its resources. Unlocking an iButton can be done only by the administrator.

The application uses check pin function, one of the features of iButton. This function gives the additional security of blocking the iButton for a certain period of time. If an incorrect PIN is supplied 5 times in a row, the PIN is blocked and cannot be used again, even with the correct PIN value for duration of 30 minutes.

The application provides 3 attempts for entering a correct PIN. If the user does not enter a correct PIN in 3 attempts, he is not authenticated and hence not given access to the applications.

In this application, the fingerprint template is segmented into two and each of the segments is stored in a different place. This makes it more difficult for the hacker to get the complete template because he has to get the template from different places.

The Authentication system can be integrated in many ways. These are discussed in future work.

Three factor authentication is one of the good ways of authenticating a user. Three factor authentication can be done using other physical devices and biometrics. The biometrics may include any thing such as scanning the iris, face recognition etc. Using three factor authentication and implementing a system by considering all the security issues makes a system more secure.

Chapter 7

FUTURE WORK

The Authentication system can be integrated further in many ways. Given below are some of the future works that can be done on the system.

Hand Shaking:

It is good to have a handshaking between the host and the applet before the host gives any commands to the applet. In the handshaking process, host application (or terminal) must authenticate the applet before sending any messages to it and the applet also must authenticate the host.

Encrypting the communication:

If the communication between the host and the applet is encrypted, it can prevent the hackers from hacking any information between the host and the applet.

iButton as a Single Sign-On (SSO) resource with authorization:

The iButton can be made as a SSO resource by saving the applications that the user can access along with the applet pin and application access List in the iButton. The user name and password for each application is saved in the iButton and once the user is authenticated using three factor authentication, he or she is automatically logged in to the application.

The application authorization features of the user can also be stored in the iButton. These authorization features specify the user's actual roles in the

application such as employee, department head etc. The application downloads the authorization information and gives access to the user according to the access privilege set for the role

Store the whole Fingerprint Template:

In the Authentication system, the complete template cannot be stored in the iButton due to memory limitations. There are iButton available with more memory. Using an iButton with more memory allows storage of the whole fingerprint template in the iButton.

Store usage statistics:

iButton can be further used to store usage information statistics such as the time when a particular application was accessed, last time the database was accessed using iButton, the time when a particular transaction took place etc for securing financial transactions, point-of-sale transactions. Storing this kind of information can be useful for the administrator as well as the user. A user can keep a record of his or her transactions and activities. It can help the administrator to keep track of the user's actions.

Store application Level information:

iButton can be used to store application level information which can be application related information such as encryption key or decryption key to gain access to an internet application or a database resource.

Match on Card:

The iButton used in the Authentication system is Java Card 2.0 compliant. A product named precise BioMatch, provides API to match a fingerprint inside the

Java Card. But the BioMatch API is Java Card 2.1.2 compliant. This is one of the reasons why iButton in this thesis could not match the fingerprint on the card. Using an iButton which is Java Card 2.1.2, we can match the fingerprint in the card using the precise BioMatch API thus upgrading the system further to Match on Card/Store on Card system, which is considered the most secure way of implementing a three factor authentication system with fingerprints as the biometrics.

REFERENCES

[1] Gollmann, Stallings; Computers and Security; July 2001; http://www.iwar.org.uk/comsec/resources/security-lecture/show50b7.html

[2] Jess Garms and Daniel Somerfield, Professional Java Security, Wrox Press Ltd., 2001

[3] Nari Kannan; How to catch some next big things and lose others; March 2004; http://blogs.ittoolbox.com/bi/entrepreneur/archives/000574.asp

[4] Fortress Technologies; Fortress Technology Unveils Three-Factor Authentication for Wireless Security; http://www.80211bnews.com/publications/page207-495001.asp

[5] Dekart Logon; Secure Logon for windows; http://www.dekart.com/products/authentication_access/logon/

[6] Trio Security Inc.; A new standard in Authentication security; http://www.findbiometrics.com/Pages/feature%20articles/trio.html

[7] Richardson Business Machines; Two & Three Factor Authentication; http://www.richardsonbus.com/products/2factor.html

[8] Rainbow Technologies Inc.; Two-Factor Authentication – Making sense of all options; February 2002; http://www.itsecurity.com/papers/rainbow2.htm

[9] Stephen M.Curry, An Introduction to Java Ring; 1998; http://www.javaworld.com/javaworld/jw-04-1998/jw-04-javadev_p.html

[10] iButton; Java-Powered Cryptographic iButton; http://www.ibutton.com/ibuttons/java.html

[11] Search Web Services; Java Ring; March 2004; http://searchwebservices.techtarget.com/sDefinition/0,,sid26_gci836660,00.html

[12] Dallas Semiconductor Maxim, Frequently Asked Questions; http://db.maxim-ic.com/ibutton/faq/index.cfm?fuseAction=FAQ.subCategories&CategoryID=5&categoryName=iButtons#What%20is%20an%20iButton?

[13] OpenCard; Open Card Framework: Frequently asked questions; http://www.opencard.org/misc/OCF-FAQ.shtml#JavaCard

[14] Howstuffworks; What is a smart card; http://electronics.howstuffworks.com/question332.htm

[15] Rinaldo Di Giorgio; Smart cards: A Primer; 1997; http://www.javaworld.com/javaworld/jw-12-1997/jw-12-javadev.html

[16] Rinaldo Di Giorgio; Smart cards and OpenCard Framework; 1998; http://www.javaworld.com/javaworld/jw-01-1998/jw-01-javadev.html

[17] Zhiqun Chen; Understanding Java Card 2.0; 1998; http://www.javaworld.com/javaworld/jw-03-1998/jw-03-javadev.html

[18] Arsalan Lodhi; A Java Card Primer; http://www.developer.com/java/other/article.php/910261
[19] Thomas Schaeck with Rinaldo Di Giorgio; How to write OpenCard services for Java Card Applets; 1998; http://www.javaworld.com/javaworld/jw-10-1998/jw-10-javadev.html
[20] Dallas Semiconductor, maxim; ftp://ftp.dalsemi.com/pub/iB-IDE_2.0
[22] Sun Microsystems Inc.; Java Card Platform Security, Technical white paper; http://java.sun.com/products/javacard/JavaCardSecurityWhitePaper.pdf
[23] International Biometric Group; http://www.biometricgroup.com/access_control.html
[24] Anil K. Jain; Fingerprint Matching; 2002; http://www.pims.math.ca/industrial/2002/mitacs-agm/jain/
[25] Jay Lyman; New Technology spots Fingerprint ploys; June 2002; http://www.newsfactor.com/perl/story/18029.html
[26] Aron Ligon; An Investigation into the Vulnerability of the Siemens ID Mouse Professional Version 4; September 2002; http://www.bromba.com/knowhow/idm4vul.htm
[27] Tsutomu Matsumoto; Impact of Artificial "Gummy" Fingers on Fingerprint Systems; http://www.totse.com/en/bad_ideas/locks_and_security/164704.html
[28] AuthenTec Inc.; Why Fingerprint Authetication; http://www.authentec.com/finalInteg/WhyFingerprints.htm
[29] AuthneTec Inc.; Why TruePrint Technology; http://www.authentec.com/finalInteg/WhyTruePrint.htm
[30] AuthenTec, Inc.; AuthenTec Windows Fingerprint Software Version 6.3 for Microsoft Windows, Programmer's Reference Manual.
[31] Magnus Pettersson, Marten Obrink.; How secure is your biometric solution?, 20th Febuary 2002

VITA

Jyothi Chitiprolu earned her Bachelor of Science, degree from the University of Madras, India, in 2001. She majored in Computer Science. She pursued a Masters of Science degree in Computer Science to gain more experience in her fields of interest. Her areas of interest include Computer security, Client server web application and distributed databases.