

IMPLEMENTATION OF ECDSA BASED ON JAVA CARDS

A DISSERTATION

*Submitted in partial fulfilment of the
requirements for the award of the degree*

of

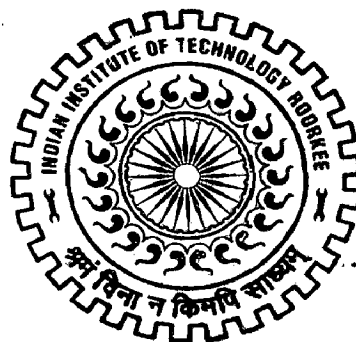
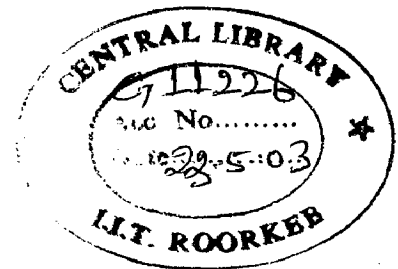
MASTER OF TECHNOLOGY

in

INFORMATION TECHNOLOGY

By

S. SURESH KUMAR



**IIT Roorkee-ER&DCI, Noida
C-56/1, "Anusandhan Bhawan"
Sector 62, Noida-201 307**

FEBRUARY, 2003

Enrollment No - 019040

621-380285-
SUR

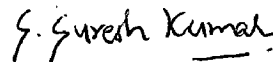
CANDIDATE'S DECLARATION

I hereby declare that the work presented in this dissertation titled "**IMPLEMENTATION OF ECDSA BASED ON JAVA CARDS**", in partial fulfillment of the requirements for the award of the degree of **Master of Technology in Information Technology**, submitted in **IIT, Roorkee – ER&DCI Campus, Noida**, is an authentic record of my own work carried out during the period from August, 2002 to February, 2003 under the guidance of **Dr. P.R. Gupta**, Reader, Electronics Research and Development Centre of India, Noida.

The matter embodied in this dissertation has not been submitted by me for award of any other degree or diploma

Date: 24-2-2003

Place: Noida



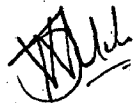
(S Suresh Kumar)

CERTIFICATE

This is to certify that the above statement made by the candidate is correct to the best of my knowledge and belief.

Date: 24.2.2003

Place: Noida



(Dr. P.R. Gupta)

Reader

ER&DCI, Noida

ACKNOWLEDGEMENT

I hereby take the privilege to express my deepest sense of gratitude to **Prof. Prem Vrat**, Director, Indian Institute of Technology, Roorkee, and **Mr. R.K. Verma**, Executive Director, Electronics Research & Development Center of India, Noida for providing me with this valuable opportunity to carry out this work. I am also very grateful to **Prof. A.K. Awasthi**, Programme Director and Dean, Post Graduate Studies and Research, **Prof. R.P. Agarwal**, course coordinator, IIT, Roorkee and **Mr. V.N. Shukla**, course coordinator, ER&DCI, Noida for providing the best of the facilities for the completion of this work and constant encouragement towards the goal.

I am grateful to my guide **Dr. Poonam Rani Gupta**, Reader, ER&DCI, Noida for her valuable guidance, advice, suggestions and constant encouragement through numerous discussions and demonstrations.

My sincere thanks to **Mr. Munish Kumar**, Project Engineer, ER&DCI, Noida for his valuable suggestions.

I thank Mr. Joseph Smith, member, JavaCard forum, jguru.com who has helped me whenever I found any difficulties. I owe special thanks to my best friends, all of my classmates and other friends who have helped me formulate my ideas and have been a constant support. Thanks to my parents and my brother who provided their support and enduring confidence in me during my entire life. Last but not the least, I thank almighty for being on my side from the conception of this idea to its implementation.



(S. Suresh Kumar)

Enrollment No. 019040

CONTENTS

Candidate's Declaration	(i)
Acknowledgement	(ii)
Abstract	1
1. Introduction	3
1.1 Overview	3
1.2 Objective	5
1.3 Scope	5
1.4 Organization of dissertation	5
2. Literature Survey	7
2.1 Digital Signatures	7
2.1.1 Classification of Digital Signatures	8
2.1.2 Characteristics of Digital Signatures	8
2.1.3 Applications of Digital Signatures	8
2.2 Smart Cards	9
2.2.1 Types of Smart cards	9
2.2.2 Java Smart Cards	10
2.2.3 Necessity of cryptography on smart cards	14
2.2.4 Limitations of available smart cards	14
2.2.5 Applications of smart cards	15
2.2.6 Java Card Vs Standard Java	16
2.2.7 Benefits of Java card technology	17
2.2.8 Java card security	18
2.2.9 Applications of Java cards	20
2.3 Elliptic Curve Cryptography	20
2.3.1 On the mathematics of fields	20
2.3.2 On the mathematics of elliptic curves	22

3. Analysis & Design	25
3.1 ECDSA (Elliptic Curve Digital Signature Algorithm)	25
3.2 Secure Hash Algorithm	27
3.3 Java Card applet	30
4. Implementation	33
4.1 Implementation of the algorithm	33
4.2 Running on Java Card	35
5. Results and Discussions	39
5.1 Running on Java Development Kit	39
5.2 Running on Java Card Kit	43
6. Conclusion	47
References	49
Appendix A – Java Card Kit	51

ABSTRACT

The security and portability of Java cards provide a safe, reliable, convenient, and effective way to ensure secure e-business and to enable a broad range of new applications. Java cards represents one of the smallest computing platforms in use today. A major factor influencing the design and implementation of Java Card is the limited availability of computing resources. Thus, the class ECDSA (Elliptic Curve Digital Signature Algorithm) is implemented, acted on the Java Card platform. ECDSA, the elliptic curve analogue of the more widely used DSA (Digital Signature Algorithm), is a robust digital signature algorithm. Compared to traditional public-key cryptographic algorithms it offers advantages such as shorter key lengths, faster execution, which are particularly important for implementation of smart card applications.

everywhere a Java smart card is found. It defines its own API and virtual machine. Also, the Java Card API insists that any applet be cryptographically signed by the card issuer. This ensures rogue applets are not accepted by a card thus reducing the risk of hackers breaking into the card.

The runtime environment encapsulates the underlying complexity and details of the smart card system. Applications request system services and resources through a well-defined high-level programming interface.

Therefore, Java Card technology essentially defines a platform on which application written in the Java programming language can run in smart cards and memory constrained devices. In addition to providing Java language support, Java Card technology defines a runtime environment that supports the smart card memory, communication, security, and application execution model.

Also to use smart card as secure authentication tokens, the basic cryptographic services must be available in the card environment. The digital signature capability is important on a smart card because the users not only access services but also authorize others by signing authorization certificates. This card-computer can be programmed to perform tasks and store information, but note that the brain is little -- meaning that the smart card's power falls far short of a desktop computer. So, to enable the digital signature capability, the selection of the algorithm plays an important role. As Java cards have small amount of memory, selection of an algorithm among all the existing digital signature algorithms is mainly based on the space complexity. That's why, ECDSA (elliptic curve digital signature algorithm) is implemented as its shorter key lengths fits to the limited environment of the cards. As smart cards are not very fast, the shorter key lengths resulting in faster execution also favours to use this algorithm when compared to other digital signature algorithms.

1.2 Objective

To understand Java card and elliptic curve cryptography technologies and to implement ECDSA using Java language and to run it in Java card environment.

1.3 Scope

This dissertation presents a study of Java card and elliptic curve cryptography technologies. Java smart cards have been discussed in detail and how to program on a Java card environment is also described. Here first step was to implement the ECDSA on PC (Personal Computer) environment where one can create a document, generate public and private keys based on the desired key size, sign the document and send across to anyone by enclosing in a file, which contains the public key, message and the signature. One can read the document sent by the signer and can verify the signature for authentication of the sender. In second step the algorithm was tested using Java card runtime environment on PC only in a simulated environment due to non-availability of Java Smart Card and CAD (Card acceptance device)

1.4 Organization of dissertation

This dissertation report contains six chapters. Chapter 1 presents the overview and discussed the objective and scope of dissertation. Chapter 2 deals with the fundamentals of smart cards, Java card environment and elliptic curve cryptography. Chapter 3 discusses about the ECDSA and how a communication takes place through APDUs (Application Program Data Units). Chapter 4 deals with the actual implementation of the algorithm and the steps used for developing Java card applet. Chapter 5 shows the user interfaces and discussions on them. Chapter 6 concludes the thesis, along with future enhancement aspects.

LITERATURE SURVEY

2.1 Digital Signatures

A digital signature is an authentication mechanism which enables the creator of the message to attach a code that acts as a signature similar to the way we sign a piece of information using our handwritten signature [3]. This signing is done by operating on the data with some key in such a way that the result is a digital signature. The signature needs to be such that only one person can create it (the signer) using a private key, but everyone else should be able to verify it using a public key. Figure 2.1 illustrates this property. This is exactly the opposite situation than in public key encryption, where we want everyone to be able to encrypt a message, but only the private key holder should be able to read it.

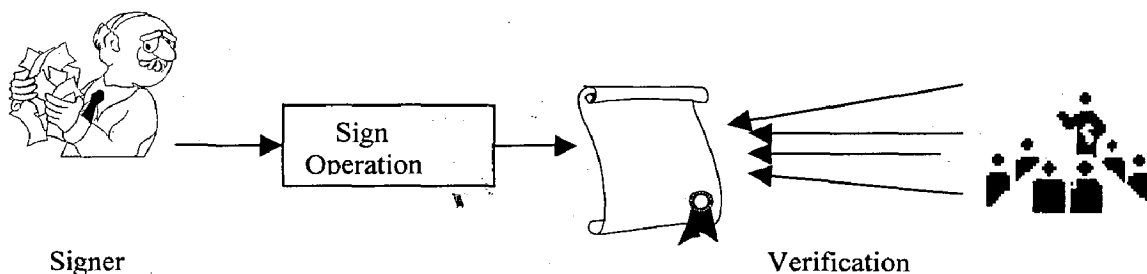


Fig 2.1: The digital signature created by the signer can be checked by everybody with a public key.

There are many digital signature algorithms with varying levels of security and Complexity: DSA (Digital Signature Algorithm), RSA (Revert Shamir Adelman) and ECDSA to name a few.

2.1.1 Classification of Digital Signatures

The digital signature schemes in use today can be classified according to the hard underlying mathematical problem which provides the basis for their security:

1. Integer Factorization (IF) schemes, which base their security on the intractability of the integer factorization problem. Examples of these include the RSA and Rabin signature schemes.
2. Discrete Logarithm (DL) schemes, which base their security on the intractability of the (ordinary) discrete logarithm problem in a finite field. Examples of these include the ElGamal, Schnorr, DSA and Nyberg-Rueppel signature schemes.
3. Elliptic Curve (EC) schemes, which base their security on the intractability of the elliptic curve discrete logarithm problem.

2.1.2 Characteristics of Digital Signature

1. The signature is authentic. The signature convinces the document's recipient that the signer deliberately signed the document.
2. The signature is unforgeable. The signature is proof that the signer, and no one else, signed the document.
3. The signature is not reusable. The signature is part of the document; an unscrupulous person cannot move the signature to a different document.
4. The signed document is unalterable. After the document is signed, it cannot be altered.
5. The signature cannot be repudiated. The signer cannot claim that he or she didn't sign it [3].

2.1.3 Applications of Digital Signatures

Digital signatures can be used to provide the following basic cryptographic services:

1. Data integrity - the assurance that data has not been altered by unauthorized or unknown means.

2. Data origin authentication - the assurance that the source of data is as claimed.
3. Non-repudiation - the assurance that an entity cannot deny previous actions or commitments.
4. Digital signature schemes are commonly used as primitives in cryptographic protocols that provide other services including entity authentication (e.g., FIPS 196 (Federal Information Processing Systems 196), ISO/IEC 9798-3(International Standards Organization/International Electrotechnical Commission) and Blake-Wilson and Menezes), authenticated key transport (e.g., Blake-Wilson and Menezes, ANSI X9.63 (American National Standards Institute) and ISO/IEC 11770-3), and authenticated key agreement (e.g., ISO/IEC 11770-3 , Diffie, van Oorschot and Wiener, and Bellare, Canetti and Krawczyk).
5. They are also intended for use in electronic mail, electronic funds transfer, electronic data interchange, software distribution, data storage, and other applications, which require data integrity assurance and data origin authentication.

2.2 Smart Cards

The smart card is a credit card sized plastic card with its own processor and memory embedded with an integrated circuit chip. One can think of the smart card as a "credit card" with a "brain" on it, the brain being a small-embedded computer chip. Card-computer can be programmed to perform tasks and store information, but note that the brain is little -- meaning that the smart card's power falls far short of desktop computer.

2.2.1 Types of smart cards

There are two types of smart card. The first is really a "dumb" card in that it only contains memory. These cards are used to store information. Examples of this might include stored value cards where the memory stores a dollar value which the user can spend in a variety of transactions. Examples might be pay phone, retail, or vending machines.

The second type of card is a true "smart" card where a microprocessor is embedded in the card along with memory. Now the card actually has the ability to make decisions about the data stored on the card. The card is not dependent on the unit it is plugged into, to make the application work. A smart purse or multi-use card is possible with this technology.

As there is a microprocessor on the card, various methods can be used to prevent access to the information on the card to provide a secure environment. This security has been touted as the main reason that smart cards will replace other card technologies.

The microprocessor type smart card comes in two flavors - the contact version and the contactless version. Both types of card have the microprocessor embedded in the card however the contactless version does not have the gold plated contacts visible on the card. The contactless card uses a technology to pass data between the card and the reader without any physical contact being made. The advantage to this contactless system is there are no contacts to wear out, no chance of an electric shock coming through the contacts and destroying the integrated circuit, and the knowledge that the components are completely embedded in the plastic with no external connections. The disadvantage to this is that there are some limitations to the use of the smart card.

2.2.2 Java smart cards

Specifically Java smart card, which is alternative to conventional smart card, is meant for running under Java programs where as conventional smart card runs under programs written in other languages. Smart card applications written in other languages cannot work in any environment and cannot interoperate [4]. So there is a need of a language by which the programs written in that language can easily fit into any type of scenario i.e., can run in any sought of environment and can easily interoperate. Java Virtual Machine fulfills such a requirement where a program written in Java can run anywhere. So this, "write once, run anywhere" capability of Java offers a solution to this problem. Also Java is a secure language. It ensures applets from different sources work and play well with each other on a smart card. Data considered private by one applet will

not be accessed by another. All these features make Java smart card advantageous when compared to conventional smart card.

Smart cards can be distinguished between high-end and low-end [4].

- Low-end smart cards only have memory inside; like magnetic stripes, they are simply storage devices. Phone card is one of the low-end smart cards.
- High-end smart cards contain microprocessor and memory. Not only it can store data, but also perform calculations. Electronic commerce and GSM (Global System for Mobile communication) phone use high-end smart cards.

By definition, Java smart card is grouped into high-end smart cards.

The major components inside Java smart card are microprocessor and memories. The basic architecture of Java smart card consists of Applets, Java Card API, Java Card Virtual Machine, and Operating System & Native Function, all included inside memories as shown in fig 2.2

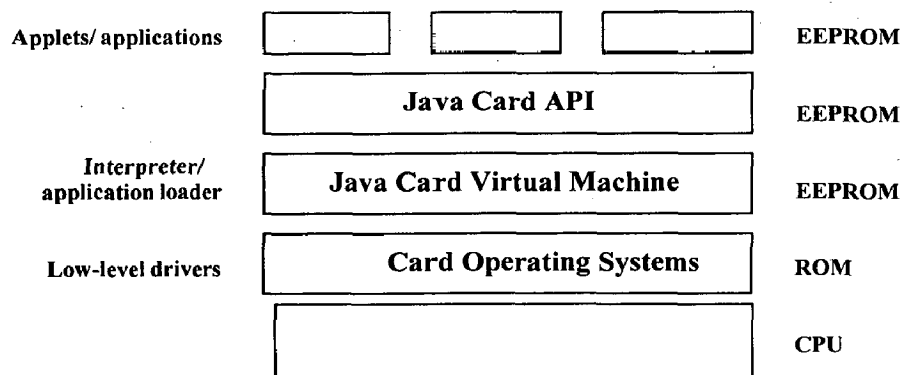


Fig 2.2: The basic architecture of a Java Smart Card

- **Applets/ applications** – It is defined as smart card application written in Java Programming language and conforming to a set of conventions so that it can run within Java Card runtime environment (JCRE). Many applets can fit inside one single Java smart card with each applet being identified by AID (Applet Identifier). The

communication between an applet and a host application is achieved through exchanging Application Program Data Units (APDUs).

- **Java Card API** – The API class library supports the Card bytecode in applets that allows inter-application communication.
- **JavaCard Virtual Machine** – Knowing that there is a limited supply of memory in Java smart card (24Kb), the size of the data has to reduce to minimum in order to fit into the card. As the result, Java smart card is designed to adopt the subset of Java language specification and JavaCard Virtual Machine is used to convert the data into this subset format which takes up less space and also optimize the performance when executing in JavaCard VM. Java provides a separate virtual machine for each of its technologies. Java Card virtual machine (JCVM) is the tiniest of all the Java virtual machines.
- **Operating System & Native function** – It deals with basic cryptography, I/O (Input/Output), memory access and application load services.
- **Microprocessor** – Calculations inside Java smart card will be done by microprocessor. Today, 8-bit microprocessor is still commonly used.
- **Memory** – Three types of memories used in the Java smart card are tabulated below:

TYPE	SIZE	USAGE
ROM	16Kb	Storing OS and native function
EEPROM	8Kb	Storing applets, JavaCard VM, JavaCard API
RAM	256 bytes	Used as a buffer for storing transmission data

Table 2.1 Types of memories used in Java smart card

There are eight contacts in Java smart cards. They are power supply, reset, clock, ground, input/output and three optional contacts. The subtle point about the physical characteristics and the 8 contacts layout of the Java smart card is that it is compatible to

ISO 7816 (part 1 & 2) – International Standardisation Organisation. This allows Java smart card interoperates with the other smart cards and also enable applications from different industries coexist in the same card. For example, it can combine the personal identification data for GSM phone with the credit card [4].

ISO 7816 consists of:

- Part 1: Physical characteristics.
- Part 2: Dimensions and location of the contacts.
- Part 3: Electronic signals and Transmission protocols.
- Part 4: Inter-industry commands for interchange.
- Part 5: Application identifiers.
- Part 6: Inter-industry data elements.
- Part 7: Inter-industry commands for SCQL (Structured Card Query Language).

Reader

So one may ask – where does the power come from? There is no power supply inside Java smart card, all the power has to be rely on its partner – Card Acceptance Device (CAD), or sometimes known as Interface Device (IFD), terminal or reader. Reader supplies power to Java smart card via contact (card is inserted to the reader) or contactless (using antenna) communication. Apart from being a power supply, reader’s main role is actually to establish data carrying connection and link up the relationship between Java smart card and the computer system [4]. The relationship is depicted in the figure 2.3 as shown below.

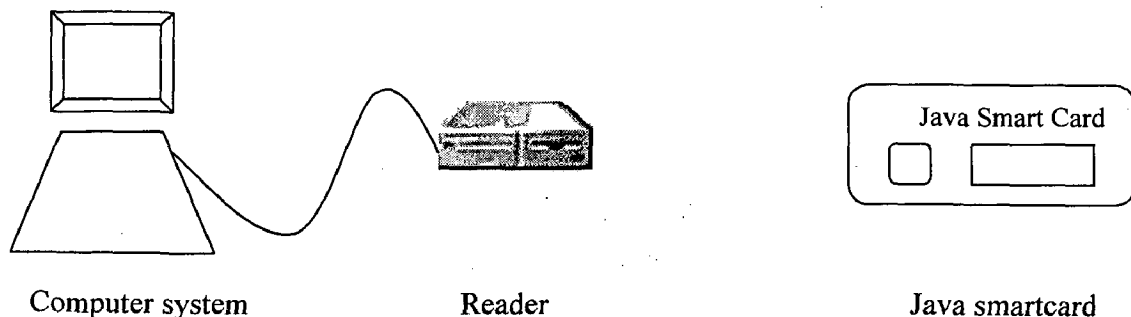


Fig 2.3 Communication between Computer system and Java smart card via Reader

2.2.3 Necessity of cryptography on smart cards

Currently, smart cards are used in telephone, transportation, banking, and healthcare transactions etc., All of these applications require sensitive data to be stored in the card, such as cryptographic keys for authentication, etc., To use smart cards as secure authentication tokens, the basic cryptographic services must be available in the card environment. Today many of those services are available, but mostly through specialized hardware integrated on the cards. The digital signature capability is important, because the user can not only access services, but also authorize others by signing authorization certificates of his/her own. The ability to sign certificates also means that it is possible to further delegate rights in the certificates issued by someone else.

Java Card is the specification of a Java programming environment for smart cards. Its introduction has made a 32-bit software environment, which is similar to the desktop programming environment, available for smart card developers. The most recent cards in the market even promise a natively 32-bit processor, which is important for the performance of the Java Card environment. At the time of writing the memory capacities of the cards are also increasing: best commercially offered cards are beginning to approach the 32KB boundary. The fitting of Java to a card does have its problems, however. For example, the memory management has to be done manually, as a garbage collector is not included in the specification. Generally, the introduction of high-level languages for smart cards makes the software production for them more appealing. For that reason, it seems likely that in the future more and more software functionality will be implemented for the cards.

2.2.4 Limitations of available smart cards

The worst shortcomings are in terms of memory on the card. Currently available cards have either 16KB or 32KB of EEPROM (Electrically Erasable and Programmable Read Only Memory), which is quite a small amount, compared to current desktop development platforms. Usually card applications are developed using a PC with at least four orders of magnitude higher capacity in terms of memory.

Another limitation is that most current cards have 8-bit processors, which adds to the memory problem since Java has been designed with 32-bit processors in mind. This means that bit manipulation has to be done before the application can be run on the card. This in turn means more memory usage and slower execution time.

Available smart cards also have very limited processing powers. Current smart cards typically have a clock frequency of about 4 MHz [5]. Because of poor performance, the amount of data processed cannot be very large, if the running time is to be reasonable.

Additionally smart cards are slow in their I/O [5]. As the user only sees the total response time, the slow I/O places additional burden to process the data fast. It also implies that the amount of data transferred between the cards and the terminal should be kept to a minimum.

2.2.5 Applications of Smart cards

Smart cards currently are used in telephone, transportation, banking, and healthcare transactions, and soon -- thanks to developers like you -- we'll begin to see them used in Internet applications. Smart cards are already being used extensively in Japan and Europe and are gaining popularity in the U.S. In fact, three significant events have occurred recently in the smart card industry in this country:

PC/SC

Microsoft and several other companies introduced *PC/SC* (Personal computer/Smart Card) a smart card application interface for communicating with smart cards from Win32-based platforms for personal computers. PC/SC does not currently support non-Win32-based systems and may never do so. We will discuss this in greater detail later on.

OpenCard Framework

OpenCard is an open standard that provides inter-operability of smart card applications across desktops, laptops, set tops, and so on. OpenCard promises to provide 100% pure Java smart card applications. Smart card applications often are

not pure because they communicate with an external device and/or use libraries on the client. (As a side note, 100% pure applications could exist without OpenCard, but without it, developers would be using home-grown interfaces to smart cards.) OpenCard also provides developers with an interface to PC/SC for use of existing devices on Win32 platforms.

JavaCard

JavaCard was introduced by Schlumberger and is submitted as a standard by JavaSoft recently [6]. Schlumberger has the only Java card on the market currently, and the company is the first JavaCard licensee. A smart card with the potential to set the overall smart card standard, JavaCard is comprised of standard classes and APIs that let Java applets run directly on a standard ISO 7816 compliant card. JavaCards enable secure and chip-independent execution of different applications.

2.2.6 Java Card vs. Standard Java

Java, as it exists on the card, has many limitations that make the development rather different from that practiced in other Java environments. Main differences and limitations are the following:

Supported Java Features	Unsupported Java Features
<ul style="list-style-type: none"> ◆ Small primitive data types: boolean, byte, short. ◆ One-dimensional arrays. ◆ Java packages, classes, interfaces and exceptions. ◆ Java object oriented features: inheritance, overloading and dynamic object creation, access scope and binding rules. ◆ The int keyword and 32-bit integer data type support are optional. 	<ul style="list-style-type: none"> ◆ Large primitive data types: long, double, float. ◆ Characters and strings. ◆ Multidimensional arrays. ◆ Dynamic class loading. ◆ Security manager. ◆ Garbage collection and finalization. ◆ Threads ◆ Object serialization ◆ Object cloning

Table 2.2: Supported and unsupported Java features

2.2.7 Benefits of Java card technology

There are several unique benefits of the Java Card technology, such as [7]:

Platform Independent - Java Card technology applets that comply with the Java Card API specification will run on cards developed using the JCAE (Java Card Application Environment) - allowing developers to use the same Java Card technology-based applet to run on different vendors' cards.

Multi-Application Capable - Multiple applications can run on a single card. In the Java programming language, the inherent design around small, downloadable code elements makes it easy to securely run multiple applications on a single card.

Post-Issuance of Applications - The installation of applications, after the card has been issued, provides card issuers with the ability to dynamically respond to their customer's changing needs. For example, if a customer decides to change the frequent flyer program associated with the card, the card issuer can make this change, without having to issue a new card.

Flexible - The Object-Oriented methodology of the Java Card technology provides flexibility in programming smart cards.

Compatible with Existing Smart Card Standards - The Java Card API is compatible with formal international standards, such as, ISO7816, and industry-specific standards, such as, Europay/Master Card/Visa (EMV).

2.2.8 Java Card Security

Java Card security has not yet been analyzed deeply in the literature, but some information is available. Many of the changes are done in order to fit Java to the limited environment of the current cards, but these changes also have security implications. Some differences of the Java Card, in comparison to Standard Java, increase the security risks and some lessen them. One thing that needs to be addressed is that much of the base security model in Standard Java is totally absent from Java Card.

Java Card properties increasing security

i. Lack of threads

The security analysis of the code is much easier when there is no threading. Threading is also difficult to implement correctly in the VM (Virtual Machine) and it is difficult to use without introducing potential security problems.

ii. Absence of dynamic class loading

It is well known that if you can confuse the VM about the types of objects it is manipulating, you can break the security model of Java. Removal of dynamic class loading makes type safety easier to enforce. Many of the Standard Java security problems have been related to type safety problems resulting from class loading.

Java Card properties decreasing security

i. Lack of garbage collection

Garbage collection is a good example of a qualitative security feature. Without an automatic system for freeing allocated memory it is difficult to program securely. Memory leaks are a well-known source of numerous security problems in the

traditional computing domain. This problem is especially bad in the card environment, where memory capacity is limited.

ii. Exception propagation problems

Uncaught exceptions could lead to a card becoming muted, i.e. non-responding. This creates another significant denial of service problem and raises the need for thorough quality testing before software release.

iii. Multiple applications and applet fire walling

There is a risk that competing stakeholders may have their applets on the same user's smart card. This might lead to attacks between applications. Applet fire walling and separation needs to be perfectly implemented in order to guarantee trusted environment for each application.

iv. Object-sharing loopholes

Object sharing is a high-demand feature, because it allows different applications to use the same code. This is very neat because there is always a shortage of memory on the card. It also allows the same tested code to be used in all necessary places. It is a potential source of security problems and some attacks have been hypothesized.

v. Access to native code

In a JCRE (Java Card Runtime Environment) environment, Java security is based on the quality of the underlying JCRE implementation. If it does contain security-related bugs, the whole idea behind Java security breaks. The same principle applies to the native code operations provided by the particular JCRE environment. If any of them provide features that do not follow the semantics of

Java security, they may offer loopholes that allow attack applets to carry out operations that would otherwise be stopped by the JCRE. It is also noteworthy, that native methods are native usually because of performance reasons i.e. the Java version of the same routine wouldn't be fast enough. The other possible reason is that nobody bothered to port the native code to Java. At least in the latter case, it is unlikely that the code would have been reviewed with Java security in mind.

2.2.9 Applications of Java Cards

In Sun Microsystems' stand #A06 in Albinoni Hall, Sun and its strategic partners demonstrates examples of products and technologies based on the open Java Card platform. Examples of products and technologies expected to be showcased at the Sun stand include multi-application PKI (Public Key Infrastructure) -enabled smart cards, digital identity cards, toolkits to enhance Java Card technology development, processors, etc.

Companies at Sun's booth include ActivCard, Banksys, Bull, Citigroup, Entrust Technologies, Gemplus, Giesecke & Devrient, Motorola, Oberthur, Schlumberger, ST Microelectronics and Visa. The displayed solutions are all written in Java Card technology and have the ability to run on any other platform. Advantages of the Java Card platform include the ability to create secure applications and services, reduce costs and shorten time to market

2.3 Elliptic Curve Cryptography

Here is a description of elliptic curve cryptography. First, a brief introduction to mathematical concepts is given.

2.3.1 On the mathematics of fields

This section deals primarily with the mathematics of the fields, and although elliptic curves are mentioned in several places, one should bear in mind that here not only

elliptic curve mathematics itself are discussed but also the underlying mathematics that the elliptic curve arithmetic needs. Elliptic curves can be defined over a mathematical structure called a *field*. Here it suffices to say that normal modulo arithmetic using integers together with multiplication and addition operations form a field. Generally, a field can consist of any kinds of objects that behave sufficiently similarly to the aforementioned example.

Currently, two fields are of special interest in implementing elliptic curve cryptography: Characteristic two finite fields, which are actually characteristic two polynomial finite fields and marked \mathbf{F}_2^m and prime fields \mathbf{F}_p . Here we present most of the necessary algorithms in \mathbf{F}_p . (Sometimes the field \mathbf{F}_p is denoted only with \mathbf{Z}_p even though it strictly means only the integers from 0 to p . \mathbf{F}_p , again strictly speaking, is a structure consisting of these elements and two operations $+$ and $*$. This distinction is not consistently held by the literature and sometimes \mathbf{Z}_p is used where \mathbf{F}_p would in fact be more correct.)

To understand the definition of field formally, we need to first introduce another mathematical structure called *group*; the definition is given shortly. Elliptic curve operations, take place on *additive (abelian) group*. Informally, this simply means that the group consists of group elements and that the only operation is called addition ($+$), which follows the well-known rules of addition. Formally, it means that the elements and operation must respect the abelian group axioms, which are defined as follows.

Definition 1. A *group* is a set G with binary operation \circ on G having the following properties:

- Associativity:

$$\text{For every } a, b, c \in G, a \circ (b \circ c) = (a \circ b) \circ c.$$

- Existence of identity:

$$\text{There is an element } e \in G \text{ such that } a \circ e = e \circ a = a,$$

for all $a \in G$.

- Existence of inverses:

For each $a \in G$, there exists an element $a^{-1} \in G$ such that $a \circ a^{-1} = a^{-1} \circ a = e$.

- Closure under the binary operation:

For all $a \in G$ and all $b \in G$ the element $(a \circ b) \in G$.

A group is called *abelian group*, if it also satisfies the following:

- Commutativity:

For all $(a, b) \in G$, $a \circ b = b \circ a$.

The number of the elements in G is called the *order* of the group G .

Next we proceed to a genuine extension of the abelian group, the definition of the field.

Definition 2. A *field* is a set F with two binary operations called addition (+) and multiplication (*) that have the following properties:

- F is abelian group with respect to addition (+). (i.e., all of the axioms from definition 1 are included here.)
- Multiplication associativity: $a * (b * c) = (a * b) * c$,
for all $a, b, c \in F$.
- Distributivity between addition (+) and multiplication (*):
 $a * (b + c) = a * b + a * c$ and $(b + c) * a = b * a + c * a$.
- The set $F \setminus \{e\}$ (non-zero elements from F) forms an abelian group under multiplication.

2.3.2 On the mathematics of elliptic curves

In the previous section, there was a brief description of field elements. Next, calculations with elliptic curves, which are defined over a field are discussed here. It should be carefully noted that elliptic curve operations (addition and scalar multiplication) take place on the curve, and because of that operation with points is done

instead of simple field elements as described in previous section. It is crucial to be able to make a distinction between adding field elements and adding points (i.e. pairs of field elements). Otherwise, it is impossible to understand the fact that the field operations are needed for doing elliptic curve operations, even though they are not the same thing.

Here is an example. Assume to add two points on some curve: (x_1, y_1) and (x_2, y_2) . That is to know what is $(x_1, y_1) + (x_2, y_2)$, (This is elliptic curve addition). Of course, somehow partition of the points to handle the elements x_1, x_2, y_1, y_2 separately is needed, and then call the result of this algorithm the sum of the two points. But these elements are exactly field elements that one knows how to add and multiply [8]! (Note that when the underlying field is F_p , then these elements x_1, x_2, y_1, y_2 are integers.)

The Elliptic curve equation

An elliptic curve E over F_p is of the form

$$y^2 = x^3 + ax + b \quad (1)$$

where $a, b \in \mathbb{Z}_p$ and $4a^3 + 27b^2 \neq 0 \pmod{p}$. Additionally there is a special point ∞ , called the point at infinity. The set $E(\mathbb{F}_p)$ has all such points (x, y) , where $x \in \mathbb{Z}_p$ and $y \in \mathbb{Z}_p$, and which also satisfy the equation (1).

Example: Elliptic curve over F_{23}

Let $p = 23$ and consider the elliptic curve $E, y^2 = x^3 + x + 4$ defined over F_{23} . As per the notation in equation (1), we have $a = 1$ and $b = 4$. Also $4a^3 + 27b^2 = 4 + 432 = 436 = 22 \pmod{23}$, so E is an elliptic curve.

EC Addition Formula

For elliptic curves, ∞ serves as an identity element. (for normal integer addition the identity element would be 0.)

1. $P + \infty = \infty + P = P$.

Informally this means that any point added with the identity element returns the point itself.

2. If $P_1 = (x_1, y_1) \in E(\mathbf{F}_p), P_2 = (x_2, y_2) \in E(\mathbf{F}_p),$ and $P_1 + P_2 = \infty.$

That is, a point P_i is a pair of elements and this point belongs to an elliptic curve defined over $\mathbf{F}_p,$ which is an ordinary integer field fixed by a prime $p.$ Then the P_2 is usually denoted by $-P_1,$ as the items cancel each other out and return the identity element (∞). In other words this means that adding a point to its inverse element returns the identity element.

3. If $P_1 = (x_1, y_1) \in E(\mathbf{F}_p)$ and $P_2 = (x_2, y_2) \in E(\mathbf{F}_p),$ where $P_1 \neq -P_2.$

The $P_1 + P_2 = (x_3, y_3) \in E(\mathbf{F}_p),$ where

$$x_3 = \lambda^2 - x_1 - x_2 \quad (2)$$

$$y_3 = \lambda(x_1 - x_3) - y_1 \quad (3)$$

where

$$\left. \begin{aligned} \lambda &= (y_2 - y_1) / (x_2 - x_1) \text{ if } p_1 \neq p_2 \\ &= (3x_1^2 + a) / 2y_1 \text{ if } p_1 = p_2 \end{aligned} \right\} \quad (4)$$

Informally this just means that any two points, that are both on the curve return a third point on the curve, whose coordinates can be calculated from the formulas (2), (3) and (4).

ANALYSIS & DESIGN

3.1 ECDSA (Elliptic Curve Digital Signature Algorithm)

The Elliptic Curve Digital Signature Algorithm (ECDSA) is the elliptic curve analogue of the DSA. Scott Vanstone first proposed ECDSA in 1992 in response to NIST's (National Institute of Standards and Technology) request for public comments on their first proposal for DSS (Digital Signature Standard). It was accepted in 1998 as an ISO (International Standards Organization) standard (ISO 14888-3), accepted in 1999 as an ANSI (American National Standards Institute) standard (ANSI X9.62), and accepted in 2000 as an IEEE (Institute of Electrical and Electronics Engineers) standard (IEEE 1363-2000) and a FIPS standard (FIPS 186-2). It is also under consideration for inclusion in some other ISO standards.

ECDSA is the elliptic curve analogue of the DSA. It operates on elliptic curves $E(\mathbb{F}_p)$ while DSA operates on the multiplicative prime group of \mathbb{Z}_p^* . That is, instead of working in a subgroup of order q in \mathbb{Z} , we work in an elliptic curve group $E(\mathbb{Z}_p)$. The ECDSA is currently being standardized within the ANSI X9F1 and IEEE P1363 standards committees. While DSA is a Discrete Logarithm (DL) scheme which base their security on the intractability of the ordinary discrete logarithm in a finite field where as ECDSA is a Elliptic Curve (EC) scheme which base their security on the intractability of the elliptic curve discrete logarithm problem. Table 3.1 shows the correspondence between some math notation used in DSA and ECDSA.

DSA Notation	ECDSA Notation
Q	N
g	P
x	d
y	Q

Table 3.1: Correspondence between DSA and ECDSA notation

The only significant difference between ECDSA and DSA is in the generation of r . The DSA does this by taking the random element $(g^k \bmod p)$ and reducing it modulo q ,

thus obtaining an integer in the interval $[1, q-1]$. The ECDSA generates the integer r in the interval $[1, n-1]$ by taking the x -coordinate of the random point kP and reducing it modulo n .

To obtain a security level similar to that of the DSA (with 160-bit q and 1024-bit p), the parameter n should have about 160 bits. If this is the case, then DSA and ECDSA signatures have the same bit length (320 bits).

Instead of each entity generating its own elliptic curve, the entities may elect to use the same curve E and point P of order n . In this case, an entity's public key consists only of the point Q . This results in public keys of smaller sizes. Additionally, there are point compression techniques whereby the point $Q = (x_Q, y_Q)$ can be efficiently constructed from its x -coordinate x_Q and a specific bit of the y -coordinate y_Q . Thus, for example, if $p = 2^{160}$ (so elements in \mathbf{Z}_p are 160-bit strings), then public keys can be represented as 161-bit strings.

Entire flow diagram of ECDSA is the following Figure 3.1.

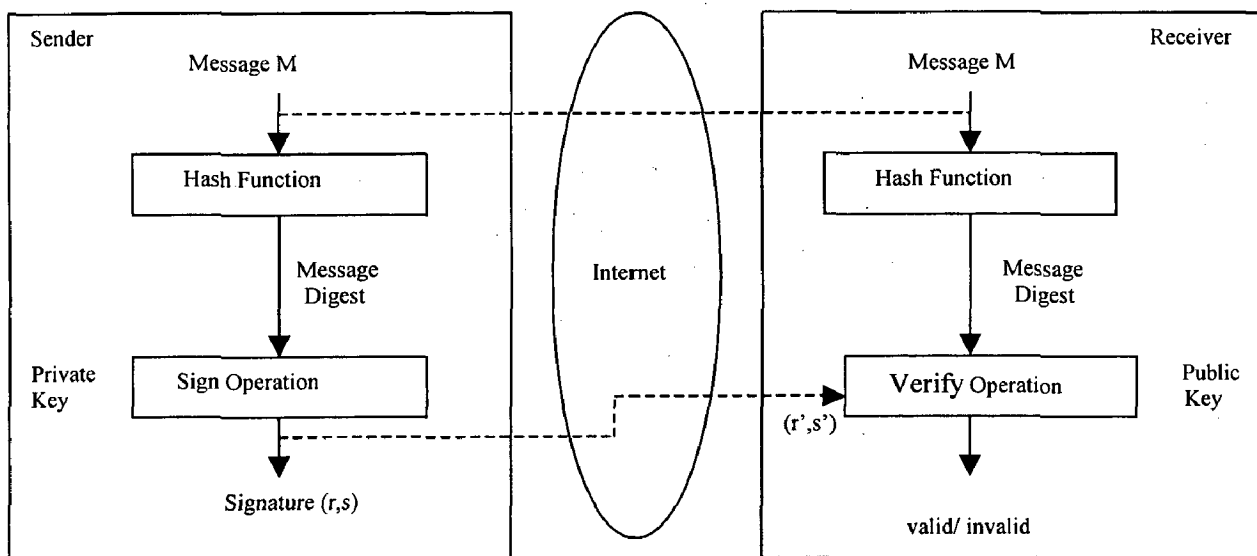


Fig 3.1 ECDSA flow diagram

3.2 Secure Hash Algorithm

The algorithm takes as input a message with a maximum length of less than 264 bits and produces as output a 160 – bit message digest. The input is processed in 512-bit blocks. The overall processing of a message follows the structure shown for MD5 in fig 3.2 .With a block length of 512 and a hash length and chaining variable length of 160 bits. The processing consists o the following steps:

Step 1: Append padding bits. The message is padded so that its length is congruent to 448 modulo 512 ($\text{length}=448\text{mod } 512$). Padding is always added even if the message is already of the desired length. Thus, the number of padding bits is in the range of 1 to 512, the padding consists of a single 1-bit followed by the necessary number of 0-bits.

Step 2: Append length. A block of 64 bits is appended to the message. This block is treated as an unsigned 64 –bit integer (most significant byte first) and contains the length of the original message (before the padding).

Step 3: Initialize MD buffer. A 160 –bit buffer is used t o hold intermediate and final results of the hash function. The buffer can be represented as five 32-bit registers (A, B, C, D, E). These registers are initialized to the following 32-bit integers (hexadecimals values):

A= 67452301

B=EFCDA89

C=98BADCFE

D=10325476

E=C3D2E1F0

Note that first four values are the same as those used in MD5 (Message Digest). However in the case of SHA-1, these values are stored in big–endian format, which is the more significant byte of a word in the low- address byte position. As 32-bit string s, the initialization values (in hexadecimals) appear as follows:

Word A: 67 45 23 01

Word B: EF CD AB 89

Word C: 98 BA DC FE

Word D: 10 32 54 76

Word E: C3 D2 E1 F0

Step 4: Process message in 512-bit (16-word) blocks. The heart of the algorithm is a module that consists of four rounds of processing of 20 steps each. The four rounds have a similar structure, but each use a different primitive logical function, which we refer to as f_1 , f_2 , f_3 , and f_4 .

Each round takes as input the current 512-bit block being processed (Y_q) and the 160-bit buffer value ABCDE and updates the contents of buffer. Each round also makes use of an adaptive constant K_t , where $0 \leq t \leq 79$ indicates one of the 80 steps across our rounds. In fact only our constants are used. The values, in hexadecimal and decimal, are as follows:

The output of the fourth round (eightieth step) is added to the input to the first round (CV_q) to produce (CV_{q+1}). The addition is done to the independently on each of five words in the buffer with each of the corresponding words in (CV_q), using addition modulo 2^{32} .

Step 5: Output. After all 512-bit blocks have been processed, the output from the L th stage is the 160-bit message digest.

We can summarize the behavior of SHA1 as follows.

$$CV_0 = IV$$

$$CV_{q+1} = \text{SUM}_{32}(CV_q, ABCDE_q)$$

$$MD = CV_L$$

Where

IV = initial value of the ABCDE buffer, defined in step 3

$ABCDE_q$ = the output of the last round of processing of the q th message block.

L = the number of the blocks in the message (including padding and length fields).

SUM_{32} = addition modulo 2^{32} performed separately on each word of the pair of inputs.

MD = final message digest value

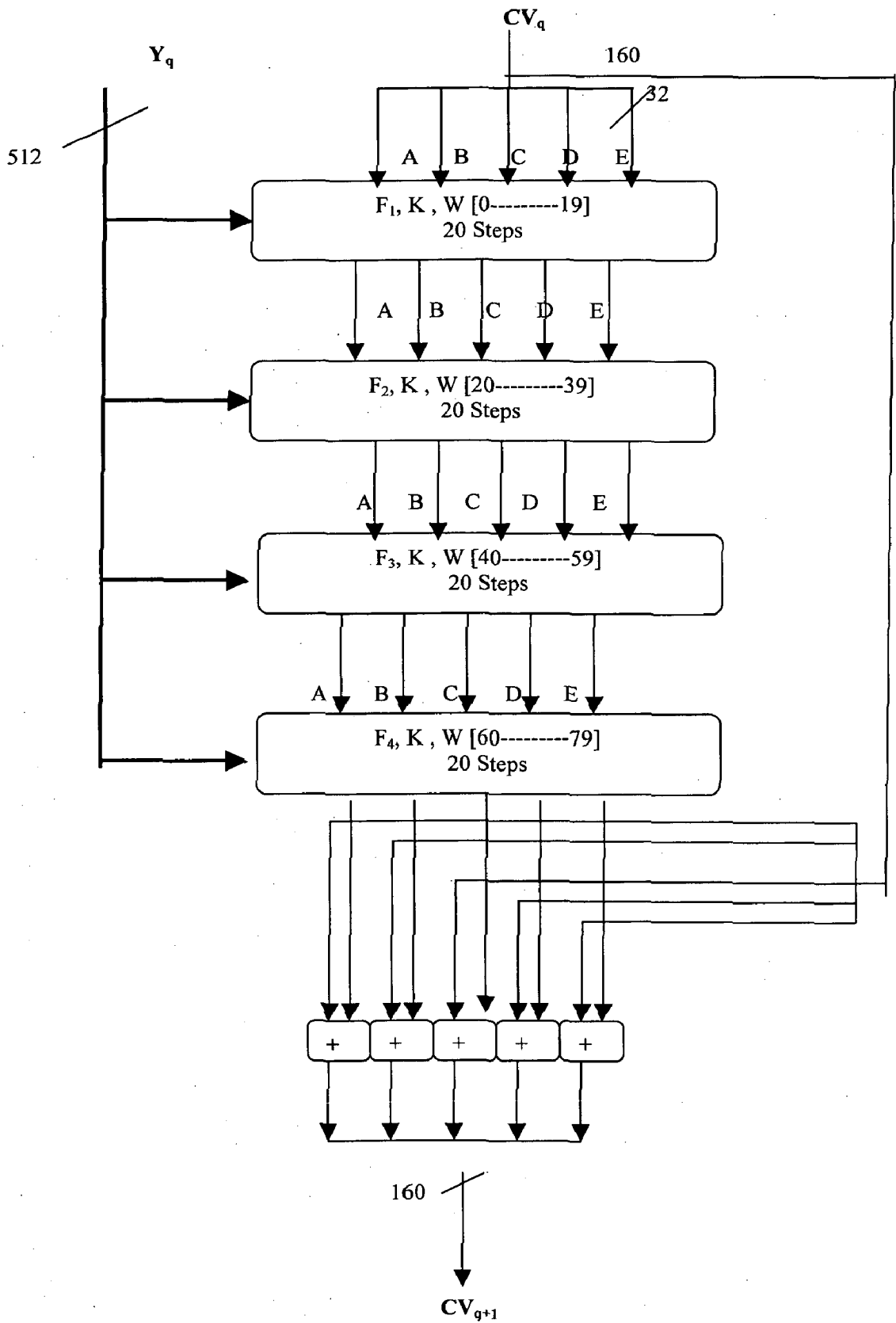


Fig 3.2: SHA-1 Algorithm

3.3 Java Card applet

A Java Card applet is a smart card application written in the Java programming language and conforming to a set of conventions so that it can run within the Java Card runtime environment (JCRE)[1]. Applets, like any smart card applications, are reactive applications. Once selected, a typical applet waits for an application running on the host side to send a command. The applet then executes the command and returns a response to the host.

This command-and-response dialogue continues until a new applet is selected or the card is removed from the card acceptance device. The applet remains inactive until the next time it is selected.

The communication between an applet and a host application is achieved through exchanging APDUs, as illustrated Figure 3.3. An APDU contains either a command or a response. A host application sends a command to an applet and the applet returns a response.

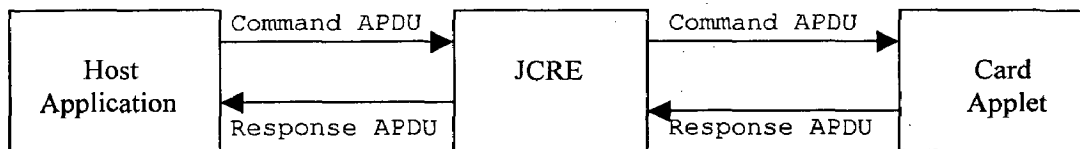


Fig 3.3: Java Card Applet Communication

APDU are data packets; they are the application level communication protocol between the application software on the card and the application software on the host side of the link. The APDU protocol, as specified in ISO 7814-4, is an application-level protocol between a smart card and a host application.

APDU message under ISO 7816-4 comprise two structures: one used by the host application at the CAD side of the channel to send commands to the card, the other used by the card to send responses back to the host application. The former is referred to as the

command APDU and the latter as the response APDU. A command APDU is always paired with a response APDU. Their structures are illustrated in Figure 3.4 and 3.5. The structure of the APDU is being standardized by ISO 7816-4.

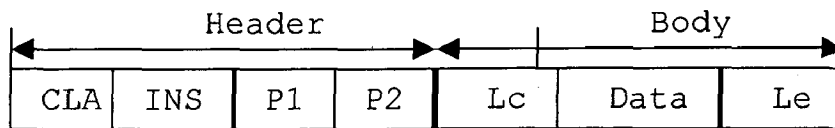


Fig 3.4: Command APDU Structure

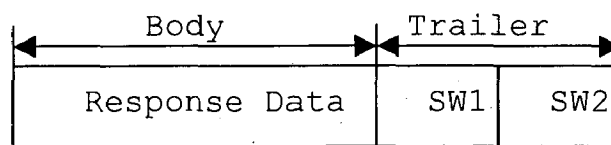


Fig 3.5: Response APDU Structure

- CLA : Class byte. It is to identify the application.
- INS : Instruction byte. To indicate the instruction code.
- P1-P2 : Parameter bytes. To provide further qualification of the APDU.
- Lc : It indicates the number of bytes in the data field.
- Data field: The slot where data is actually allocated in the package.
- Le : It is the maximum number of bytes expected in the data field in the next response APDU.
- SW : Status word. It indicates the status of the applet. Reader can notify the occurrence of exception via the status words.

IMPLEMENTATION

4.1 Implementation of the algorithm

The key generation, signature generation, and signature verification procedures for ECDSA are as follows:

Key generation

1. Select an elliptic curve E defined over F_p . The number of points in $E(F_p)$ should be divisible by a large prime number n .
2. Select a point $P \in E(F_p)$ of order n .
3. Select a statistically unique and unpredictable integer d in the interval $[1, n-1]$.
4. Compute $Q = dP$.
5. The public key is (E, P, n, Q) and the private key is d

Here Q is a point that we get by multiplying P with the private key d .

Signature generation

To sign a message m , a sender should perform the following steps:

1. Select a statistically unique and unpredictable integer k in the interval $[1, n-1]$.
2. Compute $kP = (x_1, y_1)$ and $r = x_1 \bmod n$. (Here x_1 is regarded as an integer for example by conversion from its binary representation.)
If $r = 0$ then go to step1. (This is security condition i.e., if $r = 0$, then the signing equation $s = k^{-1}\{h(m)+dr\} \bmod n$ does not involve the private key d).
3. Compute $k^{-1} \bmod n$.
4. Compute $s = k^{-1}\{h(m)+dr\} \bmod n$, where h is the Secure Hash Algorithm(SHA-1).
5. If $s = 0$ then go to step1. (If $s = 0$ then $s^{-1} \bmod n$ does not exist, s^{-1} is required in the step2 of signature verification process.)

6. The signature for the message m is the integer pair (r, s) .

Signature verification

To verify the sender's signature (r, s) on the message m , a receiver should perform the following:

1. Obtain an authentic copy of signer's public key (E, P, n, Q) . Verify that r and s are integers in the interval $[1, n-1]$.
2. Compute $w = s^{-1} \bmod n$ and $h(m)$.
3. Compute $u_1 = h(m) w \bmod n$ and $u_2 = rw \bmod n$.
4. Compute $u_1P + u_2Q = (x_0, y_0)$ and $v = x_0 \bmod n$.
5. Accept the signature if and only if $v = r$ which shows that the signature is verified and is not forged.

This algorithm is implemented in Java language and tested first using applets where a signer can create a document, generate keys, sign the document and send them across to other party in the form of a file which contains public key, document and digital signature. Upon receiving this file, the other party can verify the signature of the sender.

Here, a care taken such that all the characteristics of digital signature are fulfilled i.e.,

- This signature is unforgeable. Only the sender knows the private key, so it is difficult to forge his/her signature.
- The signed document is unalterable. If the receiver wants to alter the document, then the message digests will differ and the signature will never gets verified.

To implement this algorithm in Java, the following packages have been used:

java.math.* -- Provides classes for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal).

java.awt.* -- Contains all of the classes for creating user interfaces

java.awt.event.* -- Provides interfaces and classes for dealing with different types of events fired by AWT components.

java.security.* -- Provides the classes and interfaces for the security framework.

java.io.* -- Provides for system input and output through data streams, serialization and the file system

java.lang.*-- Provides classes that are fundamental to the design of the Java programming language.

Functions in SHA

Cshift()

This function accepts the number and the number of times that number has to be circular shifted. And then computes the value after circular shifting it for the number of times and returns the result.

Computepadbits()

This function accepts the length of the message and returns the number of padding bits.

4.2 Running on Java Card

Originally to run on Java card the following steps are followed as shown in fig. 4.1.

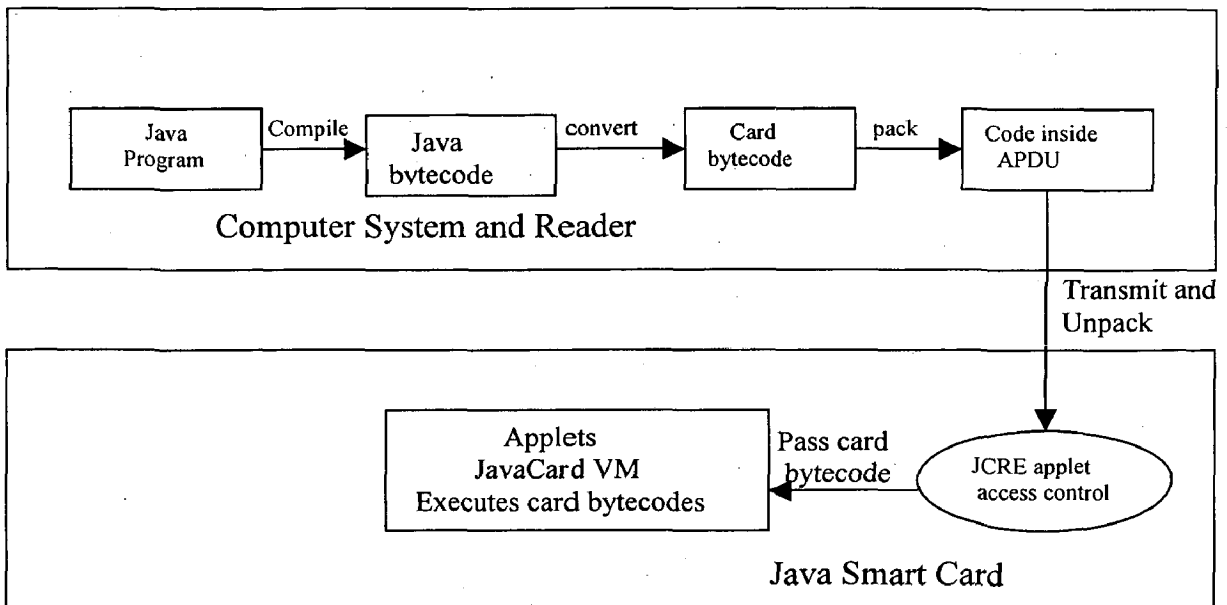


Fig 4.1 Data transfers to a Java smart card

Here is the step by step data transfer procedure.

- Like other Java program, data written in Java is compiled into Java bytecode.
- JavaCard Virtual Machine verifies and converts Java bytecode into Card bytecode.
- Card bytecode and the extended information are packed into Command APDU.
- Reader transmits the Command APDU to Java smart card.
- Unpack the Command APDU.
- JCRE identifies which applet the data belongs to.
- JCRE dispatches the data to the particular applet and perform execution in JavaCard VM.

ECDSA is implemented using Java language and is tested in Java Card platform on PC environment.

In order to run on Java card, we need to develop a Java Card Applet. Development of Java Card Applet begins with the following steps:

Step1: Setting up the environment.

A batch file named cardnew.bat for setting up the java card environment and running this batch file also helps in invoking the java and java card batch files like javac, apdutool etc.,

cardnew.bat contains

cardnew.bat
<pre>@echo off set JC_HOME=C:\java_card_kit-2_2 set JAVA_HOME=c:\jdk1.3 set PATH=%JC_HOME%\bin;%PATH% set PATH=%PATH%;%JAVA_HOME%\bin</pre>

Step2: Write the program which contains APDU processes.(here the program is eccard.java)

Step3: Compile the eccard.java source code with java compiler. The output is a class file(ie., eccard.class)

Step4: Create a command configuration file i.e., the commands to be run on a converter. The configuration file created here is eccard.opt

```
eccard.opt
-out EXP JCA
-exportpath ..\api21
-applet 0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x6:0x1
com.sun.javacard.samples.eccard.eccard
com.sun.javacard.samples.eccard
0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x6 1.0
```

Here,

-out tells the converter to output a exp (export) file and a jca (Java card assembly) file.

-exportpath specifies the root directories in which the converter will look for export files.

-applet sets the default applet AID and the name of the class that defines the applet.

com.sun.javacard.samples.eccard is package name

0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x6 is package AID

1.0 is major_version.minor_version

Step5: Using **Converter**, which loads and processes class files that make up a Java package. The Converter outputs an export file and a JCA (Java Card Assembly) file, which you then input to capgen to produce a CAP file. A JCA file is a human-readable ASCII file to aid testing and debugging. The two generated files i.e., exp and jca files will go to a folder called javacard which is automatically created. So, the converter outputs eccard.exp and eccard.jca.

Step6: **capgen** is backend to the converter and is used to convert a JCA file into a CAP file. So it outputs eccard.cap.

Step7: Generate script file for **apdutool** using the **scriptgen** tool. So it outputs a script file called **eccard.scr**. There are certain additional things to be added to the script file that is generated.

Add the following code at the beginning of script file

```
powerup;  
  
// Select the installer applet  
0x00 0xA4 0x04 0x00 0x09 0xA0 0x00 0x00 0x00 0x62 0x03 0x01 0x08 0x01 0x7F;
```

Add the following to the end of the script file.

powerdown;

A powerup command must be executed prior to sending any C-APDUs.

A powerdown command must be executed after sending the C-APDUs.

RESULTS AND DISCUSSION

5.1 Running on Java Development Kit

Open a command window and set the JDK path by invoking go.bat batch file which contains

```
@echo off  
set path=c:\jdk1.3\bin
```

Then run the following command: `c:/fresh/appletviewer/menupro.java`. It will open the following window as shown in fig 5.1

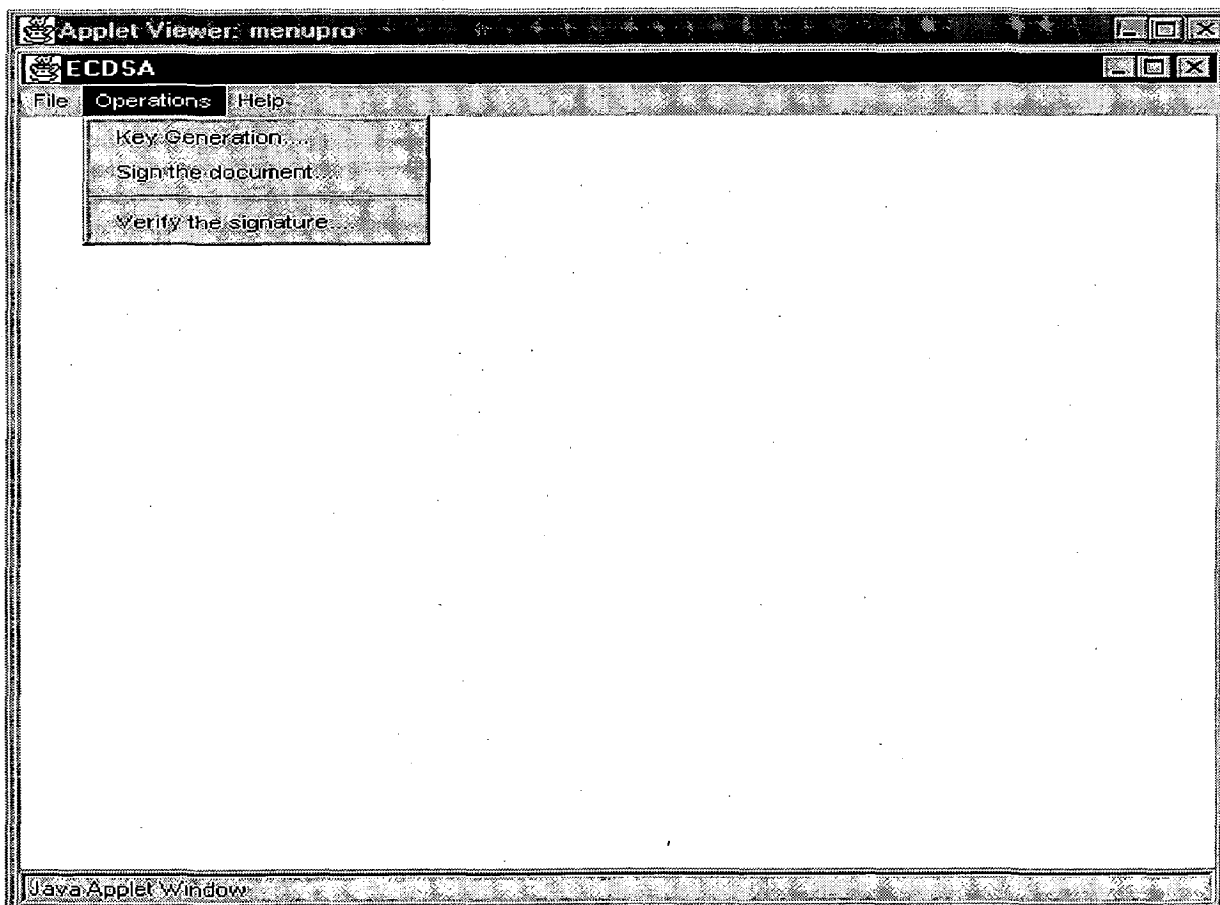


Fig 5.1: Main window which contains menu bar

The main window contains three menu headings as File, Operations and Help. To create a document, click on file->Create Document.... option. This will open a notepad where one can type a document or a message which he/she wants to sign.

For key generation: click on Operations->Key Generation... option, it will open a new window as shown in fig 5.2

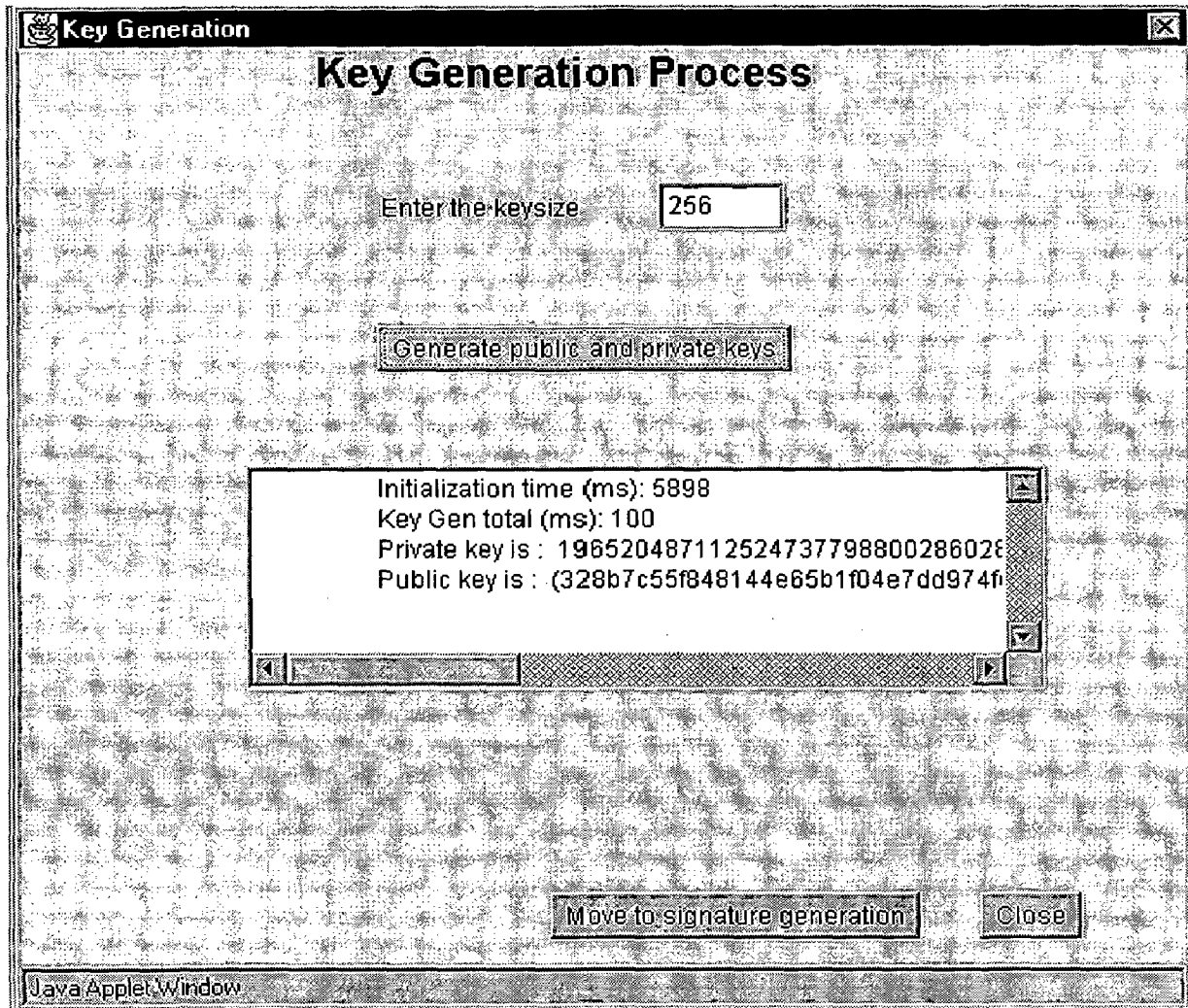


Fig 5.2: Key generation process

First enter the key size in the text box provided. This key size that is chosen is meant to generate public and private keys. The allowed key sizes are 160, 192, 224, 256, 384, 521. For example if 176 is entered as key size it will take it as 192. If 193 is entered as key size it will take 224. After entering key size, click on "Generate public and private keys" button, will display the public and private keys generated in the text area provided.

Also the time taken to initialize and generate keys is displayed. Next to sign the document, click on Move to signature generation button. It will open a new window as shown in fig 5.3

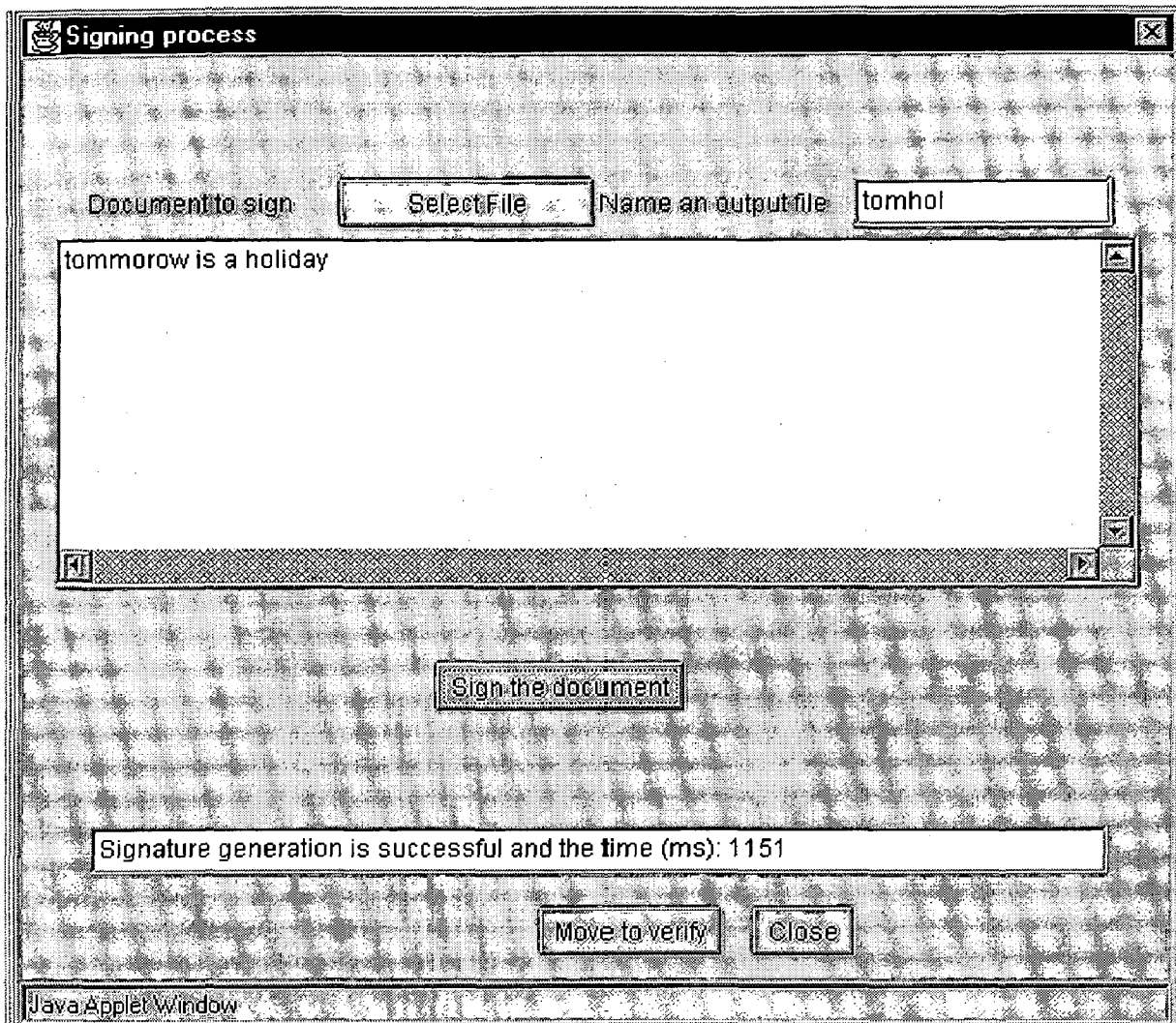


Fig 5.3 Signature generation process

Select the document to sign by just clicking on Select File button. Now, select the document that is to be signed. Name the output file, which has to be sent across to others. This output file contains public key, message and the signature. If giving name to an output file is forgotten, it defaults it to resout.txt. Then click on Sign the document. The signature generation process is successful if a message in the below text box where its successful state along with time taken for signature generation is displayed. To verify the signature, if the signer wants to verify the signature of the document that is to be sent then click on move to verify button provided on signature generation applet. If the

receiver wants to verify the signature then click on Operations->Verify the signature....This will open a verification window as shown in fig 5.4

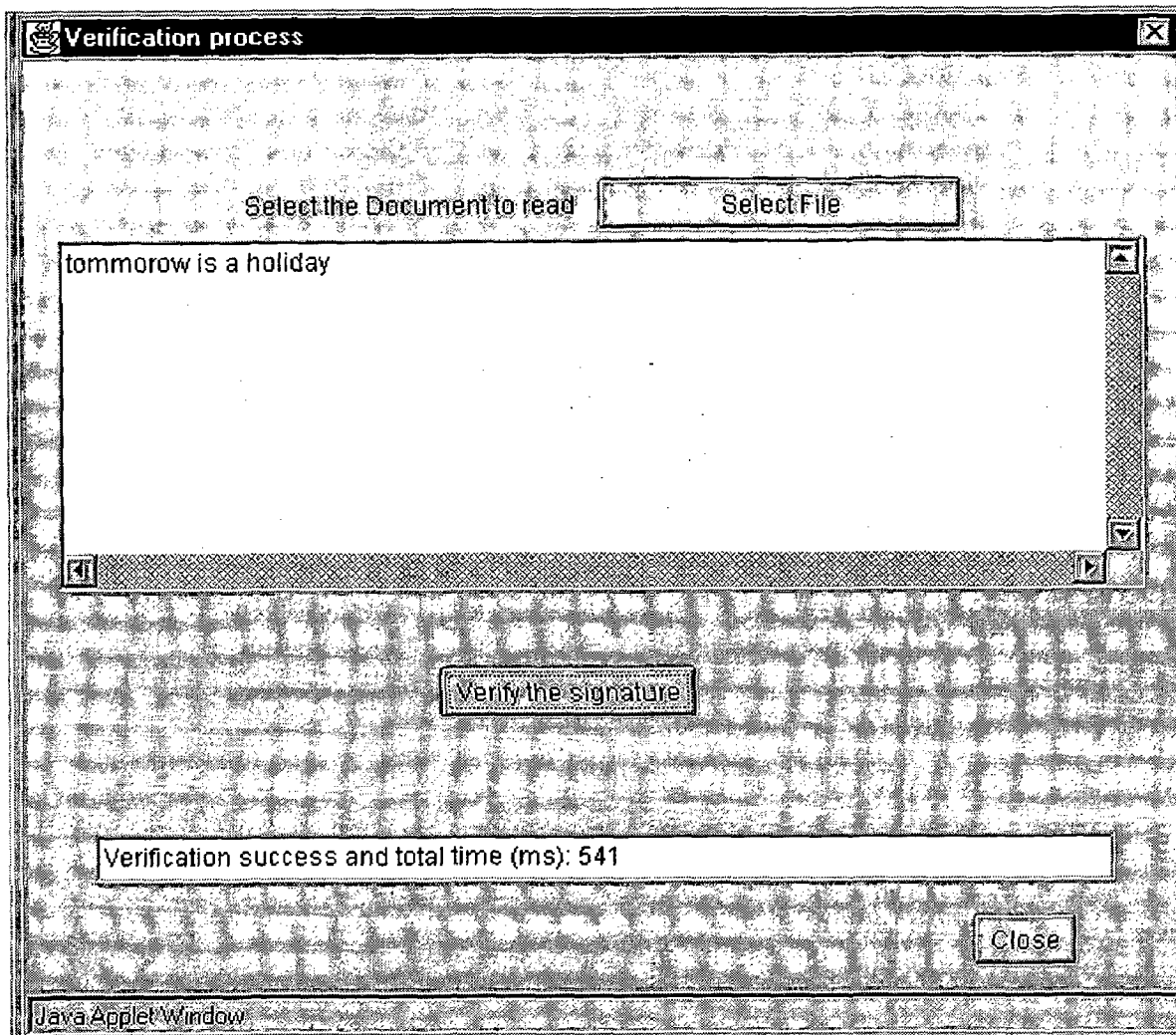


Fig 5.4 Verification process

Here first select the file that is received from the signer by clicking on select file. As soon as the file is selected, the message is displayed in the text area. To verify the signature of the sender, click on Verify the signature. The verification time taken along with the status is shown based on whether it is verified or not.

To know to how to run this software, click on Help->How to run....

5.2 Running on Java Card Kit

Follow the steps as shown in section 4.2 of chapter 4. Following those steps will result in the output as shown below.

Step1: Setting up the environment C:\>cardnew

Step2: Compiling the created java file

Commands to be given
C:\>cd jc21\samples\com\sun\javacard\samples\eccard
C:\jc21\samples\com\sun\javacard\samples\eccard>javac -classpath c:\java_card_kit-2_2\lib\api.jar eccard.java

Step3: Use converter, which processes class file

Commands to be given
C:\jc21\samples\com\sun\javacard\samples\eccard>cd\
C:\>cd jc21\samples
C:\jc21\samples>converter-config com\sun\javacard\samples\eccard\eccard.opt

Output:
Java Card 2.2 Class File Converter (version 1.3) Copyright 2002 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms. conversion completed with 0 errors and 0 warnings.

Step4: Create a cap file from jca file that is generated using capgen tool.

Command to be given
C:\jc21\samples>capgen -o eccard.cap com\sun\javacard\samples\eccard\javacard\eccard.jca

Output:
Java Card 2.2 CAP File Builder (version 0.55) Copyright 2002 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.

Step5: Generate the script file with the help of cap file created.

Command to be given
C:\jc21\samples>scriptgen -o eccard.scr eccard.cap

Output:
Java Card 2.2 APDU Script File Builder (version 0.11) Copyright 2002 Sun Microsystems, Inc. All rights reserved. APDU script file for CAP file download generated.

There are certain additional things to be added to the script file that is generated.

Add the following code at the beginning of script file

powerup; // Select the installer applet 0x00 0xA4 0x04 0x00 0x09 0xA0 0x00 0x00 0x00 0x62 0x03 0x01 0x08 0x01 0x7F;
--

Add the following to the end of the script file.

powerdown;

Step6: Open two command windows. In one window enter the following command to start C-JCRE, C language Java Card Runtime Environment which has the ability to simulate persistent memory (EEPROM), and to save and restore the contents of EEPROM to and from disk files. In another window start the APDU tool which reads a script file containing APDUs and sends them to the C-JCRE. Each APDU is processed by the JCRE and returned to the APDUTool, which displays both the command and response APDUs on the console or redirects the output to a file that is specified. Both these command window looks like this:

Command to be given
C:\jc21\samples>cref

Output:
Java Card 2.2 C Reference Implementation Simulator (version 0.41) Copyright 2002 Sun Microsystems, Inc. All rights reserved. Memory configuration
Type Base Size Max Addr
RAM 0x0 0x500 0x4ff
ROM 0x1000 0x8000 0x8fff
E2P 0x9020 0x3fe0 0xcfff
ROM Mask size = 0x4a31 = 18993 bytes
Highest ROM address in mask = 0x5a30 = 23088 bytes
Space available in ROM = 0x35cf = 13775 bytes
Mask has now been initialized for use
C-JCRE was powered down.

Command to be given

```
C:\jc21\samples>apdutool -o eccard.scr.out eccard.scr
```

Output:

Java Card 2.2 ApduTool (version 0.20)

Copyright 2002 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.

Opening connection to localhost on port 9025.

Connected.

Received ATR = 0x3b 0xf0 0x11 0x00 0xff 0x00

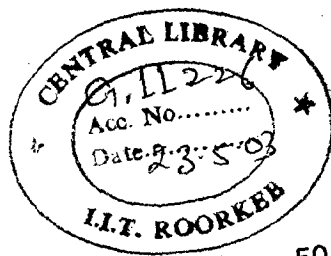
CONCLUSION

To use smart card as secure authentication tokens, the basic cryptographic services must be available in the card environment. The digital signature capability is important on a smart card because the users want not only to access services but also authorize others by signing authorization certificates. In this dissertation there is design and implementation of ECDSA using Java programming language acted in the Java Card platform because of its shorter key lengths which correctly fit to the limited environment of the cards. As smart cards are not very fast, the shorter key lengths resulting in faster execution also favours to use this algorithm when compared to other digital signature algorithms.

Here, ECDSA is implemented by using Java programming language and tested firstly on JDK and then on Java Card Kit. It has been simulated on the PC itself. As a part of future work, one needs to test it by importing the applet that is created on the Java card and insert it into the actual card acceptance device.

REFERENCES

1. Chen, Zhiqun, "Java CardTM Technology for Smart Cards", ADDISON-WELSEY Company, 2000 42-72p.
2. Rinaldo Di Giorgio, "Smart cards: A primer", An Article, Java developer series, 1997.
3. Bruce Schneier, Applied Cryptography: Digital Signatures, Second Edition, John Wiley & Sons, Inc., 1996, p. 34.
4. Y.L.Chan, H.Y. Chan, "Java Smart Cards", 1998.
5. Don B. Johnson, Alfred J. Menezes, "*Elliptic Curve DSA (ECDSA): An Enhanced DSA*", Certicom ECC Whitepapers, available through <http://www.certicom.com/>.
6. Dr. Dobb, "Java Card History", Dr.Dobb's journal, February 1999.
7. <http://java.sun.com/products/javacard/datasheet.html>
8. Lasse Leskelä, "Implementing Arithmetic for Elliptic Curve Cryptosystems", Master's Thesis, Helsinki University of Technology, January 1999.
9. Java Card 2.2TM Application Programming Interface, Sun Microsystems, Inc., September, 2002.



Java Card Kit

The Java Card™ 2.2 Development Kit tools run on a workstation using a Java Virtual Machine (VM). The Java Card 2.2 Virtual Machine is written in the "C" programming language. Separate bundles for the Solaris® and Windows NT® versions are available [9]. Figure A.1 describes the family of different Java architectures for different purposes.

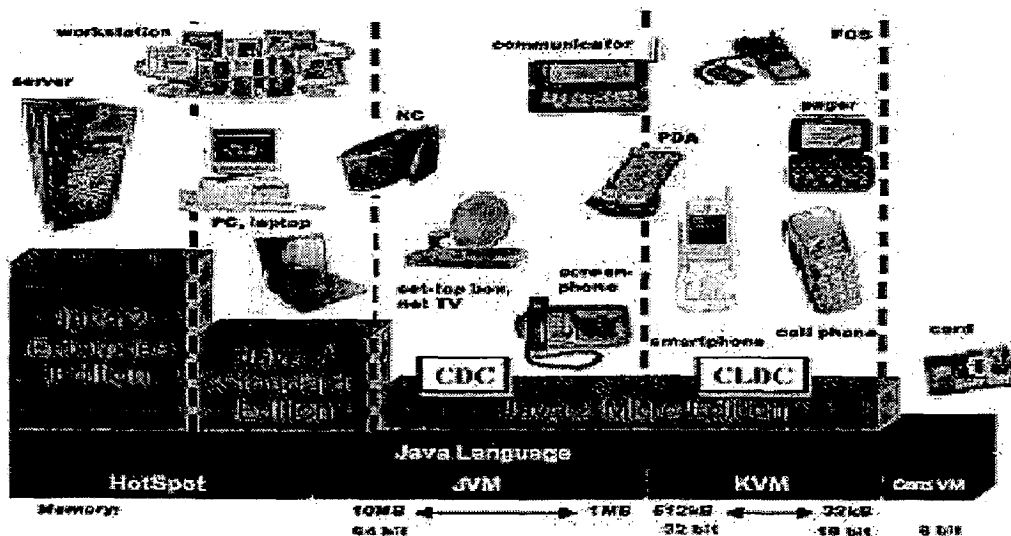


Fig A.1: The picture illustrates the different VMs and APIs of Java 2

Java Card API 2.1 consists of the following four fundamental packages [9]:

java.io A subset of the java.io package in the standard Java programming language.

java.lang Provides classes that are fundamental to the design of the Java Card technology subset of the Java programming language.

java.rmi The java.rmi package defines the Remote interface which identifies interfaces whose methods can be invoked from card acceptance device (CAD) client applications.

javacard.framework Provides a framework of classes and interfaces for building, communicating with and working with Java Card applets.

javacard.framework.service Provides a service framework of classes and interfaces that allow a Java Card applet to be designed as an aggregation of service components.

javacard.security Provides classes and interfaces that contain publicly-available functionality for implementing a security and cryptography framework on Java Card.

javacardx.crypto Extension package that contains functionality, which may be subject to export controls, for implementing a security and cryptography framework on Java Card.

Files Installed for the Binary Release

The files and directories that the binary installation procedure installs under `java_card_kit-2_2` are [9]:

`api_export_files` Directory contains the export files for the Java Card 2.2 API packages.

`bin` Directory contains all shell scripts and batch files for running the tools (such as the `apdutool`, `capdump`, `converter` and so forth), and the `cref` binary executable.

`doc` The `doc/en/guides` directory contains the English-language guides for this release. It includes the present document: the *Java Card. 2.2 Development Kit User's Guide* and the *Java Card. 2.2 Application Programming Notes*. The `doc/en/whitepapers` directory contains the English language white papers for this release. It contains the *Java Card™ 2.2 Off-Card Verifier* and the *Java Card™ 2.2 RMI Client Application Programming Interface* white papers.

lib Directory contains all Java jar files required for the tools. It also contains api.jar that is needed to write Java Card applets and libraries, javacardframework.jar, apduio.jar that is used by the apdutool, jcwde.jar that is used by JCWDE(Java Card workstation Development Environment), jcclientsamples.jar that contains the client part of the Java Card RMI samples, and jermiclientframework.jar that contains the classes of the Java Card RMI(Remote Method Invocation) Client API.

Samples Directory contains sample applets and demonstration programs. For more information on the contents of this directory, see "Sample Programs and Demonstrations".

COPYRIGHT_gl (COPYRIGHT_gl.txt on Windows) Contains the copyright notice for the product.

README.html Contains general information about this release.

RELEASENOTES.html Contains important information about this release.

LICENSE.html Contains the text of the license agreement.

