Evaluating Elliptic Curve Cryptography for Use on Java Card

Nadejda Pachtchenko

Master of Science (M.Sc)

Letterkenny Institute of Technology

Dr. Mark Leeney

Submitted to the Higher Education and Training Awards Council, September 2003

Declaration

I hereby declare that with effect from the date on which this dissertation is deposited in Library of Letterkenny Institute of Technology, I permit the Librarian of Letterkenny Institute of Technology to allow the dissertation to be copied in whole or in part without reference to the author on the understanding that such authority applies to the provision of single copies made for study purposes or for inclusion within the stock of another library. This restriction does not apply to the copying or publication of the title or abstract of the dissertation. It is a condition of use of this dissertation that anyone who consults it must recognise that the copyright rests with the author, and that no quotation from the dissertation, and no information derived from it, may be published unless the source is properly acknowledged.

Acknowledgement

This Master's thesis has been done for Letterkenny Institute of Technology.

I want to thank my supervisor, Dr Mark Leeney, for his help and comments.

I wish to thank my co-worker Jim Stevens who read and commented on the draft versions of this thesis.

I would also like to thank Dmitri Surkov for giving me references and information on Smart Cards.

My gratitude also goes to Thomas Dowling for his comments.

Finally, I would like to thank my family for their patience and advice.

Nadejda Pachtchenko

Abstract

Smart cards are used as trusted storage and data processing systems to store cryptographic private keys and other valuable information. Java Card promises the ease of programming in Java to the world of smart cards. Java's memory model however is resource intensive especially for smart card hardware. In this paper the software implementation of the elliptic curve cryptography on Java Card is discussed. This work also covers the description and implementation of the elliptic curves used in application and Nyberg-Rueppel elliptic curve algorithms. Furthermore, The testing methods and the test results concerning the performance of the operations, security, and the space required to store the keys are discussed.

Contents

Table of Figures

Table of Tables

Chapter 1. Introduction

Smart cards are often used as trusted storage and data processing systems to store cryptographic private keys and other valuable information. This means that they are usually used as part of larger access control or authorization architectures, which are becoming more commonplace. As a result of this, both the usage of smart cards and the corresponding development environments have greatly expanded since their introduction. Java Card promises the ease of programming in Java to the world of smart cards. Java programmers can develop smart card code and that code can be downloaded to cards that have already been issued to customers. This flexibility and post-issuance functionality can significantly extend smart cards potential uses. However, until very recently, such promises have not been backed by real implementations; Java Card uses has been limited to reference implementations – better known as simulations.

Java Card is a typical smart card: it conforms to all smart card standards and thus requires no change to existing smart card-aware applications. However, programming smart cards is inherently harder than programming in a desktop environment for several reasons:

- They lack natural input and output.
- Their processor and memory capacities are limited.
- The standards are few and not very well followed by the industry.
- The development environments and languages for the cards have been archaic.

Lack of proper input and output means that other systems such as PC's are needed as essential parts of the development. Having additional complete systems in the development process tends to make things harder as there are more things that can go wrong. Additionally, more skills and tools are required to get the actual job done.

The focus of this document is the software implementation on Java Card of elliptic curve cryptography. The most significant constraint in this environment is processing power and memory. Java's memory model is resource intensive even for smart card hardware. The results of elliptic curve implementation in Java are presented. These results serve to validate the statement that 'the Java platform provides reliability and trust through three key attributes: Simplicity, Safety and Security' [54].

To reach this objective, elliptic curve based cryptography has been studied to see how it can be applied to sign and verify messages, and to encrypt and decrypt messages on Java cards. To facilitate the studies the Nyberg-Rueppel signing and verification algorithms for smart cards in Java have been implemented, and some tests on a smart card emulator have been carried out to check the performance of the operations. In addition have been found out of how much memory is used with elliptic curve implementation and how well the statement "write once run anywhere" [54] works in practice.

This work also covers the description and implementation of the elliptic curves used in application and Nyberg-Rueppel elliptic curve algorithms. Furthermore, the testing methods and the test results concerning the performance of the operations, security, and the space required to store the keys are described. On the basis of the test results some analysis and

estimates of how suitable is Java Card for implementing elliptic curve cryptosystem have been presented.

This thesis has the following structure:

Chapter 2 includes a description of smart cards, including the components of smart cards such as operating system, transport protocol, security and attacks.

Chapter 3 describes Java Card technology. Since Java Card is a typical smart card and conforms to all applicable standards [1] this chapter concentrates on building and installing applications on Java Card.

Chapter 4 explains why encryption and digital signatures are used, and describes the two main cryptosystem types: private key cryptosystems and public key cryptosystems, including digital signatures.

Chapter 5 describes the basic mathematical theory behind the elliptic curve cryptosystem and explains the Nyberg-Rueppel signature algorithm and why this algorithm was chosen for implementation.

Chapter 6 describes a Java Card emulator and presents detailed description of how the application was implemented. The implementation of elliptic curve cryptosystems is also described.

Chapter 7 presents test results obtained with elliptic curve cryptosystem implementations.

Chapter 8 summarizes the information gained in this thesis and gives an opinion on how well Java Card is working.

Chapter 2. Smart Cards

Few years ago were predicted that smart cards one day would be as important as computers. This statement is not entirely correct because it implies that smart cards are not computers, when in fact, they are. In this chapter the history of smart cards, some different types, their low-level properties, the standards that affect their adoption in mainstream society, and how they relate to today's computer security systems is described.

2.1 Introduction

A smart card is a portable computer with a programmable data store. It is the exact shape and size of a credit card, and holds and processes 16KB or more of sensitive information. The central processing unit in a smart card can be an 8-, 16- and 32-bit micro controller with embedded application logic ROM. To make a computer and a smart card communicate the card is placed in or near a smart card reader (see 2.7 for more details on card reader), which is connected to the computer.

Today, there are more than two billion smart cards in use. In 1999 Dataquest market research forecasted that by the year 2001, 3.4 billion smart cards would be used worldwide. Smart card activities will grow at 30 percent a year [73]. Over the last five years, the industry has experienced steady growth, particularly in cards and devices to conduct electronic commerce and to enable secure access to computer networks. Within the same time frame, smart cards were used in 95 percent of the digital wireless phone services offered worldwide [74].

2.2 History

The proliferation of plastic cards started in the USA in the early 1950s. At first, the cards' functions were quite simple. They initially served as data carriers that were secure against forgery and tampering. The first improvement consisted of a magnetic strip on the back of the card. This allowed digital data to be stored on the card in machine-readable form, as a supplement to the visual data. The embossed card with a magnetic strip is still the most commonly used type of payment card. Magnetic strip technology suffers from a critical weakness; in that the data stored on the strip can be read, deleted and rewritten at will by anyone with access to programming equipment. Thus it is unsuitable for the storage of confidential data [7].

The development of the Smart Card, combined with the expansion of electronic data processing, has created completely new possibilities for solving this problem. In addition to a high degree of reliability and security against tampering, Smart Card technology promised the greatest flexibility in future applications. With modern cryptographic procedures, the strength of the security mechanism in electronic data processing systems can be mathematically calculated. It is not necessary to rely on the very subjective assessment of conventional techniques, whose security essentially rested on the secrecy of the procedures used [7].

2.3 Types of Cards

Smart Cards are defined according to the type of chip implanted in the card and its capabilities [59]. There is a wide range of options to choose from for the new system.

Memory Cards. Typical memory card applications are pre-paid telephone cards and health insurance cards. Memory cards are not very expensive but at the same time are not very functional.

Straight Memory Cards. These cards store data and have no data processing capabilities. These cards are the lowest cost per bit for user memory. They might be regarded as floppy disks.

Protected / Segmented Memory Cards. These cards have built-in logic to control the card memory access. Sometimes referred to as Intelligent Memory cards these devices can be set to write protect some or all of the memory blocks.

Stored Value Memory Cards. These cards are designed for the specific purpose of storing values or tokens. The cards are either disposable or rechargeable. Most cards of this type incorporate permanent security measures at the point of manufacture.

Cryptographic Coprocessor Cards. These cards are in the same category as microprocessor cards, but are different in cost and functionality. Cryptographic coprocessing is added to the architecture to reduce some mathematical operations to around a few hundred microseconds.

Contactless Smart Card. These cards don't need to be inserted into a reader, which could improve end user acceptance. No chip contacts are visible on the surface of the card so that card graphics can be used with more freedom.

Optical Memory Cards. These cards can carry huge amounts of data, but with today's technology the cards can only be written once and data cannot be erased.

CPU/MPU Microprocessor Multifunction Cards. These cards have on-card dynamic data processing capabilities. Multifunction smart cards allocate card memory into independent sections assigned to a specific function or application. Within the card is a microprocessor or micro controller chip that manages the memory allocation and access. This capability allows different and multiple functions and/or different applications to reside on the card, allowing businesses to issue and maintain a diversity of 'products' through the card.

Microprocessor Cards. Components of this type of architecture include a CPU, RAM, ROM, and EEPROM. The operating system is typically stored in ROM. The CPU uses RAM as its working memory. Application data is stored in EEPROM. A rule of thumb for smart card memory banks is that RAM requires four times as much space as EEPROM, which in turn requires four times as much space as ROM. Typical conventional smart card architectures have similar properties to that reflected in Table 2.1.

RAM	256 bytes to 1 Kbytes
EEPROM	1 Kbytes to 16 Kbytes
ROM	6 Kbytes to 24 Kbytes
Microprocessor	8 bits at approximately 5 MHz
Interface Speed	9600 BPS minimum, half duplex

Table 2.1 Components of Microprocessor Cards

2.4 Smart Card Architecture

Smart card architecture consists of a communication interface, memory, and a CPU for performing calculations and processing information. A smart card is pictured in Figure 2.1.

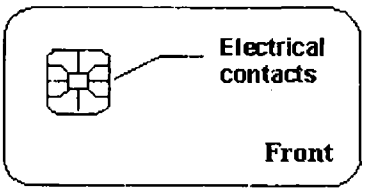

Figure 2.1 Smart Card

A smart card does not contain its own power supply, display, or keyboard. To interact with a Card Acceptance Device (CAD) it uses a communication interface, provided by a collection of eight electrical or optical contact points, as pictured in Figure 2.2.

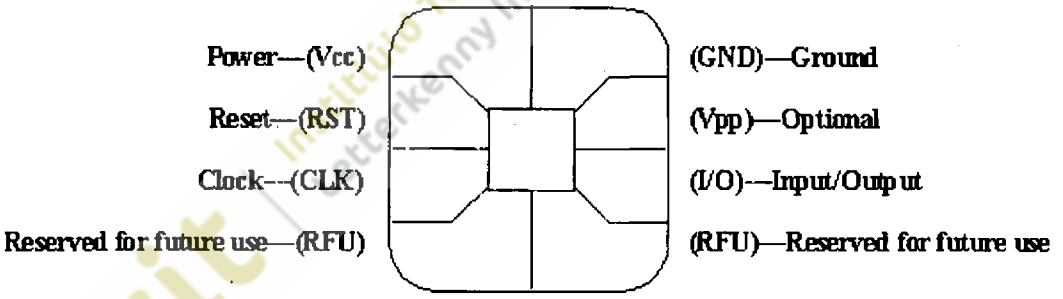

Figure 2.2 Eight Contact Points of the Smart Card Chip

CAD (also called a card reader, device reader, or card terminal) serves as a conduit for information into and out of the card. The card must be inserted into the CAD to provide the card with power (through its contacts, as described above).

2.5 Memory Allocations

ROM

The smallest memory element is read-only memory. This type of memory can be read by typical software, but it requires very special equipment in order to write information into the memory. The start of ROM memory is written during the manufacturing process. The advantage of this is that this technique tends to enhance the security of the chip, since it is difficult to examine the contents of the ROM without destroying the chip even with very expensive probing equipment including ultraviolet readers [59]. This type of memory is used for permanently encoded routines, but it is useless for storage of dynamic information such as application state variables that change during the normal use of the card.

EEPROM

Significantly larger memory allocation is required for the electrically erasable and programmable read-only memory. The contents of this type of memory in a smart card chip can actually be modified during normal use of the card. Programs or data can be stored in EEPROM during normal operation of the card and then read back by applications that are using the card. The electrical characteristics of EEPROM memory are such that it can only be erased and then reprogrammed a finite (but reasonably large) number of times, generally around 100,000 times. Data and program code can be written to and read from EEPROM under the control of the operating system. EEPROM is non-volatile memory. The information content of the memory is unchanged when the power to the memory is turned off. Information content is preserved across power-up and power-down cycles on the smart card chip.

RAM

Larger still is a memory type known as Random Access Memory. This is the type of memory used in typical computer systems, such as a desktop PC [59]. Information can be written and erased many times in this type of memory very quickly. In the smart card chip, however, a RAM memory cell is approximately four times larger than an EEPROM memory cell. RAM is volatile memory. The contents of the memory are lost when power is removed from the memory cell. Information in RAM is not preserved across a power-down and power-up cycle on a smart card. RAM is useful where application speed is essential. This can be extremely important when a smart card is interacting with a PC application in which the timing of responses from the card to the PC are important [7]; this is often the case in the mobile telecommunications area (that is, smart card-based cellular telephones).

Smart card chips tend to make use of varying amounts of each memory type, depending on the specific application for which the smart card is to be used. The most powerful chips used in smart cards today have RAM sizes in the 256-byte to 32-KB range, ROM sizes in the 16-KB to 128-KB range, and EEPROM sizes in the 1-KB to 256-KB range [7].

2.6 Operating System

The smart card operating system controls the basic relationship between a smart card terminal - master, such as personal computer, and the smart card itself - slave. The terminal sends a command to the smart card, the smart card executes the command, returns the result, if any, to the terminal and waits for another command.

The program modules are written as ROM code and this means that the operating system must be robust. From the perspective of operating system design, it is an unfortunate fact that the implementation of certain mechanisms is influenced by the hardware that is used [7]. Also the smart card operating system must be closely coupled to the hardware of the micro controller used.

2.7 File System

Smart card operating systems provide functional support for the usual set of file operations such as create, delete, read, write, and update. Associated with each file on smart card is an access control list. This list records what operations, if any, each card identity is authorized to perform on the file.

The file system supports a special root directory file ("master file"), optional sub-directory files ("dedicated files"), and data files ("elementary files"). The identifiers of all files along the path from the master file down to a specific file unambiguously identify the specific file. All three categories of files contain control information, such as a file identifier, file name, specifications of record or data lengths in the file, etc.

The draft standards specify various types of elementary file structures: a sequence of records of identical length, a sequence of records of variable length, a sequence of records with identical length organized as a ring, and a "transparent structure" that is seen at the interface as a sequence of data units. It is up to the developers to select the functionality they require and implement it.

2.8 Data Transmission

The requirement for two-way communications is a prerequisite for all interactions between a smart card and terminal. All communications to and from the smart card are carried out over the C7 (I/O) contact. Thus, only one party can communicate at a time, whether it is the card or the terminal. This is termed "half-duplex". Communication is always initiated by the terminal, which implies a type of client/server relationship between card and terminal.

After a card is inserted into a terminal, its contacts are first mechanically connected to those of the terminal. Then it is powered up by the terminal, executes a Power-on-Reset, and sends an Answer-to-Reset (ATR) to the terminal. The ATR is parsed, various parameters are extracted, and the terminal then submits the initial instruction to the card. The card generates a reply and sends it back to the terminal. The client/server relationship continues in this manner until processing is completed and the card is removed from the terminal.

The physical transmission layer is defined in ISO/IEC 7816-3. It defines the voltage level specifics which end up translating into the "0" and "1" bits at the software interface.

Communication between the card and reader proceeds according to various state transitions illustrated in Figure 2.3. The communication channel is single-threaded; once the reader sends a command to the smart card, it blocks until a response is received. A full-duplex procedure, in which both parties can send and receive simultaneously, is presently not implemented for Smart Cards [30].

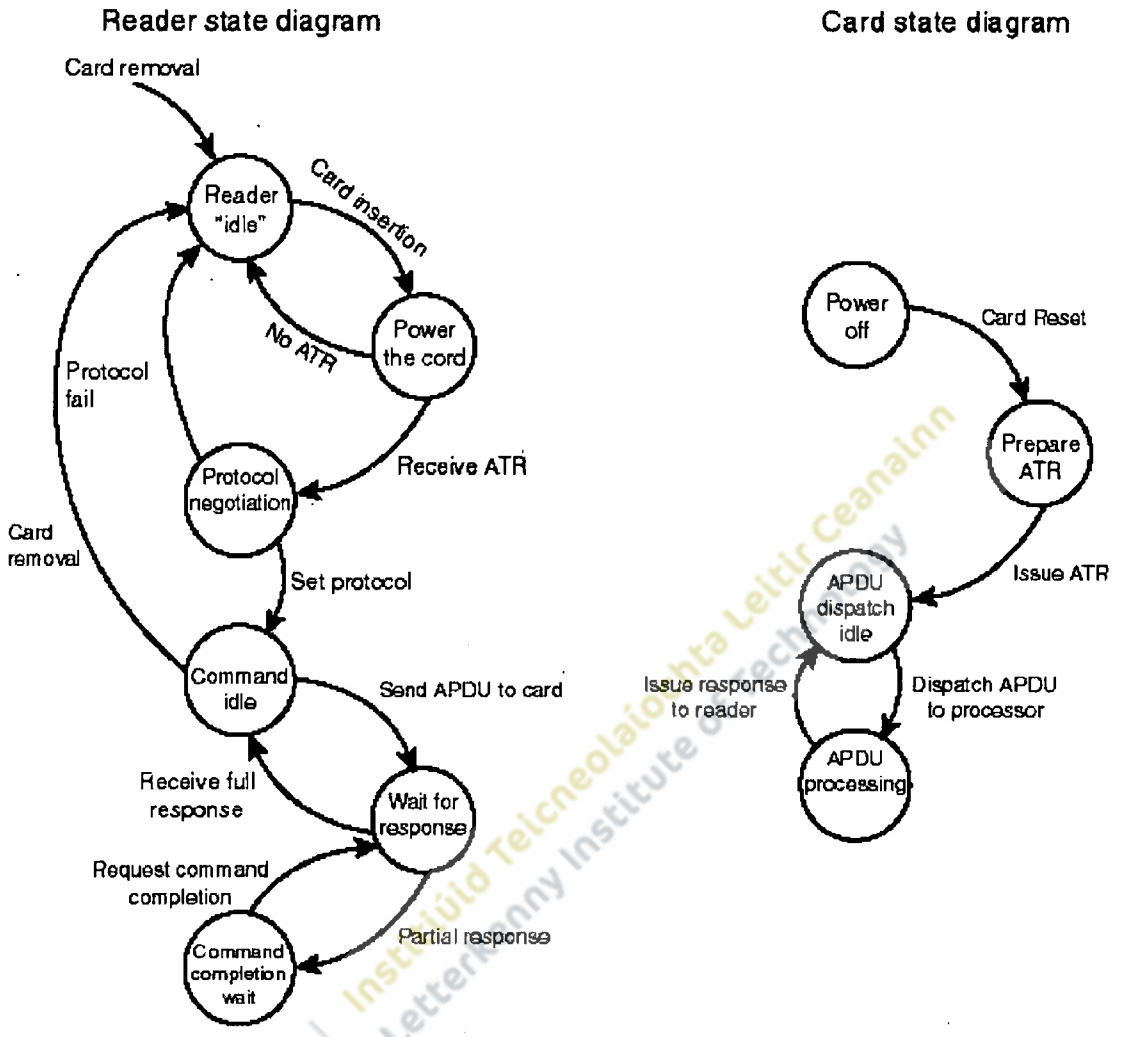

Figure 2.3 Data Communication between Smart Card and Reader

Logically, there are several different protocols for exchanging information in the client/server relationship. They are designated "T=" plus a number, and are summarized in Table 2.2.

PROTOCOL	DESCRIPTION
T = 0	Asynchronous, half-duplex, byte oriented, see ISO/IEC 7816-3
T = 1	Asynchronous, half-duplex, block oriented, see ISO/IEC 7816-3, Adm.1
T = 2	Asynchronous, full-duplex, block oriented, see ISO/IEC 10536-4

T = 3	Full duplex, not yet covered
T = 4	Asynchronous, half-duplex, byte oriented, (expansion of T = 0)
T = 5 TO T = 13	Reserved for future use
T = 14	For national functions, no ISO standard
T = 15	Reserved for future use

Table 2.2 Transmission Protocols [59]

The two protocols most commonly seen are T=0 and T=1, where T=0 is being the most popular [7]. A brief overview of the T=0 protocol is given below. More detailed information and descriptions of all the protocols can be found in [7].

Figure 2.4 Typical T=0 instructions

In the T=0 protocol, the terminal initiates communications by sending a 5 byte instruction header which includes a class byte (CLA), an instruction byte (INS), and three parameter bytes (P1, P2, and P3). A data section follows this optionally. Most commands are either incoming or outgoing from the card's perspective and the P3 byte specifies the length of the data that will be incoming or outgoing. Error checking is handled exclusively by a parity bit appended to each transmitted byte. If the card correctly receives the 5 bytes, it will return a one-byte acknowledgment equivalent to the received INS byte. If the terminal is sending more data (incoming command) it will send the number of bytes it specified in P3. Now the card has received the complete instruction and can process it and generate a response. All commands have a two-byte response code, SW1 and SW2, which reports success or an error condition. If a successful command must return additional bytes, the number of bytes is specified in the

SW2 byte. In this case, the GET RESPONSE command is used, which is itself a 5-byte instruction conforming to the protocol. In the GET RESPONSE instruction, P3 will be equal to the number of bytes specified in the previous SW2 byte. GET RESPONSE is an outgoing command from the card's point of view. The terminal and card communicate in this manner, using incoming or outgoing commands, until processing is complete.

2.9 Instruction Set

There are four international standards that define typical smart card instruction sets. More than 50 instructions and their corresponding execution parameters are defined. Though found in four separate standards, the instructions are largely compatible. The specifications are GSM 11.11 (prETS 300608), EN 726-3, ISO/IEC 7816-4 [68], and the preliminary CEN standard prEN 1546. Instructions can be classified by function as follows:

- File selection
- File reading and writing
- File searching
- File operations
- Identification
- Authentication
- Cryptographic functions
- File management

- Instructions for electronic purses or credit cards
- Operating system completion
- Hardware testing
- Special instructions for specific applications
- Transmission protocol support

Typically, a smart card will implement only a subset of the possible instructions, specific to its application. This is due to memory or cost limitations [57].

2.10 Smart Card Readers

Though commonly referred to as "smart card readers", all smart card enabled terminals, by definition, have the ability to read and write as long as the smart card supports it and the proper access conditions have been fulfilled. In contrast to smart cards, which all have very similar construction, smart card readers come in a variety of forms with varying levels of mechanical and logical sophistication. Electrically, the reader must conform to the ISO/IEC 7816-3 standard [59].

PHYSICAL CONNECTION	PROS	CONS
Serial Port	Very common, robust, inexpensive. Cross platform support for Windows, Mac, and Unix.	Many desktop computers have no free serial ports. Requires external power tap or battery.
PCMCIA	Excellent for traveling users with	Can be slightly more expensive.

	laptop computers	Many desktop systems don't have PCMCIA slots.
PS/2 Keyboard Port	Easy to install with a wedge adapter. Supports protected PIN path.	Slower communication speeds.
Floppy	Very easy to install	Requires a battery. Communications speed can be an issue.
USB	Very high data transfer speeds.	Not yet widely available. Shared bus could pose a security issue.
Built-in	No need for hardware or software installation.	Not yet widely available.

Table 2.3 Pros and Cons of Various Readers [59]

2.11 Security Related Standards

The following are emerging as important standards with respect to the integration of smart cards into computer and network security applications:

- PKCS#11: Cryptographic Token Interface Standard - This standard specifies an Application Programming Interface (API), called Cryptoki, for devices, which hold cryptographic information and perform cryptographic functions. Cryptoki follows a simple object-based approach, addressing the goals of technology independence (any kind of device) and resource sharing (multiple applications accessing multiple devices). PKCS#11 presents to applications a common, logical view of the device called a cryptographic token. The standard was created in 1994 by RSA with input from industry, academia, and government [69].

- PC/SC - The PC/SC Workgroup was formed in May 1997. It was created to address critical technical issues related to the integration of smart cards with the PC. PC/SC Workgroup members include Bull Personal Transaction Systems, Gemplus, Hewlett-Packard, IBM, Microsoft Corp., Schlumberger, Siemens-Nixdorf Inc., Sun Microsystems, Toshiba Corp., and VeriFone. The specification addresses limitations in existing standards that complicate integration of ICC devices with the PC and fail to adequately address interoperability, from a PC application perspective, between products from multiple vendors. It provides standardize interfaces to Interface Devices (IFDs) and the specification of common PC programming interfaces and control mechanisms. Version 1.0 was released in December of 1997 [70].

- OpenCard - Open Card is a standard framework announced by International Business Machines Corporation, Inc., Netscape, NCI, and Sun Microsystems Inc. that provides for interoperable smart card solutions across many hardware and software platforms. The Open Card Framework is an open standard providing architecture and a set of APIs that enable application developers and service providers to build and deploy smart card aware solutions in any Compliant-compliant environment. It was first announced in March 1997 [61].

- JavaCard - The JavaCard API is a specification that enables the Write Once, Run Anywhere™ capabilities of Java on smart cards and other devices with limited memory. The JavaCard API was developed in conjunction with leading members of the smart card industry and has been adopted by over 95% of the manufacturers in the smart card industry, including Bull/CP8, Dallas Semiconductor, De La Rue, Geisecke

& Devrient, Gemplus, Inside Technologies, Motorola, Oberthur, Schlumberger, and Toshiba [53, 54].

- Common Data Security Architecture - Developed by Intel, the Common Data Security Architecture (CDSA) provides an open, interoperable, extensible, and cross-platform software framework that makes computer platforms more secure for all applications including electronic commerce, communications, and digital content. The Open Group adopted the CDSA 2.0 specifications in December 1997 [55].
- Microsoft Cryptographic API - The Microsoft® Cryptographic API (CryptoAPI) provides services that enable application developers to add cryptography and certificate management functionality to their Win32® applications. Applications can use the functions in CryptoAPI without knowing anything about the underlying implementation, in much the same way that an application can use a graphics library without knowing anything about the particular graphics hardware configuration [72].

2.12 Attacking Smart cards

Usually, smart cards implement three levels of logical access control. The first is the association of a set of privileges with a user's password, and the ability to control access to files on the card based on those privileges. The second level of logical access control is the ability to detect and respond to a sequence of invalid access attempts. The third level is the "logical channel" - a logical link between the host system and a file on the smart card. When

logical channels are in use, the selection of a file associates the file and its security status with the logical channel encoded in a reserved field of the selection command header.

Logical channels provide a mechanism for allowing multiple, independent applications to use the storage capabilities of the card. The card interface software on the host system must manage the mapping between processes and logical channels; the channel numbers are either assigned by the external world or by the card itself.

Based on this information attacks on smart cards generally fall into four categories [30]:

- Logical attacks - Logical attacks occur when a smart card is operating under normal physical conditions, but sensitive information is gained by examining the bytes going to and from the smart card. There are logical countermeasures to this attack but not all smart card manufacturers have implemented them. This attack does require that the PIN to the card be known, so that many private key operations can be performed on chosen input bytes.
- Physical attacks - Physical attacks occur when normal physical conditions, such as temperature, clock frequency, voltage, etc, are altered in order to gain access to sensitive information on the smart card. Other physical attacks that have proven to be successful involve an intense physical fluctuation at the precise time and location where the PIN verification takes place. This type of attack can be combined with the logical attack mentioned above in order to gain knowledge of the private key. Most physical attacks require special equipment.

- Trojan Horse attacks - This attack involves a rogue, Trojan horse executable code application that has been planted on an unsuspecting user's workstation. The Trojan horse waits until the user submits a valid PIN from a trusted application, thus enabling usage of the private key, and then asks the smart card to digitally sign some rogue data. The countermeasure to prevent this attack is to use "single-access device driver" architecture. This prevents the attack but also lessens the convenience of the smart card because multiple applications cannot use the services of the card at the same time.

- Social Engineering attacks - In computer security systems, this type of attack is usually the most successful, especially when the security technology is properly implemented and configured. An example of a social engineering attack has a hacker impersonating a network service technician. The serviceman approaches a low-level employee and requests their password for network servicing purposes.

Any security system, including smart cards, is breakable. However, there is usually an estimate for the cost required to break the system, which should be much greater than the value of the data being protected by the system. Independent security labs test for common security attacks on leading smart cards, and can usually provide an estimate of the cost in equipment and expertise of breaking the smart card. When choosing smart card architecture, one can ask the manufacturer for references to independent labs that have done security testing. Using this information, designers can strive to ensure that the cost of breaking the system would be much greater than the value of any information obtained [60].

In 1998, Researchers at Cryptography Research, Inc., led by Paul Kocher, publicly announced a new set of attacks against smart cards called Differential Power Analysis (DPA). DPA can be carried out successfully against most smart cards currently in production [71].

DPA is a complicated attack that relies on statistical inferences drawn on power consumption data measured during smart card computation. The equipment required to perform DPA is simple: a modified smart card reader and some off-the-shelf PCs. The algorithm itself is quite complex, but details have been widely published.

Chips inside a smart card use different amounts of power to perform different operations. By hooking a card up to an oscilloscope, a pattern of power consumption can be measured. Particular computations create particular patterns of spikes in power consumption. Careful analysis of the peaks in a power consumption pattern can lead to the discovery of information about secret keys used during cryptographic computations. Sometimes the analysis is straightforward enough that a single transaction provides sufficient data to steal a key. More often, thousands of transactions are required. The types of sensitive information that can leak include PINs and private cryptographic keys. Figure 2.5 is a conceptual diagram of DPA.

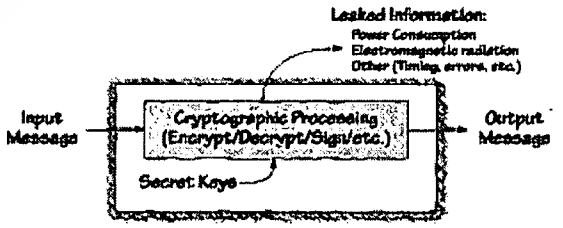

Figure 2.5 DPA Diagram

Possible solutions include masking power consumption with digital noise or throwing random calculations into the mix. Another potential solution is randomising the order of card

computations so that in the end, the same computation is performed using different patterns of primitives. All of these potential technological solutions are ways to mask the giveaway patterns in the power consumption of the card.

DPA is actually a variation on an earlier attack discovered by Kocher. The earlier attack exploited the fact that some operations require different amounts of time to finish, depending on which values they are computing. In the same way that DPA allows an attacker to piece together key information based on variations in power consumption, Kocher's timing attack allows an attacker to piece together a key based on variations in the amount of computing time required to encrypt various values [75].

One thing to note is that legitimate users of smart cards don't have to worry too much about DPA or timing attacks, because the attack requires physical access to the card itself. Unless you lose your card or insert it directly into an attacker's machine, there is not much threat that your card will be cracked. The main risk that DPA presents is to companies that must concern themselves with widespread fraud of the sort carried out by organized crime.

The best approach is to assume information will leak from a smart card and design systems in such a way that they remain secure even in the face of leaking information. An approach of this sort may preclude smart card systems designed to do all processing offline without a centralized clearinghouse [19].

2.13 Conclusion

Smart cards provide access control/vending systems, which interface to a range of host products such as photocopiers, printers, food, beverage and product vending machines, telecommunication equipment, cash registers and point of sale computer systems. The Smart Card may provide an easier way of identifying employees, tracking attendance, automating food service and controlling access for every employee in your company.

The benefits of using Smart Cards are measured in added security, accountability, and controlled costs.

Every card can be personalized with a photograph and is "forgery proof", your company logo can be displayed, attendance is recorded and compiled, access is controlled to any location or personal computer, all data for staff is securely stored and protected.

Smart cards have proven to be useful for transaction, authorization, and identification media. As their capabilities grow, they could become the ultimate thin client, eventually replacing all of the things we carry around in our wallets, including credit cards, licenses, cash, and even family photographs. By containing various identification certificates, smart cards could be used to voluntarily identify attributes of ourselves no matter where we are or to which computer network we are attached.

The current state of the art smart cards have sufficient cryptographic capabilities to support popular security applications and protocols [7, 30, 59].

Chapter 3. Java Card

Java Card is the new programming language for smart cards (compliant with the ISO 7816 standard) developed by Sun. It simplifies the programming of smart cards because of its object-oriented features. The aim of this chapter is to describe the Java Card environment and the formal semantics of the Java Card language.

3.1 Java Card Introduction

Today the market for embedded devices spans a wide variety of consumer and business products, including devices such as mobile phones, pagers, set-top boxes, process controllers, office printers, and network routers and switches. Typically, embedded devices have dedicated functionality - they are designed strictly for a specific set of tasks. Engineered for long life and high reliability, embedded devices incorporate low-speed microprocessors and may have a limited amount of memory.

To meet performance and size requirements, embedded device manufacturers will typically use a real-time operating system (RTOS) and custom, proprietary development tools, well suited for meeting devices' memory limitations. There are numerous different RTOS vendors that exist today, each with a proprietary operating environment and many with tightly integrated and specialized development tools [1].

Early environments for embedded devices were developed in assembler. As these devices matured, some manufacturers shifted to higher-level languages like C and C++. Although

using higher-level languages made it easier to find developers, the complexity of these languages continued to contribute to long schedules and high non-recurring engineering (NRE) costs [7]. In addition, customers were constantly demanding new functionality in devices. As manufacturers responded by adding capabilities, more memory and software complexity was required. This raised manufacturing costs as well as increased NRE costs [7]. Thus, embedded device manufacturers faced the constant challenge of managing increasing development costs.

To aggravate the problem, there were a greater number of target operating systems and processors, sometimes even within the same product families. Occasionally there were also new product categories and innovations, such as set-top boxes. Manufacturers faced intense competitive pressures, and were often required to put out more products in a shorter timeframe. As a result, manufacturers sought a more open, standards-based development environment - one that would lower costs and speed development [49].

Embedded device manufacturers have turned to the Java programming language to answer their needs. It has many advantages - simplicity, portability, security model, and an object-oriented nature.

- Portability. By using an underlying Java run-time environment, applications can be easily developed on a desktop system using standard software development tools. Hardware-specific code can be simulated on a desktop system, saving valuable development time. By taking into account the underlying target hardware characteristics, developers can then move applications with minimal effort to the specific target device.

- Software reuse. Because the Java language is object-oriented and platform-independent, developers can migrate commonly used software modules or entire applications between products and across product lines.
- Simplicity. The Java language is easy to learn and use, which shortens development cycles and lowers costs. Unlike C++, the Java language features automatic memory management and a single inheritance model. Most importantly, the lack of pointers eliminates a common source of memory leaks.
- Safety and security. The Java language provides a secure, isolated environment in which applications can execute safely.
- Longevity. Since the Java APIs have been developed with the involvement of many companies within the industry, the Java platform has gained a level of maturity that promises a long lifespan. This will simplify support and maintenance issues for device manufacturers concerned with long product life cycles.

While Java Card does solve some problems associated with writing smart card software, it introduces problems of its own. One of the main benefits of the Java is its syntax, which is the same in all variations of Java and makes it easier for a programmer to write code, as there no need to learn another syntax. However, familiar syntax does not mean ease of portability. It is easy to suffer from a misconception in thinking that Standard Java and Java Card are very similar. This can happen because of the same syntax, similar names and the large amount of marketing hype, but in fact, these two environments are relatively different from each other from the programming point of view [1].

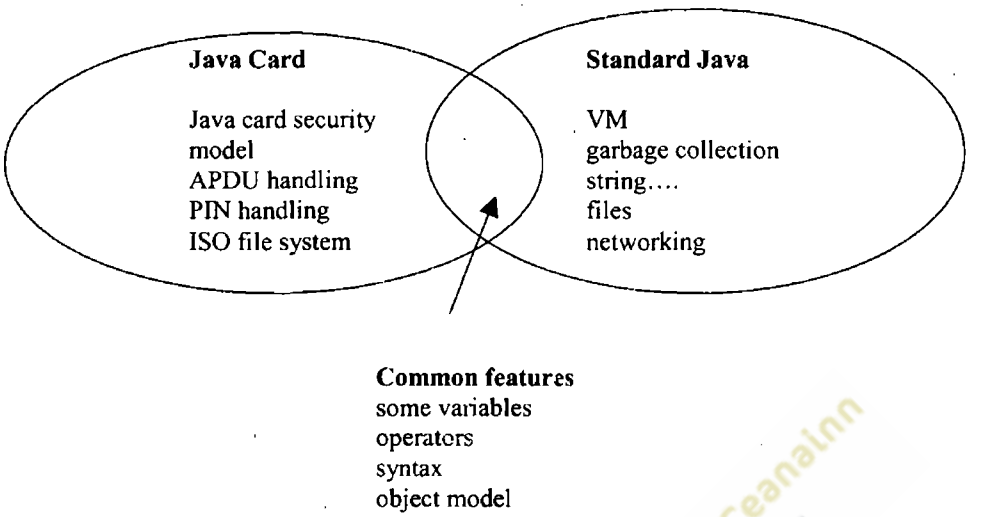

Figure 3.1 Common features between Java Card and Standard Java

In the following sections the differences between Java Card and standard Java are described in detail.

3.2 Java Card Overview

Java Card is a standard platform that aims to do for smart cards what Java has done for larger machines, primarily the promise of 'Write Once, Run Anywhere' [53].

One obstacle blocking widespread use of smart cards has been the large number of incompatible and often obscure development languages available for writing smart card applications. Regardless of the ISO 7816 specifications, programming languages for smart cards have traditionally amounted to special-purpose assembly languages. Few developers were familiar with card application languages, the upshot being that only a handful of people could develop smart card code [29]. As cards become computationally more powerful, new

application languages are being designed and put into use. One of the most interesting new systems is Java Card.

A Java Card is a smart card that is able to execute Java byte code, similar to the way Java-enabled browsers can. Because standard Java is far too big to fit on a smart card, a solution to this problem is to create a stripped-down Java - Java Card. It's based on a subset of the Java API plus some special-purpose card commands.

Besides providing developers with a more familiar development environment, Java Card also allows smart cards to have multiple applications on them. Most of the time existing smart card products have only one application per card. This application is automatically invoked when power is provided to the card or the card is otherwise reset. Java Card allows multiple applications, potentially written by different organizations, to exist on the same card.

These are the major benefits of Java Card as advanced by Sun:

- Cross-platform: JCRE (Java Card Runtime Environment) is alleged to be ported to all smart card processor types
- Interoperable: Java Card applets developed by one vendor will run on any card that conforms to the Java Card specifications.
- Java Card adheres to existing smart card standards.
- Supports all kinds of smart card applications.

3.3 Java Card Language subset

Because of its small memory footprint, the Java Card platform supports only a carefully chosen, customized subset of the features of the Java language [1, 77]. This subset includes features that are well suited for writing programs for smart cards and other small devices while preserving the object-oriented capabilities of the Java programming language.

Supported Java Features	Unsupported Java Features
Small primitive data types: Boolean, byte, short	Large primitive data types: long, double, float
One dimensional arrays	Characters and strings
Java packages, classes, interfaces, and extensions	Multidimensional arrays
Java object-oriented featured: inheritance, virtual methods, overloading and dynamic object creation, access scope, and binding rules	Dynamic class loading
	Security manager
	Garbage collection and finalization
	Threads
	Object serialization
	Object cloning

Table 3.1 Java Card language subset

3.4 Java Card Architecture Overview

Java Card technology provides architecture for open application development for smart cards, using the Java[tm] programming language. The technology can also be used to develop applications for other devices that have extremely small memories, such as: subscriber identity module (SIM) cards for wireless phones [58].

Java Card technology comprises a set of specifications for the following:

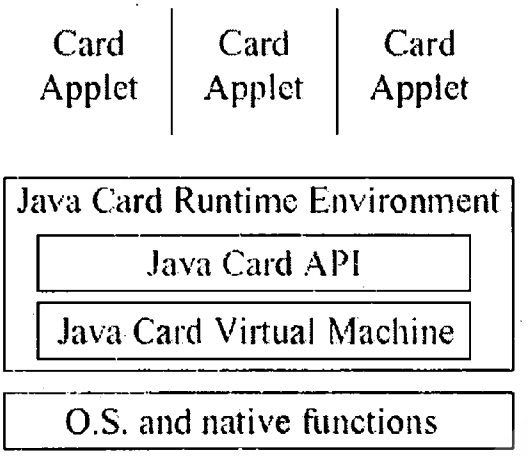

Figure 3.2 Java Card architecture

- Java Card API - an application programming interface, identifies the core Java Card class libraries.
- Java Card Virtual Machine - describes the characteristics of the virtual machine for handling Java Card applications.
- JCRE - Java Card Runtime Environment (JCRE) details runtime behaviour, such as how memory is managed or how security is enforced.

3.4.1 Java Card Runtime Environment

The JCRE consists of Java Card system components that run inside a smart card. The JCRE is responsible for card resource management, network communications, applet execution and on-card system and applet security.

The JCRE sits on top of the smart card hardware and native system. The JCRE consists of the JCVM (the bytecode interpreter), the Java Card application framework classes API, industry

specific extensions, and the JCRE system classes. The bottom layer of the JCRE contains the JCVM and native methods. The JCVM executes bytecode, controls memory allocation, manages objects, and enforces the runtime security.

The system classes act as the JCRE executive. They are analogues to an operating system core. The system classes are in charge of managing transactions, managing communications between the host application and Java Card applets, and controlling applet creations, selection and deselection. To complete tasks the system classes typically invoke native methods [53].

3.4.2 Java Card Virtual Machine

A primary difference between the Java Card Virtual Machine (JCVM) and Java Virtual Machine (JVM) is that JCVM is implemented as two separate pieces. The on-card option of the JCVM includes the Java Card bytecode interpreter. The Java Card converter runs on a PC or a workstation. The converter is the off-card piece of the virtual machine. Taken together, they implement all the virtual machine functions – loading Java class files and executing them with a particular set of semantics. The converter loads and preprocesses the class files that make up a Java package and outputs a CAP file.

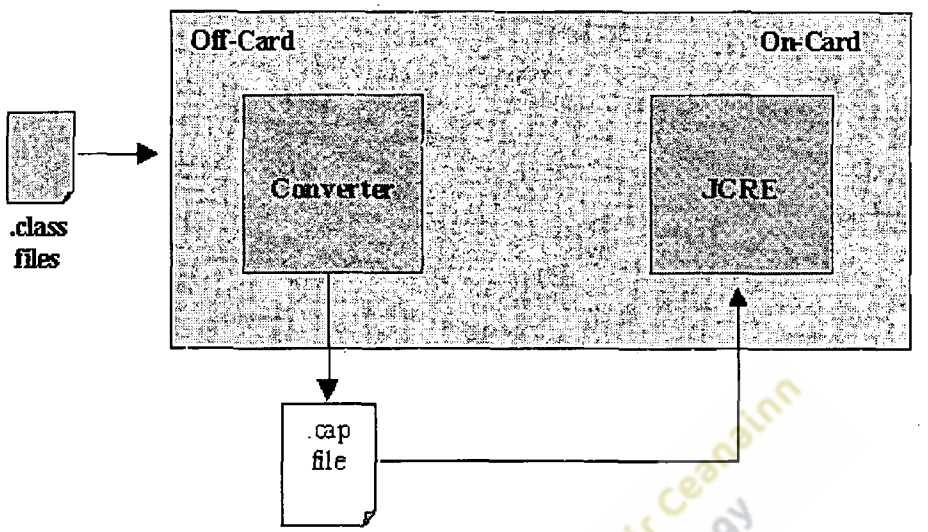

Figure 3.3 Java Card Virtual Machine

The JCVM provides bytecode execution and Java language support, including exception handling. The JCRE includes a VM and core classes to support APDU routing, ISO communication protocols, and transaction-based processing.

The off-card JCVM contains a Java Card Converter (JCC) tool for providing many of the verifications, preparations, optimisations, and resolutions that the JCVM performs at class-loading time. Dynamic class loading at runtime is not supported by the JCVM because:

- Dynamic class loading requires access to the storage location of the class file (refers to the disk or Internet), which is unavailable within a smart card environment
- Security aspects of the smart card environment prohibit most dynamic behaviour (object dynamic binding is allowed)
- There are limited resources within the smart card environment

The JCC tool is an "early-binding" implementation of the JVM. Every class referenced directly or indirectly by an applet must be bound into the applet's binary image when the applet is installed on the card. The JCC acts as an early-binding post-processor on the Java platform class files. The JCC performs the following steps:

- Verification - checks that the load images of the classes are well formed, with proper symbol tables and checks for language violations, specific to the Java Card specifications
- Preparation - allocates the storage for and creates the VM data structures to represent the classes, creates static fields and methods, and initialises static variables to default values
- Resolution - replaces symbolic references to methods or variables with direct references, where possible

Performing these three steps in the JCC, before an applet is installed on the card, allows the on-card JCVM to be more compact and efficient [1, 77].

Once an applet is installed on a Java Card-based smart card, it is considered loaded and ready to run (although some initialisations and personalisations of the applet may be required). The JCRE then performs additional load-time initialisation, which involves setting static constant initialises and initialising parameters declared with default values in interfaces. Although the JCC performs as much early binding and resolution as possible, some late binding is also supported by the JCRE [1].

3.4.3 Java Card Installer and Off-Card Installation Program

The following tools require installing Java Card application on Java Card:

- A Converter tool to convert a Java Card applet into a format required for installation.
- Off-card verification tools to check the integrity of files produced by the Converter.
- An off-card installer to install a Java Card applet onto a smart card.

The Java Card applets are developed using these classes and tools on workstations or PCs. Specifically it allows the developer to:

- Compile the applet.
- Optionally, test the applet in the JCWDE, and debug the applet. The JCWDE, which runs on workstations or PCs, simulates the Java Card runtime environment on a Java virtual machine. It's not a complete simulation, for example, the JCWDE does not simulate the applet firewall of a JCVM. However the JCWDE does provide a simulation that allows a good initial test of a Java Card applet. It allows running an applet as if it was masked in the read-only memory of a smart card. And importantly, it allows running the test in a workstation or PC, without having to convert the applet, generate a mask file, or installing the applet.
- Convert the applet and all the classes for installation to a CAP file, and possibly an export file. An export file is used to convert another package if that package imports classes from this package. Unlike the JVM, which processes one class at a time, the conversion unit of the converter is a package. The Java compiler produces class files

from source code, and then a converter preprocesses all the class files that make up a Java package and converts the package to a CAP file.

In Java Card technology, a Java Card applet does not directly incorporate into a mask. Similarly, after a smart card is manufactured, a Java Card applet does not directly download for installation onto a smart card. Instead, for masking, an applet class and all the classes in its package convert to a JCA file [53].

The JCA file and JCA files for any other packages to be included in the mask are then converted into a format compatible with the target runtime environment. It's this converted output for the target runtime environment that is incorporated into the mask. Both a JCA file and a CAP file are self-descriptive files. These files contain information about the converted package.

As mentioned earlier, a Java Card applet does not install onto a smart card, instead its CAP file is installed. The off-card installer produces a script file that contains command APDUs that identify the beginning and end of the CAP file, its components, and component data. The script file is used as input to the APDUTool Utility. The APDUTool Utility submits command APDUs to a Java Card runtime environment, or to a simulated runtime environment such as the JCWDE. After the script file is tailored, the APDUTool utility is run, specifying the script file as input. The APDUTool starts the on-card installer, which downloads the CAP file. If requested in the script file, the on-card installer creates the applets that are defined in the CAP file, so that the applets are available in the Java Card runtime environment.

An APDU protocol sends APDU commands to the JCWDE or to a Java Card runtime environment. Command APDUs are the way operational requests are made to a smart card.

The APDU class in the Java Card APIs provides a powerful and flexible interface for handling APDUs whose command and response structures conform to the ISO 7816-4 specification [68].

APDU commands are always sets of pairs (see Figure 3.4). Each pair contains a *command APDU,* which specifies a command, and a *response APDU,* which sends back the execution result of the command. In the card world, smart cards are *reactive* communicators - that is, they never initiate communications, they only respond to APDUs from the outside world. 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. The currently selected applet processes the command and returns a response APDU to the terminal application. *Command* APDUs and *response* APDUs are exchanged alternately between a card and a CAD [53].

Command APDU						
Mandatory header				Optional body		
CLA (1 byte)	INS (1 byte)	P1(1 byte)	P2(1 byte)	Lc (1 byte)	Data (bytes = Lc)	Le (1 byte)

Table 3.2 APDU command description for the applet

- CLA: Class of instruction. Indicates the structure and format for a category of command and response APDUs
- INS: Instruction code. Specifies the instruction of the command
- P1 (1 byte) and P2 (1 byte): Instruction parameters. Provide further qualifications to the instruction
- Lc (1 byte): Number of bytes present in the data field of the command.

- Data field (bytes equal to the value of Lc): A sequence of bytes in the data field of the command.
- Le (1 byte): Maximum of bytes expected in the data field of the response to the command.

Response APDU		
Optional body	Mandatory trailer	
Data field (variable length)	SW1 (1 byte)	SW2 (1 byte)

Table 3.3 APDU response description for the applet

- Data field: A sequence of bytes received in the data field of the response
- SW1 and SW2: Status words. Denote the processing state in the card

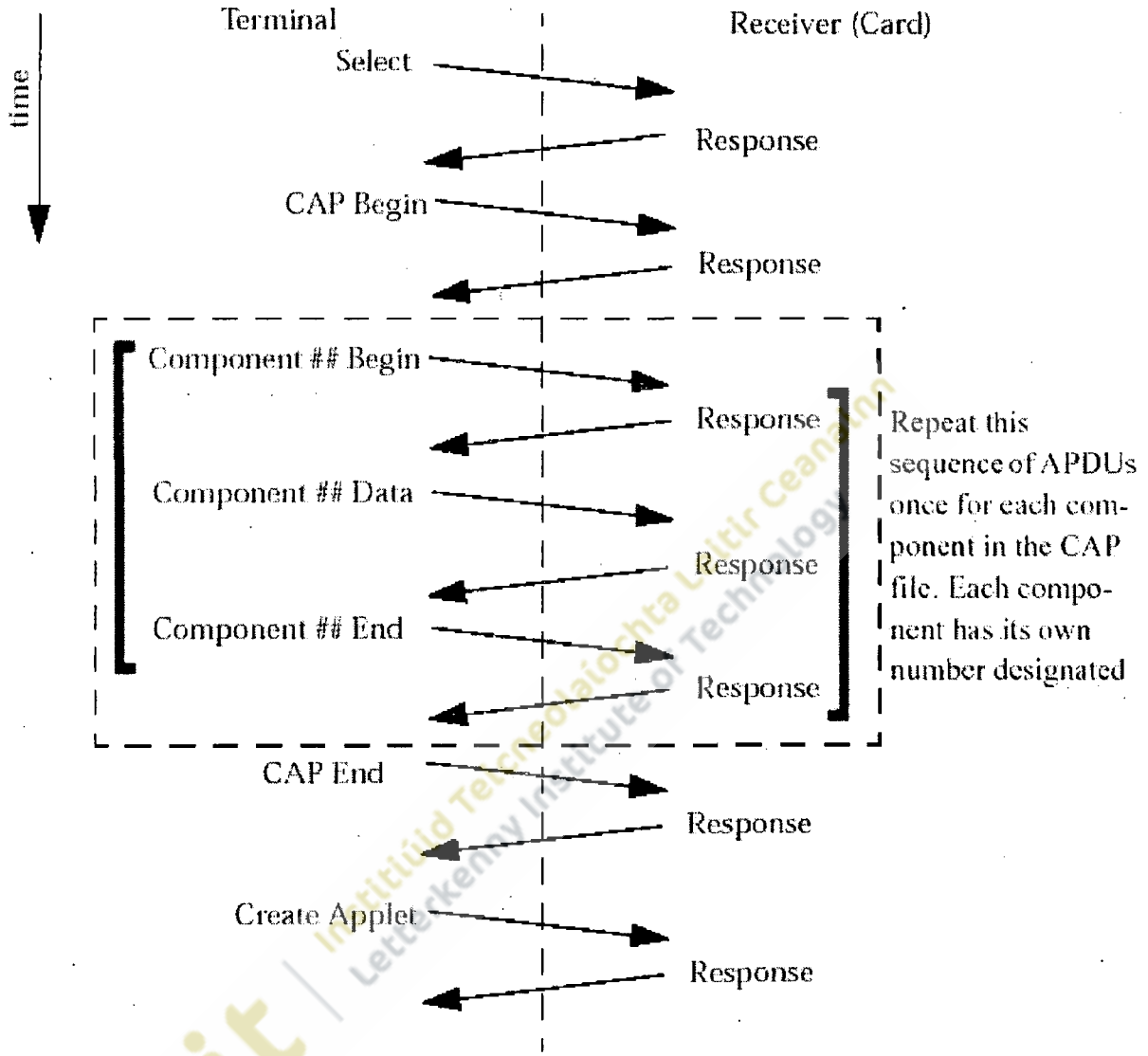

Figure 3.4 Installer APDU Transmission Sequence

APDUs are transmitted between the host and the card by the lower-level transport protocol. Any application must be aware of the transport protocol employed by the underlying system. In the Java Card platform applets can be written so that they will work correctly regardless of whether the platform is using the T=0 or the T=1 protocol.

3.4.4 Java Card APIs

Since major Java Card applications are likely to involve multiple card issuers [49], interoperability must be designed into the system right from the start. From a technical perspective, the key is a Java Card API. This is a layer of software that allows an application to communicate with smart cards and readers from more than one manufacturer.

The API can be thought of as a specialised device, software, in this instance that acts as a translation layer between an application and the card. The API allows the agency running the application to select smart cards from multiple vendors. Opening an application to multiple Java Cards encourages competition among card vendors and the benefits of that competition—better quality and lower prices.

An API is not a universal interface that will work with all Java Cards; rather, it provides a way for applications to send commands to the specific chip operating system (COS) of more than one card [78]. Programmers can begin by developing an API for two or three cards and, over time, expand the software to include a dozen or more Java Cards. There is a practical limit on the size of the software program that can be stored in some portable terminals, but the API should be sufficiently versatile to accommodate cards from competing vendors.

The API can also be used to control data versions. If changes need to be made in the data elements on the card after it has been issued, the API can be utilized to update cards without having to recall the cards for reformatting.

3.5 Package and Applet Naming Convention

Most familiar applications are named and identified by a string name. In Java Card technology, however, each applet is identified and selected using an "application identifier" (AID). Also, each Java package is assigned an AID. This is because a package, when loaded on a card, is linked with other packages, which have already been placed on the card via their AIDs. This naming convention is in conformance with the smart card specification as defined in ISO 7816 [68].

An AID is a sequence of bytes between 5 and 16 bytes in length. Its format is depicted in

Application identifier (AID)	
National registered application provider (RID)	Proprietary application identifier extension (PIX)
5 bytes	0 to 11 bytes

Table 3.4. AID structure

ISO controls the assignment of RIDs to companies, with each company obtaining its own unique RID from the ISO. Companies manage assignment of PIXs for AIDs.

The package AID and the applet AID have the same RID value; their PIX values differ at the last bit.

3.6 Applet Installation

Applet installation occurs at the factory or at the office of the issuer and may also occur post-issuance, through a secure installation process (if one is defined by the card manufacturer) [1]. This process involves downloading a digitally signed applet, which the JCRE verifies as legitimate, before installing the applet. Applets that are installed through downloads cannot contain native method calls since they are not trusted.

Applets with native method calls must be installed at the factory or another trusted environment. This is done for security reasons, since native calls bypass the Java technology security framework and so must be highly trusted before being allowed on the card [51].

Once installed, Java Card platform classes do not interact directly with the card accepting device or off-card applications. Installed classes may interact directly with only the JCRE or with other installed classes. The JCRE selects an applet and then passes APDUs to the selected applets. In essence, the JCRE shields the developer from the smart card CPU, the CAD, and the particular ISO communication protocol employed. The JCRE also translates uncaught exceptions thrown by classes or normal return statements in applet methods into standard ISO return values.

The storage for an installed applet cannot be reclaimed; if a newer version of the applet is installed, it occupies a new storage location and the earlier version of the applet becomes unreachable. The Java Card applet can also be made unreachable by removing its reference from the JCRE applet registry table. Once the reference is removed, the applet can no longer be reached.

Installing the Java Card applet causes its static members to be initialised. Java Card technology supports constant static initialisers. The initialiser cannot execute Java software code, nor can it set the static member to a non-constant (variable) value. Installation also results in a call to the applet's *install()* method (unlike Java applets) [1, 2].

Applications running in a Java smart card communicate with host applications at the CAD side by using the APDU. For each command APDU, the applet first decodes the value of each field in the header. Knowing how to interpret the command and read the data, the applet can then execute the function requested by the command. For the response APDU, the applet should define a set of status words to indicate the result of processing the corresponding command APDU. During normal processing, the applet returns the success status word. If an error occurs, the applet must return a status word other than success to denote its internal state or diagnosis of the error.

Applet installation refers to the process of loading applet classes in a CAP file, combining them with the execution state of the Java Card runtime environment, and creating an applet instance to bring the applet into a selectable and execution state. On the Java Card platform, the loading and installable unit is a CAP file. A CAP file consists of classes that make up a Java package. A minimal applet is a Java package with a single class derived from the class *javacard.framework.Applet*. A more complex applet with a number of classes can be organized into one Java package or a set of Java packages.

To load an applet, the off-card installer takes the CAP file and transforms it into a sequence of APDU commands, which carry the CAP file content. By exchanging APDU commands with the off-card installation program, the on-card installer writes the CAP file content into the

card's persistent memory and links the classes in the CAP file with other classes that reside on the card. The installer also creates and initialises any data that is used internally by the JCRE to support the applet. If the applet requires several packages to run, each CAP file is loaded on the card.

As the last step during applet installation, the installer creates an applet instance and registers the instance with JCRE. To do so, the installer invokes the install method:

public static void install (byte[] bArray, short offset, byte length)

The install method is an applet entry point method, similar to the main method in a Java application. An applet must implement the install method. In the install method, it calls the applet's constructor to create and initialise an applet instance. The installation parameters are sent to the card along with the CAP file.

After the applet is initialised and registered with the JCRE, it can be selected and run. The JCRE identifies a running applet (an applet instance), using an AID. The applet can register itself with JCRE by using the default AID found in the CAP file, or it can choose a different one. The installation parameters can be used to supply an alternative AID.

The JCRE is a single-thread environment. This means that only one applet is running at a time. When an applet is first installed, it is in an inactive state. The applet becomes active when it is explicitly selected by a host application [53].

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 [1, 7].

3.7 Optimising Java Card Applets

One of the major factors influencing the design and features of Java Card applets is the limited availability of program and data memory in the smart card environment.

The Java Card platform accommodates environments in which only 512 bytes of RAM are available. The JCRE (including the Java Card VM and the system heap) must be contained within the available ROM and the Java Card applets and class libraries need to be stored within the available EEPROM space on the device.

To optimise memory usage, the following restrictions apply when creating Java Card applets:

- A maximum of 127 instance methods in any class (including inherited methods)
- A maximum of 255 bytes of instance data
- Object space is allocated from EEPROM

3.7.1 Reusing Objects

Most of the Java Cards do not include garbage collection. Because of that, applets should not instantiate objects using *new*. The rule is that, a single instantiation of an object should be "recycled" repeatedly, with each new use "customising" the member variables of the object instance.

In Java technology, an instance of an object is created as needed, its instance variables are set, and then the object is discarded (typically by going out of scope). In Java Card technology objects are not allowed to go out of scope; they will become unreachable, but the storage

space they occupy will never be reclaimed. Objects should remain in scope for the life of the applet and should be reused by writing new values to their member variables. This does not require all objects to be declared as static.

3.7.2 Allocating Memory

Memory for primitive types and arrays is allocated at object-declaration time. Memory for class-member variables is allocated from the system heap, and cannot be reclaimed (unless the smart card implements a garbage collector). Any memory allocated by new is taken from the heap. Memory for method variables, locals, and parameters is allocated from the stack and is reclaimed when the method returns.

The *install()* method is called only once, when the applet is installed on the card, so that a *new* in *install()* results in only a single instance of the object for the lifetime of the applet.

3.7.3 Accessing Array Elements

When accessing an array element, bytecodes are generated to fulfil the array-access instruction. To optimise memory usage, if the same element of an array is accessed multiple times from different locations in the same method, the array value is saved to a variable on the first access and then access the variable in subsequent accesses. Using the array value as a variable in this way creates more compact bytecodes than re-accessing the array.

3.8 Conclusion

Java Card technology preserves many of the benefits of the Java programming language - productivity, security, robustness, tools, and portability - while enabling Java technology for use on smart cards. The Virtual Machine (VM), the language definition, and the core packages have been made more compact and succinct to bring Java technology to the resource - constrained environment of smart cards.

The Application Programming Interface (API) for the Java Card technology defines the calling conventions by which an applet accesses the Java Card Runtime Environment and native services. The Java Card API allows applications written for one Java Card-enabled platform to run on any other Java Card-enabled platform.

The Java Card API is compatible with formal international standards, such as, ISO7816, and industry-specific standards, such as, Europay/Master Card/Visa (EMV) [64].

Chapter 4. Encryption and Digital Signatures

Today a secure computing environment is not completed without encryption technology. Encryption is the process of encoding data to prevent unauthorised parties from viewing or modifying it. The aim of this chapter is to give an overview of encryption and look in details of the way it used in smart card environment.

4.1 Introduction to Encryption

With the need for information security in today's digital systems both acute and growing, cryptography has become one of their critical components [40]. Cryptography services are required across a variety of platforms in a wide range of applications such as secure access to private networks, stored values, electronic commerce, and health care. Incorporating these services into solutions presents an ongoing challenge to manufacturers, system integrators, and service providers because applications must meet the market requirements of mobility, performance, convenience, and cost containment. The following are just few of the main cryptographic services:

- Authentication. A merchant must know the identity of the customer. For some kinds of businesses it is not sufficient that the customer authenticates themselves by the use of a password. In these cases an electronic version of today's identity or credit card is required. The recipient of a message or an order should know the identity of the sender and should also be sure that the data wasn't altered during its transmission. These

challenges, the authentication of the user and the integrity of the sent messages, can be met using various cryptographic methods.

- Non-repudiation. It is often necessary to assert that a particular person sent an order or message and that no other person could possibly have sent it. In traditional business the personal hand-written signature is used to assert this, combined with a witness of the signing act in cases of "great" importance. In electronic commerce, this challenge is met using digital signatures, based on public key cryptography.
- Privacy. The exchange of data between the merchant and the customer in most cases should be kept secret. No unauthorized party should be able to read or copy such a communication. This challenge, confidentiality, is met using encryption.

The two main types of cryptography are private-key and public-key cryptography. Both are based on complex mathematical algorithms and are controlled by keys.

4.2 Private Key Cryptosystems

The private key cryptosystem can be defined as follows [6]. Let M denote the set of all possible plaintext messages, C the set of all possible cipher text messages (encrypted messages), and K the set of all possible keys. A private key cryptosystem consists of a family of pairs of functions $e_k : M \rightarrow C$, $d_k : C \rightarrow M$, $k \in K$, such that

$$d_k(e_k(m)) = m \text{ for all } m \in M \text{ and } k \in K.$$

To use such a system, two parties initially agree upon a secret key $k \in K$. If at a later time one party, name it Alice, wishes to send another party, name it Bob, a message $m \in M$, then Alice sends the cipher text $c = ek(m)$ to Bob, from which Bob can recover m by applying the decryption function d_k.

Some desirable properties of a private key cryptosystem are that the functions e_k and d_k should be easy to apply, and that it should be infeasible for an eavesdropper who sees c to determine the message m (or the key k). The latter property should hold even if the opponent knows everything about the cryptosystem being used (except, of course, the particular key chosen).

Although private key cryptography is adequate for many applications, it has the following disadvantages, which make it unsuitable for use in certain applications:

- Key distribution problem. As described above, the two users have to select a key in secret before they can start communications over an insecure channel. A secure channel for selecting a key may not be available.
- Key management problem. In a network of n users, every pair of users must share a secret key, for a total of $\frac{n(n-1)}{2}$ keys. If n is large, then the number of keys becomes unmanageable.
- No signatures possible. A digital signature is an electronic analogue of a hand-written signature. This means that a digital signature allows the receiver of a message to convince any third party that the message in fact originated from the sender. In a private key cryptosystem, Alice and Bob have the same capabilities for encryption and

decryption, and thus Bob cannot convince a third party that a message he received from Alice in fact originated from Alice.

4.3 Public Key Cryptosystems

The basic idea that led to public key algorithms was that keys could come in pairs of an encryption and decryption key and that it should be impossible to compute one key given the other. This concept was invented by Whitfield Diffie and Martin Hellman in 1976 and independently by Ralph Merkle in 1978 [9].

Since then, many public key algorithms have been proposed, most of them insecure or impractical. All public key algorithms are very slow compared to private key algorithms [4]. The RSA algorithm takes about 1000 times longer than the popular private key encryption algorithm, DES, when implemented in hardware, and 100 times longer in software to encrypt the same amount of data.

However, public key algorithms have a big advantage when used for ensuring privacy of communication [4, 50]: Public key algorithms can be used for signing and decryption, and for encryption and signature verification. The private key may only be known to its owner and must be kept secret. It may be used for generation of digital signatures or for decrypting private information encrypted with the public key. The public key may be used for verifying digital signatures or for encrypting information. It does not need not to be kept secret, because it is infeasible to compute the private key from a given public key. Thus, users can post their public key to a directory, where everybody who wants to send an encrypted message or verify

a signature can look it up. Each entity in the network only needs to store its own private key and a public directory can store the public keys of all entities, which is practical even in large networks.

To construct a public key cryptosystem, we need a family $f_k : M \rightarrow C;\ k \in K$, of trapdoor one-way functions (TOF) [6]. The family should have the property that for each $k \in K$, the trapdoor, denoted $t(k)$, is easy to obtain. In addition, for each $k \in K$, it must be possible to describe an efficient algorithm for computing f_k, such that it is infeasible to recover k (and thus $t(k)$) from this description.

A one-way function $f : M \rightarrow C$ is an invertible function, such that for each $m \in M$ it is easy to compute $f(m)$, while for most $c \in C$ it is hard to compute $f^{-1}(c)$ [6].

The term "hard" will usually mean computationally infeasible, i.e. infeasible using the best-known algorithms and best available computer technology. At present, it is not known whether one-way functions truly exist, although there are several candidate one-way functions.

A one-way function $f: M \rightarrow C$ is said to be a trapdoor one-way function (TOF) if there is some extra information with which f can be efficiently inverted [6, 9]. This extra information is called the trapdoor.

Given such a family of TOFs, each user selects a random $a \in K$ and publishes the algorithm E_a for computing f_a in a public directory. E_a is the Alice's public key, while the trapdoor $t(a)$, which is used to invert f_a, is Alice's private key. To send a message $m \in M$ to Alice, user Bob simply looks up Alice's public key E_a in the directory and transmits $f_a(m)$ to Alice. Since Alice is the only person who has the ability to invert f_a, only Alice can recover the message m. There

is no need to exchange keys in secret prior to communicating and there is only one key pair associated with each user. Public key cryptosystems thus overcome the key distribution and management problems inherent with private key systems [6].

Since the invention of public-key cryptography, numerous public-key cryptographic systems have been proposed [9,]. Each of these systems relies on a difficult mathematical problem for its security. Today, three types of systems, classified according to the mathematical problem on which they are based, are generally considered both secure and efficient. The systems are:

- The Integer Factorisation System (RSA) - RSA cryptosystem based on the hard mathematical problem of integer factorisation, i.e. given a number that is a product of two large prime numbers, factorise the number to find the primes. The security of RSA is thought to be equivalent to the difficulty of factorising the modulus, n [40]. The size of an RSA key is usually measured in terms of the number of bits in the modulus. In general, the larger the key the higher the security level. RSA is regarded as highly secure algorithm and if the method's parameters chosen carefully the feasible way to attack it is to perform a 'brute-force' attack on the modulus.[69]
- The Discrete Logarithm Systems (DSA) - The DSA was proposed in August 1991 by the U.S. National Institute of Standards and Technology (NIST) and became a U.S. Federal Information Processing Standard (FIPS 186) in 1993. It was the first digital signature scheme accepted as legally binding by a government. The algorithm is a variant of the ElGamal signature scheme. It exploits small subgroups in Z^*_p in order to decrease the size of signatures. The security of the DSA relies on two distinct but related discrete logarithm problems. One is the discrete logarithm problem in Z^*_p where

the number field sieve algorithm applies. The second discrete logarithm problem works to the base g: given p, q, g, and y, find x such that $y \equiv g^x \pmod{p}$.[69]

- The Elliptic Curve Cryptography Systems (ECC) - Elliptic curve cryptography is emerging as a viable security method for use in certain constrained environments such as smart cards, pagers, cell phones and PDA's where memory, processing power or communications bandwidth may be limited. ECC is well suited to these applications because it requires a shorter key size than other cryptographic methods to achieve equivalent security against currently known attacks, and can therefore be implemented more efficiently [41].

ECC cryptosystems have experienced only fragmented deployment to date, due to the many incompatible representations, and lack of memory-efficient way to convert between the two popular basis types [45]. ECC delivers the highest strength per bit of any known public-key system because of the difficulty of the problem upon which it is based. The greater difficulty of the "hard" problem – the elliptic curve discrete logarithm problem (ECDLP) – means that smaller key sizes yield equivalent levels of security [45]. Table 4.1 shows the differences between key-size of different cryptosystems.

Time to break in MIPS years	RSA/DSA key size	ECC key size	RSA/ECC key size ratio
10^4	512	106	5:1
10^8	768	132	6:1
10^{11}	1.024	160	7:1
10^{20}	2.048	210	10:1
10^{78}	21.000	600	35:1

Table 4.1 Key sizes of different cryptosystems

The difficulty of the problem and the resulting equivalent-strength key sizes add several direct benefits to smart card implementations.

The main security issue is that the true difficulty of the ECC is not fully understood. Recent research has shown that that some elliptic curves that were believed suitable for elliptic curve cryptography are, in fact, not appropriate [45].

4.4 Digital Signatures

If goods or services are ordered, a contract on paper has to be signed testify that the order is placed and is liable for payment. If the same deal is made over a network instead, the electronic equivalent of signing on paper: a digital signature is used. Such a digital signature must guarantee that a person cannot repudiate their order or statement.

The different methods for digital signing are based on public key cryptography [4]. The signing person has a private key, which cannot be accessed or used by anyone else. A second key is known to the public and is associated with the private key.

Only the unique owner of the private key can sign an order or statement, while anybody can check the signature using the corresponding public key. With a conventional signature, a signature is physically part of the document being signed. However, a digital signature cannot be physically attached to the message that is signed, so the algorithm that is used must somehow bind the signature to the message [9].

To bind the digital signature to the message, we need to assume that $M = C$. If Alice wishes to send Bob a signed message m, she simply sends Bob the quantity $s = f_a^{-1}(m)$ together with m. Now, anyone can verify that $m = f_a(s)$ by using Alice's public key E_a, but only Alice could have computed s. Hence the quantity s serves as Alice's signature for the message m [6]. To keep the size of a signature relatively small, a one-way hash function is usually used to create a hash m' from a original message m, and sign m' instead of m.

For digital signatures it is crucial that the private key remains absolutely private [23]. If any person could copy another person's private key, the digital signature would no longer be unique to the owner. Therefore the private key has to be stored in a very secure place where nobody could possibly copy it and where nobody but the owner can use it.

Another fundamental difference between conventional and digital signatures is that a copy of a signed digital message is identical to the original. A conventional signature is verified by comparing it to other, authentic signatures. On the other hand, a copy of a signed paper document can usually be distinguished from an original [9].

4.5 Smart Cards and Cryptography

Smart card software security is based on cryptography. Keys are stored in files on the card and algorithms and protocols are implemented in software on the card. Cryptography is used primarily to authenticate system entities, such as users, cards, and terminals, and to encrypt communications between the smart card and the outside world. The cryptographic functions built into a smart card for its own security requirements may also be used to implement security functionality in other systems [30].

Encryption can be applied to all message traffic to and from the smart card or only to particular messages. If a smart card is communicating with two applications simultaneously, it may be using a different encryption key or technique with each.

Smart card programmers typically do not have to design new authentication or encryption algorithms. Rather, they use the facilities that are built into the smart card. These facilities have been field tested and come with a certain level of assurance of correctness. Designing new algorithms is not easy, and validating the correctness of a new algorithm is probably not a subtask that a smart card application developer wants to assume [24]. Table 4.2 lists a number of cryptographic algorithms, which find use in various smart cards [60].

Algorithm	*Sample Uses*
DES	Communication channels
A3 and A8	GSM mobile telephone
Elliptic curve	Digital signature
TSA7	Health records
RSA	Digital signature

Table 4.2 Cryptographic algorithms used on Smart Card

Without smart cards, distributed computer environment (DCE) uses one-factor authentication: users authenticate by proving they know a secret that is shared with the DCE Security Service, i.e., the user's password. Passwords are known not to be very secure, as they can be lost, stolen, shared, or guessed fairly easily.

By using smart cards to store each user's long-term DCE key, the cards introduce a second authentication factor: users now must not only prove they know a secret (the password used to gain access to the card), but they must also prove they have physical possession of the smart card (by using the password and successfully retrieving the long-term key).

The use of two-factor authentication dramatically improves security [30]. Cards limit vulnerability to sharing, since the card can be in the physical possession of only one individual at a time. They also effectively prevent vulnerability to guessing, since the long-term key stored on the card is a 56-bit random number rather than a password. Cards can be lost or stolen, but without the accompanying card-access password, they are not usable. Should the card be lost or stolen, the owner is highly motivated to report the loss promptly, as the owner will be unable to access the computer system without the card.

The second potential value that smart cards present to DCE is their secure storage capability [30]. This is utilized in two-factor authentication for storing a user's long-term key. It would also be feasible to decrypt and store DCE credentials on the card, hiding them from the host system and from attackers on the host. DCE applications could also make use of secure storage on the card for application-specific information.

The third potential value to DCE is the encryption and key generation capabilities that smart cards have. Licensing agreements for public key technology are typically much less restrictive when the technology is confined to a physical device. By storing the user's private keys in secure card storage, and using card encryption capabilities to generate authentication information from the keys, the encryption technology need not be implemented in the host system. The cards' ability to generate random numbers may also be useful as a source for keys.

However, implementation of public-key cryptography in a smart card application poses numerous challenges. Smart cards present a combination of implementation constrains that other platforms do not. Constrained memory and limited computing power are two of them. Smart cards are also slow transmitters, so to achieve acceptable application speeds, data elements must be small. While cryptographic services that are efficient in memory usage and processing power are needed to contain costs, reductions in transmission times are also needed to enhance usability.

The strength of the ECDLP algorithm means that strong security is achievable with proportionately smaller key and certificate sizes. The smaller key size in turn means that less EEPROM is required to store keys and certificates and the less data needs to be passed between the card and the application so that transmission times are shorter.

As smart card applications require stronger and stronger security, ECC can continue to provide the security with proportionately fewer additional system resources [11]. This means that with ECC, smart cards are capable of providing higher levels of security without increasing their cost.

The nature of actual computations – more specifically, ECC's reduced processing times – also contribute significantly to explaining why ECC meets the smart card platform requirements so well [17]. Other public-key systems involve so much computation that a dedicated hardware device known as a crypto processor is required. The crypto coprocessors not only takes up precious space – it adds about 20 to 30 percent to the cost of the chip, and about three to five dollar to the cost of each card [17]. With ECC, the algorithm can be implemented in available ROM, so no additional hardware is required to perform strong, fast authentication.

As mentioned earlier, the private key in a public-key pair must. At all cost, be kept secret. Secure environments meet this requirement by personalizing cards, i.e. keys are loaded or injected into the cards. Because of the complexity of the computation required, generating keys on the card is inefficient and often impractical.

With ECC, the time needed to generate a key pair is so short that even a device with the very limited computing power of a smart card can generate the key pair, provided a good random number generator is available [10]. This means that the card personalization process can be streamlined for applications in which nonrepudiation is important.

4.6 Conclusion

What is the point of using cryptography? Why consider using cryptography as part of the commercial security strategy? Suppliers, customers and staff are trustworthy and treat confidential information with respect, and can be trusted not to fake messages, documents and instructions in the course of their work?

The fact is that the modern organisation faces a number of threats to its corporate existence. Corporations have always had to deal with theft, fraud and vandalism; various remedies have been adopted through the years including fences, security guards, time clocks, double entry accounting methods, auditing, and of course, controlled delegation of signoff authority.

The main threat to organisations today is no longer physical – it is *information*. Money, corporate secrets, intellectual property, and even information such as customer lists, supplier approvals, and purchase orders are no longer stored in a safe in the organisation; it is stored digitally in a networked computer system. Procedures and processes that have been in use for many years now are revised to take into account the fact that a organisation with a successful e-business strategy can no longer afford to trust a signature on a piece of paper. Electronic systems are moving to the point where a system, which allows networked computer systems to authenticate instructions, messages and documents, gives a degree of assurance as to the confidentiality of information transmitted and received and also the integrity of the information, i.e. that it has not been changed in transit across the network from one computer to another.

Cryptographic solutions can provide a technical underpinning to good security policies, practices, guidelines and procedures. It can be used to establish the integrity and authenticity of information stored, processed or transmitted in electronic form.

Chapter 5. Elliptic Curve Cryptography Overview

Elliptic curves provide public-key methods that are fast and use small keys, while providing high level of security. This chapter provides an introduction to Elliptic Curves and how they are used to create a secure and powerful cryptosystem.

5.1 Introduction to ECC Cryptography

While the 20-year history of public key cryptography has seen a diverse range of proposals for candidate hard problems, only two have stood the test of time. These problems are known as the discrete logarithm problem over a finite field and integer factorisation [6].

In 1985, Neal Koblitz and V.S. Miller independently proposed using elliptic curves for public key cryptosystems. They did not invent a new cryptographic algorithm with elliptic curves over finite fields, but they implemented existing algorithms, like Diffie-Hellman, using elliptic curves [9].

Elliptic curves are rich mathematical structures, which have shown themselves to be useful in a range of applications including primality testing and integer factorisation [26]. One potential use of elliptic curves is in the definition of public key cryptosystems that are close analogues of existing schemes [26]. In this way, variants of existing schemes can be devised so that they rely for their security on a different underlying hard problem.

Introduction to some relevant mathematical terminology can be found in Appendix 1.

The mathematics behind elliptic curve cryptography is described next.

5.2 Weierstrass Equation and Elliptic Curves

Let F_q denote the finite field containing q elements, where q is a prime power. If K is a field, let $\overline{K}$ denote its algebraic closure. (If $K = F_q$ then $\overline{K} = \bigcup_{m \geq 1} F_{q^m}$)

The projective plane $P^2(K)$ over K is the set of equivalence classes of the relation ~ acting on $K^3 \setminus \{0; 0; 0\}$, where $(x_1, y_1, z_1) \sim (x_2, y_2, z_2)$ if and only if there exists $u \in K^*$ such that

$$x_1 = ux_2;\ y_1 = uy_2,\ and\ z_1 = uz_2.$$

We denote the equivalence class containing (x, y, z) by (x : y : z). A Weierstrass equation is a homogeneous equation of degree 3 of the form

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3$$

where $a_1, a_2, a_3, a_4, a_6 \in K$. The Weierstrass equation is said to be smooth, or non-singular, if for all projective points $P = (X : Y : Z) \in P^2(\overline{K})$ satisfying

$$F(X,Y,Z) = Y^2Z + a_1XYZ + a_3YZ^2 - X^3 - a_2X^2Z - a_4XZ^2 - a_6Z^3 = 0$$

at least one of the partial derivatives $\frac{\partial F}{\partial X}, \frac{\partial F}{\partial Y}, \frac{\partial F}{\partial Z}$ is non-zero at P. If all three partial derivatives vanish at some point P, then P is called a singular point, and the Weierstrass equation is said to be singular [5, 32].

An elliptic curve E is the set of all solutions in $P^2(\overline{K})$ of a smooth Weierstrass equation. There is exactly one point in E with Z-coordinate is equal to 0, namely (0 : 1 : 0). This point is the point at infinity and it is denoted by O [5, 32].

For convenience, the Weierstrass equation for elliptic curves can be written using non-homogeneous (affine) coordinates, where $x = X/Z$, $y = Y/Z$,

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \quad (5.1)$$

An elliptic curve E is then the set of solutions to equation above in the affine plane $A^2(\overline{K}) = \overline{K} \times \overline{K}$, together with the extra point at infinity O [5, 32].

5.3 Discriminant and j-invariant

Let E be a curve given by a non-homogeneous Weierstrass equation (4.1). Define the quantities

$$d_2 = a_1^2 + 4a_2$$

$$d_4 = 2a_4 + a_1a_3$$

$$d_6 = a_3^2 + 4a_6$$

$$d_8 = a1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2$$

$$c_4 = d_2^2 - 24d_4$$

$$\Delta = -d_2^2d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6$$

$$j(E) = c_4^3/\Delta$$

The quantity Δ is called the discriminant of the Weierstrass equation, while $j(E)$is called the j-invariant of E if $\Delta \neq 0$. The curve E is non-singular, if and only if $\Delta \neq 0$ [5].

5.4 Fields

When implementing an elliptic curve cryptosystem, an important consideration is how to implement the underlying field arithmetic. There are two possible field types from which to choose: Fields of odd characteristic and fields of characteristic two.

Two questions of particular importance are whether to use even or odd characteristic fields and secondly, whether to restrict implementation to fields of a special type, for efficiency, or to support any type of finite field [18].

5.4.1 Fields of Odd Characteristic

Recall that the field F_p uses the numbers from 0 to $p - 1$, and computations end by taking the remainder on division by p. An elliptic curve with the underlying field of F_p can be formed by choosing the variables a and b within the field of F_p. The elliptic curve includes all points (x,y) which satisfy the elliptic curve equation modulo p (where x and y are numbers in F_p).

There are four standard arithmetic operations needed in F_p, namely addition, subtraction, multiplication and division. It is, however, the last two of these (and particularly the last), which produce the greatest challenge [18].

If an elliptic curve E is defined over a field K whose characteristic is neither 2 nor 3, then the Weierstrass equation for the curve can be simplified considerably:

$$E : y^2 = x^3 + ax + b,\ a,\ b \in K.$$

That is, a Weierstrass equation can be selected for E so that $a1 = a2 = a3 = 0$ [32].

5.4.2 Fields of Characteristic Two

We now go through the case of arithmetic in F_2^n, where $n \geq 1$. In this case, the expression for the j-invariant reduces to $j(E) = a_1^{12}/\Delta$. In fields of characteristic two, the condition $j(E) = 0$, i.e. $a_1 = 0$, is equivalent to the curve being supersingular [18]. This very special type of curve is avoided in cryptography because of the MOV attack [21].

Field elements in fields of characteristic two are represented as binary vectors of dimension n, relative to given basis $(\alpha_0, \alpha_1, ..., \alpha_{n-1})$ of F_2^n as a linear space over F_2. Field addition and subtraction are implemented as component-wise exclusive-or, while the implementations of multiplication and inversion depend on the basis chosen [18].

Finite fields of characteristic two are attractive to implementers because of their carry-free arithmetic, and the availability of different equivalent representations of the field, which can be adapted and optimised for the computational environment at hand [18].

If an elliptic curve is defined over a field K, which is of characteristic two, the Weierstrass equation for the curve can be simplified considerably [18]. The simplification depends on the j -invariant of E, $j(E)$, as follows:

If $j(E) \neq 0$, transforms E to the curve

$$E : y^2 + xy = x^3 + a_2x^2 + a_6$$

Else if $j(E) = 0$ (i.e. E is supersingular), transforms E to the curve

$$E : y^2 + a_3y = x^3 + a_4x + a_6$$

The three different bases, which can be used to implement fields of characteristic two are polynomial base, normal base, and subfield base [5, 32].

- Polynomial base. A polynomial (or standard) base is of the form $(1, \alpha, \alpha^2, ..., \alpha^{n-1})$, where α is a root of an irreducible polynomial $f(x)$ of degree n over F^2.The field is then realized as $F^2[x] = (f(x))$, and the arithmetic is that of polynomials of degree at most $n-1$, modulo $f(x)$. $(f(x))$ is the cyclic group generated by $f(x)$ [5].

- Normal base. A normal base of F_2^n over F^2 has the form $(\alpha, \alpha^2, \alpha^{2^2}, ..., \alpha^{2^{n-1}})$ for some $\alpha \in F_2^n$. It is known that such bases exist for all $n \geq z$. Normal bases are useful mostly in hardware implementations. First, the field squaring operation is trivial in normal base representations, as it amounts to just cyclic shifting of the binary vector representing the input operand. More importantly, normal bases allow for the design of efficient bit-serial multipliers [18].

  The existence of optimal normal bases (ONB) has been completely characterized in [37] and [20]. In particular, an ONB of F_2^n over F^2 exists if and only if one of the following conditions holds [18]:

  1. $n+1$ is prime, and 2 is primitive in F_{n+1}; then the n non-trivial $(n+1)$st roots of unity form an ONB of F_2^n over F_2, called a Type I ONB.

  2. $2n + 1$ is prime, and either

     (a) 2 is primitive in F_{2n+1} or

     (b) $2n + 1 \equiv 3 (mod\ 4)$ and the multiplicative order of 2 in F_{2n+1} is n; that is 2 generates the quadratic residues in F_{2n+1};

then, $\alpha = \gamma + \gamma^{-1}$ generates an ONB of F_2^n over F_2, where is a primitive $(2n + 1)$st root of unity; this is called a Type II ONB.

The bit-serial multipliers that can be very effective for ONBs in hardware do not always map nicely to efficient software implementations, as single bit operations are expensive in the latter. It turns out, that by applying simple permutations, operations on ONB representations of both Types I and II can be handled through polynomial arithmetic, in a manner similar to the case of standard bases [18].

- Subfield base. When $n = n_1 n_2$, we can regard F_2^n as an extension of degree n_2 of F_2^{n1}, and represent elements of F_2^n using a base of the form $\alpha_i \beta_j : 0 \leq i \leq n1, 0 \leq j \leq n2$, where $\beta_0, \beta_1, \ldots, \beta_{n2-1}$ form a base of F_2^n over $F_2^n{}_1$, and $\alpha_0, \alpha_1, \ldots, \alpha_{n1-1}$ form a base of F_2^{n1} over F_2 [18]. Thus, arithmetic can be done in two stages, with an outer section doing operations on elements of F_2^n as vectors of symbols from $F_2^n{}_1$; and inner section performing the operations on the symbols as binary words.

Any combination of bases can be used, e.g. normal base for the outer section, and polynomial base for the inner one.

5.5 Arithmetic

5.5.1 Group Law

The points on an elliptic curve form an Abelian group under a certain addition. Let E be an elliptic curve given by the Weierstrass equation (5.1). The additional rules for points P and Q are as follows [32]:

For all $P, Q \in E$,

1. $O + P = P$ and $P + O = P$. That is, O is identity element.
2. $-O = O$.
3. If $P = (x_1, x_2) \neq O$, then $-P = (x_1, -y_1 - a_1x_1 - a_3)$.
4. $Q = -P$, then $P + Q = O$.
5. If $P \neq O$, $Q \neq O$, $Q \neq -P$, then let R be the third point of intersection (counting multiplicities) of either the line which intersects P and Q if $P \neq Q$, or the tangent line to the curve at P if $P = Q$, with the curve. Then $P + Q = -R$.

5.5.2 Point Addition

Let P and Q be two distinct rational points on E, $E : y^2 = x^3 + ax + b$, $a, b \in K$. The straight line joining P and Q must intersect the curve at one further point, R, since we are intersecting a line with a cubic curve. The point R will also be rational since the line, the curve and the points P and Q are themselves all defined over K [5, 32].

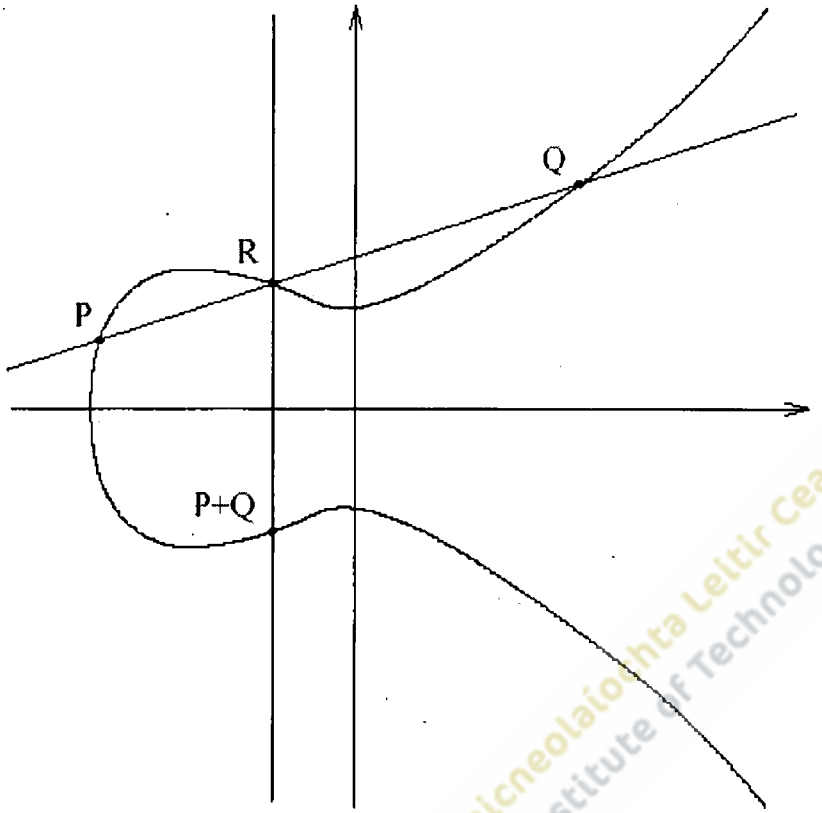

Figure 5.1: Adding two points on an elliptic curve

If we then reflect R in the x-axis, we obtain another rational point, which we shall call $P + Q$ (see Figure 5.1) [5, 32].

There are different addition formulas for fields of characteristic $p > 3$ and for fields of characteristic two. Curve is defined in fields of characteristic $p > 3$ if $K = F_q$, where $q = p^n$ for a prime $p > 3$ and an integer $n \geq 1$ [5, 32].

5.5.3 Addition formula for fields of characteristic p > 3:

Let $P = (x_1, y_1) \in E$, then $-P = (x_1, -y_1)$. If $Q = (x_2, y_2) \in E$, $Q \neq -P$ and $P \neq Q$, then $P + Q = (x_3, y_3)$, where

$$x_3 = \lambda^2 - x_1 - x_2$$

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

and

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}$$

5.5.4 Addition formula for fields of characteristic two

Let $P = (x_1, y_1) \in E_1$; then $-P = (x_1, y_1 + x_1)$. If $Q = (x_2, y_2) \in 2\ E_1$, $Q \neq -P$ and $P \neq Q$, then $P + Q = (x_3, y_3)$, where

$$x_3 = \left(\frac{y_1 + y_2}{x_1 + x_2}\right)^2 + \frac{y_1 + y_2}{x_1 + x_2} + x_1 + x_2 + a_2$$

and

$$y_3 = \left(\frac{y_1 + y_2}{x_1 + x_2}\right)(x_1 + x_3) + x_3 + y_1$$

5.5.5 Point Doubling

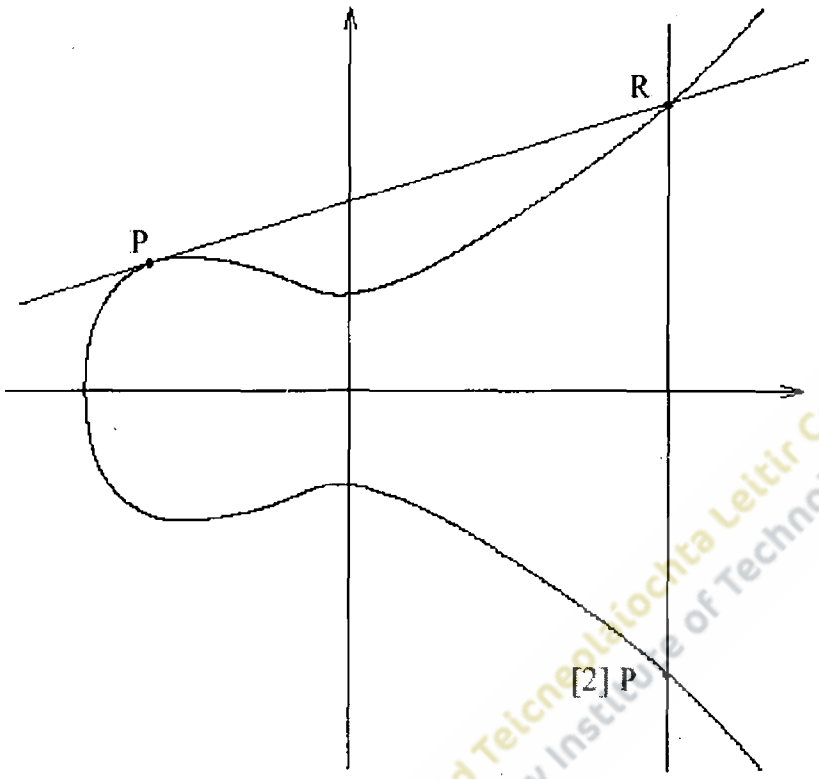

Figure 5.2: Doubling a point on an elliptic curve

To add P to itself, or to double P, we take the tangent to the curve at P. Such a line must intersect E in exactly one other point, say R, as E is defined by a cubic equation. Again we reflect R in the x-axis to obtain a point which we call $[2]P = P + P$ (see Figure 5.2) . If the tangent to the point is vertical, it intersects the curve at the point at infinity and $P + P = O$, i.e. P is a point of order 2 [5, 32].

5.5.6 Doubling formula for fields of characteristic p > 3

Let $P = (x_1, y_1) \in E$, $Q = (x_2, y_2) \in E$, $P = Q$ then $P + Q = (x_3, y_3)$, where

$$x_3 = \lambda^2 - x_1 - x_2$$

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

and

$$\lambda = \frac{3{x_1}^2 + a}{2y_1}$$

5.5.7 Doubling formula for fields of characteristic two

Let $P = (x_1, y_1) \in E_1$, $Q = (x_2, y_2) \in E_1$, and $P = Q$, then $P + Q = (x_3, y_3)$, where

$$x_3 = {x_1}^2 + \frac{a_6}{{x_1}^2}$$

and

$$y_3 = {x_1}^2 + \left(x_1 + \frac{y_1}{x_1}\right) x_3 + x_3$$

5.5.8 Doubling formula, when j(E) = 0 (i.e. E is supersingular)

Let $P = (x_1, y_1) \in E_2$, $Q = (x_2, y_2) \in E_2$, and $P = Q$, then $P + Q = (x_3, y_3)$, where

$$x_3 = \frac{{x_1}^4 + {a_4}^2}{{a_3}^2}$$

and

$$y_3 = \left(\frac{{x_1}^2 + a_4}{a_3}\right)(x_1 + x_3) + y_1 + a_3$$

The best solution would be to select a curve and field *K* so that the number of field operations involved in adding two points and doubling a point are minimized.

5.6 Elliptic Curve Discrete Logarithm Problem

At the foundation of every cryptosystem is a hard mathematical problem that is computationally infeasible to solve. The discrete logarithm problem is the basis for the security of many cryptosystems including the Elliptic Curve Cryptosystem.

There are two geometrically defined operations over certain elliptic curve groups. These two operations were point addition and point doubling. By selecting a point in an elliptic curve group, one can double it to obtain the point *2P*. After that, one can add the point *P* to the point *2P* to obtain the point *3P*. The determination of a point *nP* in this manner is referred to as Scalar Multiplication of a point. The ECDLP is based upon the intractability of determining *n* given *P* and *nP*.

In the multiplicative group *Zp**, the discrete logarithm problem is: given elements r and q of the group, and a prime *p*, find a number *k* such that $r = qk \bmod p$. If the elliptic curve group is described using multiplicative notation, then the elliptic curve discrete logarithm problem is: given points *P* and *Q* in the group, find a number that $Pk = Q$; *k* is called the discrete logarithm of *Q* to the base *P*.

When the elliptic curve group is described using additive notation, the elliptic curve discrete logarithm problem is: given points P and Q in the group, find a number k such that $Pk = Q$ One of the advantages of ECC is that the elliptic curve discrete logarithm problem is believed to be harder than both the integer factorisation problem and discrete logarithm problem modulo p. This extra difficulty implies that ECC is one of the strongest public key cryptographic systems known today [39].

The elliptic curve discrete logarithm problem is relatively easy for a small class of elliptic curves, known as supersingular elliptic curves and also for certain anomalous elliptic curves [39]. In both cases, the weak instances of the problem are easily identified, and an implementation merely checks that the specific instance selected is not one of the classes of easy problems.

Basically, elliptic curve cryptography is constructed on similar concepts to those used for discrete logarithm systems, but the discrete logarithm functions are performed on elliptic curves over finite fields [7]. A major factor in accepting ECC is the smaller cryptographic key size [7].

With small electronic commerce and banking type transactions this may be an important consideration in overall system performance. There are many possible algorithms to use for encryption with elliptic curves. As stated before (in the beginning of Chapter 4), many discrete logarithm problems can be converted to use elliptic curves. The newest version of IEEE's P1363 standard does not however define any encryption algorithm to use with elliptic curves.

Only two elliptic curve signature schemes are given in the IEEE P1363 standard: Nyberg-Rueppel and ECDSA. They are similar in overall security. The security of both schemes

depends on the order of the base point being a large prime number [6]. I chose to implement the Nyberg-Rueppel signature scheme because Certicom's test results listed in Table 5.1 claim that its operations are faster than ECDSA's.

Function	Security Builder 1.2 163-bit ECC (ms)	BSAFE 3.0 1024-bit RSA (ms)
Key Pair Generation	3.8	4708.3
Sign	2.1 (ECNRA) 3.0 (ECDSA)	228.4
Verify	9.9 (ECNRA) 10.7 (ECDSA)	12.7

Table 5.1 Performance time for RSA and ECC systems

5.7 Nyberg-Rueppel signature scheme

The Nyberg-Rueppel signature scheme is defined as follows [25, 45]. Let E be an elliptic curve defined over Zp (p > 3 prime) such that E contains a cyclic subgroup H in which the discrete logarithm problem is intractable.

Let $P = Z^*p \times Z^*p$, $C = E \times Z^*p \times Z^*p$, and define

$$K = \{(E, \alpha, a, \beta) : \beta = a\alpha\}$$

where $\alpha \in E$. The values α and β are public, and a is secret.

For $K = (E; \alpha; a; \beta)$, for a (secret) random number $k \in Z_{|H|}$, and for $x = (x1, x2) \in Z^*p \times Z^*p$, define

$$sig_K(x, k) = (c, d)$$

where

$$(y_1, y_2) = k\alpha$$

$$c = y_1 + \text{hash}(x) \bmod p$$

$$ver_K(x, c, d) = \text{true} \Leftrightarrow \text{hash}(x) = e,$$

where

$$(y_1, y_2) = d\alpha + c\beta$$

$$e = c - y_1 \bmod p$$

All signature schemes require a hash of a document that is to be signed. The IEEE's P1363 standard [56] suggests SHA-1 [50], defined by NIST [52], or RIPEMD-160 [80], defined by the ISO-IEC. The reason for using the hash algorithm is to make it impossible to find a match between the real input and some minor changed version that would give the same hash value. The problem is considered exceptionally difficult to solve with the above hash algorithms [45].

5.7 Conclusion

Curves over $K = F_2^n$ are preferred for the following reasons:

1. The arithmetic in F_2^n is easier to implement in computer hardware than the arithmetic in finite fields of characteristic greater than 2.
2. When using a normal basis representation for the elements of F_2^n, squaring a field element becomes a simple cyclic shift of the vector representation, and thus the multiplication count in adding two points is reduced.
3. With curves over F_2^n it is easy to recover the y-coordinate of a point given its x-coordinate plus a single bit of extra information.
4. For supersingular curves over F_2^n, the inverse operation in doubling a point can be eliminated by choosing $a_3 = 1$, further reducing the operation count.

In general software environments, the use of F_2^n offers significant performance advantages over F_p. This holds true for platforms such as a Sun Sparc station, a HP server, an embedded system, and more importantly, for a low-cost, 8-bit smart card. To achieve equivalent performance with F_p, a crypto coprocessor is required [15].

The security of the ECC rests on the difficulty of the elliptic curve discrete logarithm problem. As is the case with the integer factorisation problem and the discrete logarithm problem modulo p, no efficient algorithm is known at this time to solve the elliptic curve discrete logarithm problem [15].

One of the advantages of ECC is that the elliptic curve discrete logarithm problem is believed to be harder than both the integer factorisation problem and discrete logarithm problem modulo p. This extra difficulty implies that ECC is one of the strongest public key cryptographic systems known today [15].

The elliptic curve discrete logarithm problem is relatively easy for a small class of elliptic curves, known as supersingular elliptic curves and also for certain anomalous elliptic curves [15].

Elliptic curve cryptography is constructed on similar concepts to those used for discrete logarithm systems, but the discrete logarithm functions are performed on elliptic curves over finite fields [17].

A major factor in accepting ECC is the fact of smaller cryptographic key sizes [17]. With small, electronic commerce and banking type transactions this may be an important consideration in overall system performance.

There are many possible algorithms to use for encryption with elliptic curves. As stated before, many discrete logarithm problems can be converted to use elliptic curves. The newest version of IEEE's P1363 standard does not however define any encryption algorithm to use with elliptic curves [30].

Chapter 6. Application Implementation

6.1 Implementation

Java Card applet was implemented using Java development environment Java Developer Kit (for more details see [58]).

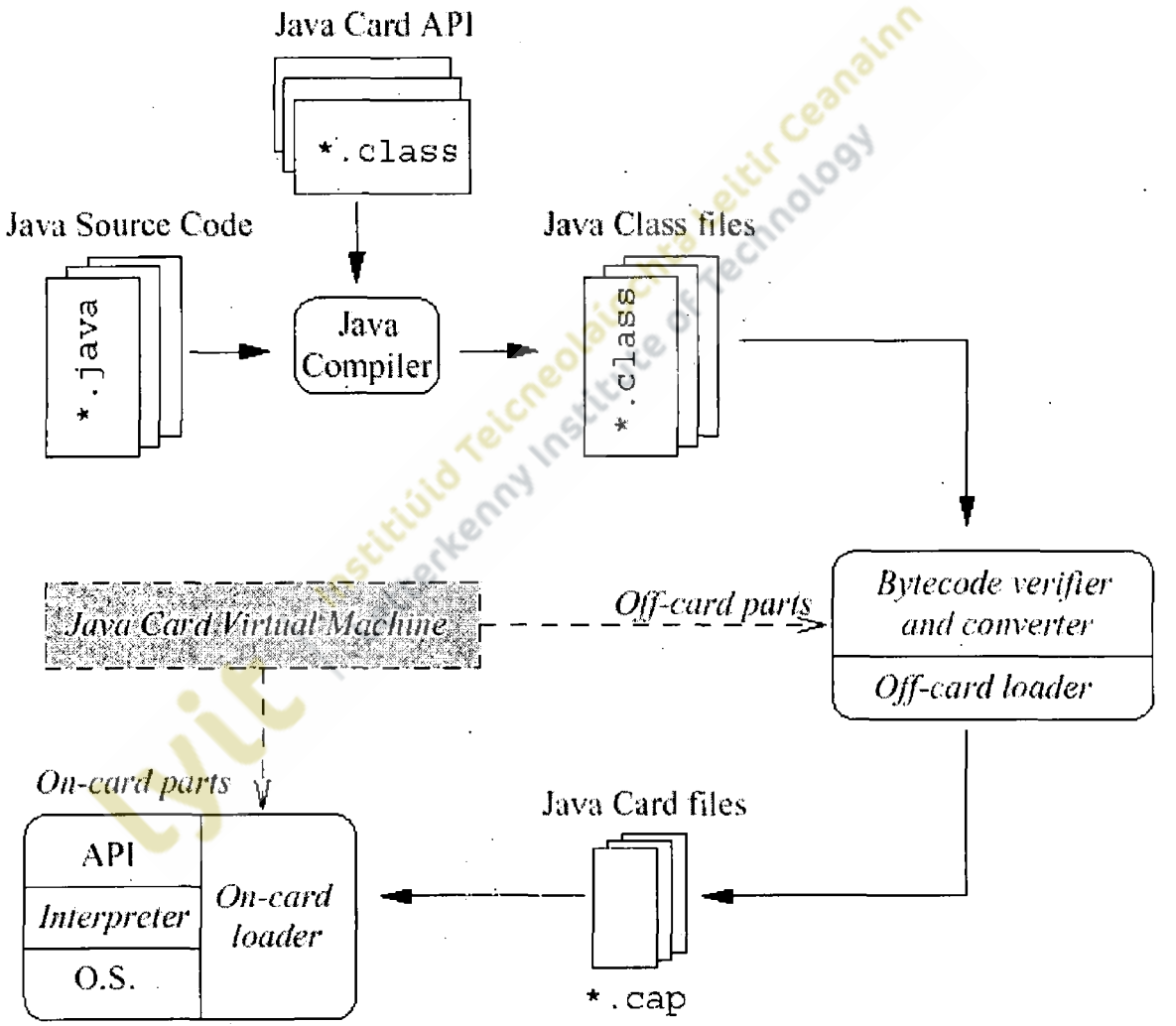

Figure 6.1 Java Card Development tools

This environment offers a Java Card simulator, which allowed to concentrate on application functionality and correct API usage. To this end, the applet is run in a typical desktop JVM but accessed only over the smart card-specific communication interface.

Thus, code that will later interact with the card's applet can be tested during applet development. In integrated development environments, the only visible differences between applet development for the PC and the Java Card are the particular set of API calls for Java Card applets and a post-processing stage, which is required to adapt the Java bytecode to the resource-constrained smart card environment (see Figure 6.1).

After compiling the applet's code, a converter is run on the Java class files preparing them for download. All classes of one package are downloaded to a card at the same time. To accomplish this, classes are packaged and converted into a Java Card Cardlet Package (CAP) file. The Converter removes the space-intensive link information of a typical Java class file: the field, method, and class names and loads and processes class files that make up a Java package. In addition to class files, the Converter loads export files.

The Converter verifies that the class files comply with all limitations. It also checks the correctness of export files. This means that:

- all input class files are compatible with each other.
- export files of imported packages are consistent with class files that were used for compiling the converting package.

A Java Card export file contains the public API linking information of classes in an entire package. The Unicode string names of classes, methods and fields are assigned unique

numeric tokens. Export files are not used directly on a device that implements a Java Card virtual machine. However, the information in an export file is critical to the operation of the Virtual Machine on a device. The Converter produces an export file when a package is converted. This package's export file can be used later to convert another package that imports classes from the first package. Information in the export file is included in the CAP file of the second package, and then it is used on the device to link the contents of the second package to items imported from the first package. During the conversion, when the code in the currently converted package references a different package, the Converter loads the export file of the different package.

Conversion removes a significant amount of data that is necessary only for supporting the dynamic download of new classes at runtime. The Java Card platform has no need to support such runtime downloads as a smart card's primary purpose is to "do away with" the need for online connectivity

The data flow of the installation process is as follows (see Figure 6.2):

1. An off-card installer takes a CAP file, produced by the Java Card converter, as the input, and produces a text file that contains a sequence of APDU commands.

2. This set of APDUs is then read by the APDUTool and sent to the on-card installer.

3. The on-card installer processes the CAP file contents contained in the APDU commands as it receives them.

4. The response APDU from the on-card installer contains a status and optional response data.

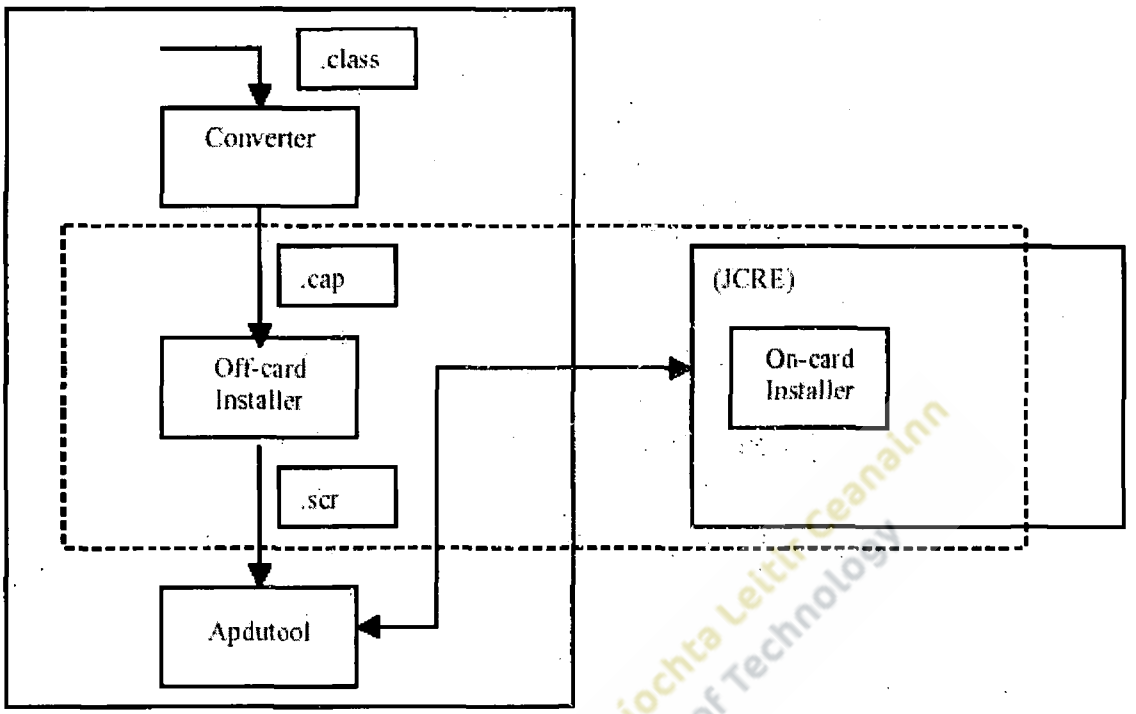

Figure 6.2 Components of the installer

6.2 Applet Specifications

6.2.1 Specifying functions of the applet.

The idea behind this application is to allow the card user to recognize a counterfeit Java Card terminal. The user does not need any additional technical aids or equipment to check the terminal.

The proposed solution involves storing a password, known only to the card user, in a file in Java Card. The terminal can read this file only after it has successfully authenticated itself with respect to the Java Card via a secret key.

After the authentication process, the terminal is allowed to read the password from the file and show it on the display. As soon as the card user sees the password and verifies that it is correct, he or she can assume that the terminal is genuine, since only the user knows the password.

The following is a brief description of Java Card – Terminal communication:

Java Card	**Terminal**
	Signature Key Generation
	Signature Generation
Signature Verification	
IF Signature = OK	
THEN show password on the screen	
ELSE abort	

Table 6.1 Java Card ⇔ Terminal Communication

6.2.2 Specifying AIDs

The Java classes of the *Examples* applet are defined in a Java package. The fictitious AIDs for the *Examples* applet and the applet package are defined as illustrated in Table 6.2.

Package AID		
Field	Value	Length
RID	0xF2, 0x34, 0x12, 0x34, 0x56	5 bytes
PIX	0x10, 0x00, 0x00	3 bytes
Applet AID		

Field	Value	Length
RID	0xF2, 0x34, 0x12, 0x34, 0x56	5 bytes
PIX	0x10, 0x00, 0x01	3 bytes

Table 6.2 AID for the applet

The package AID and the applet AID have the same RID value; their PIX values differ at the last bit.

6.2.3 Defining the class structure and method functions of the applet

A Java Card applet class extends from the *javacard.framework.Applet* class. This class is the superclass for all applets residing on a Java Card. It defines the common methods an applet must support in order to interact with the JCRE during its lifetime.

Table 6.3 lists the public and protected methods defined in the class *javacard.framework.Applet*:

Method summary	
public void	*deselect()* Called by the JCRE to inform the currently selected applet that another (or the same) applet will be selected.
public Shareable	*getShareableInterfaceObject (AID client AID, byte parameter)* Called by the JCRE to obtain a sharable interface object from this server applet on behalf of a request from a client applet.
public static void	*install (byte[] bArray, short bOffset, byte bLength)* The JCRE calls this static method to create an instance of the *Applet* subclass.
public *abstract void*	*process (APDU apdu)*

void	Called by the JCRE to process an incoming APDU command.
protected final void	*register ()* This method is used by the applet to register this applet instance with the JCRE and assign the default AID in the CAD file to the applet instance.
protected final void	*register (byte[] bArray, short bOffset, byte bLength)* This method is used by the applet to register this applet instance with the JCRE and to assign the specified AID in the array *bArray* to the applet instance.
public boolean	*select ()* Called by the JCRE to inform this applet that it has been selected.
protected final boolean	*selectingApplet ()* This method is used by the applet *process()* method to distinguish the *SELECT APDU* command that selected this applet from all other SELECT *APDU* APDU commands that may relate to file or internal applet state selection.

Table 6.3 Methods for javacard.framework.Applet class

The class *javacard.framework.Applet* provides a framework for applet execution. Methods defined in this class are called by the JCRE when the JCRE receives APDU commands from the CAD.

After the applet code has been properly loaded on a Java Card and linked with other packages on the card, an applet's life starts when an applet instance is created and registered with the JCRE's registry table [56]. An applet must implement the static method *install()* to create an applet instance and register the instance with the JCRE by invoking one of the two *register()* methods. The *install()* method takes a byte array as a parameter. This array contains the installation parameters for initialising or personalizing the applet instance.

```
        public static void install( byte[] bArray, short bOffset, byte bLength )
        {
                new Examples(bArray, bOffset, bLength);
        }
int NR_Verify( Message, length, public_curve, signer_point, signature)
```

Examples performs memory allocation, where all parameters received by install().

```
protected Examples(byte[] bArray, short bOffset, byte bLength)
{
        Message = new byte[MSG_LENGTH];
        PCurve = new byte[CURVE_LENGTH];
        SPoint = new byte[POINT_LENGTH];
        SIGNATURE = new short[SIG_LENGTH];

        TN = 0;
        PUN = 0;
        isPersonalized = false;

        //Create transient objects.
        TransientShorts = JCSystem.makeTransientShortArray
                                ( NUM_TRANS_SHORTS,
                                JCSystem.CLEAR_ON_DESELECT);
        transientBools = JCSystem.makeTransientBooleanArray
                                ( NUM_TRANS_BOOLS,
                                JCSystem.CLEAR_ON_DESELECT);
         CAD_ID_array = JCSystem.makeTransientByteArray( (short)4,
                                  JCSystem.CLEAR_ON_DESELECT);
        byteArray8 = JCSystem.makeTransientByteArray( (short)8,
                                JCSystem.CLEAR_ON_DESELECT);

        byte aidLen = bArray[bOffset];
        if (aidLen== (byte)0)
                register();
        else
                register(bArray, (short)(bOffset+1), aidLen);
}
```

An applet on a Java Card is in an inactive stage until it is explicitly selected [56]. When the JCRE receives a *SELECT* APDU command, it searches its internal table for the applet whose

AID matches the one specified in the command. If a match is found the applet returns *true* to the *select()* method if it is now ready to become active and to process subsequent APDU commands. Otherwise, the applet returns *false* to decline its participation, and if so, no applet will be selected. The *javacard.framework.Applet* class provides a default implementation for both the *select()* and *deselect()* methods. A subclass of the *Applet* class may override these two methods to define the applet's behaviour during selection and deselection.

Once an applet is selected, the JCRE forwards all subsequent APDU commands (including the *SELECT* command) to the applet's *process()* method. In the *process()* method, the applet interprets each APDU command and performs the task specified by the command. For each command APDU, the applet responds to the CAD by sending back a response APDU, which informs the CAD of the result of processing the command APDU. The *process()* method in class *javacard.framework.Applet* is an abstract method: a subclass of the *Applet* class must override this method to implement an applet's functions.

This command-and-response dialogue continues until a new applet is selected or the card is removed from the CAD [77]. When deselected, an applet becomes inactive until the next time it is selected.

6.2.4 Defining the interface between an applet and its terminal application

An applet running in a smart card communicates with the terminal application at the CAD using APDU. In essence, the interface between an applet and its terminal application is a set of

APDU commands that are agreed upon and supported by both the applet and the terminal application.

Applet supports a set of APDU commands, comprising a *select* APDU command and a few *process* APDU commands.

The *select* command instructs the JCRE to select the applet on the card. The set of *process* commands defines the commands the applet supports. These are defined in accordance with the functions of the applet.

```
private void SelectExamples(APDU apdu)
{
        byte[] buffer = apdu.getBuffer();
        buffer[0] = FCI_TEMPLATE_TAG;

        //AID
        buffer[2] = FCI_AID_TAG;
        buffer[3] = JCSystem.getAID().getBytes(buffer, (short)4);
        short offset=(short)(3+buffer[3]);

        // PROPRIETARY DATA
        buffer[offset++] = (byte)FCI_PROPERIETARY.length;
        offset = Util.arrayCopyNonAtomic(FCI_PROPERIETARY, (short)0,
                                buffer, offset,
                                (short)FCI_PROPERIETARY.length);

        // FCI template length
        buffer[1] = (byte)(offset-(short)2);

        apdu.setOutgoingAndSend((short)0, offset);
}
```

Java Card technology specifies the encoding of the *select* APDU command. Developers can define *process* commands as long as they comply with the structure outlined in 3.4.

```
public void process(APDU apdu)
```

```
{
        byte[] buffer = apdu.getBuffer();

        // Mask channel info out
        buffer[ISO.OFFSET_CLA]=(byte)(buffer[ISO.OFFSET_CLA]& (byte)0xFC);

        if (buffer[ISO.OFFSET_CLA] == EXAMPLE_CLA)
        {
                switch (buffer[ISO.OFFSET_INS])
                {
                        case INITIALIZE:        processSelectVerify(apdu); break;
                        case COMPLETE:          processCompleteVerify(apdu); break;
                        default:
                                ISOException.throwIt(ISO.SW_INS_NOT_SUPPORTED);
                }
        }
        else
        {
                if (buffer[ISO.OFFSET_CLA] == ISO.CLA_ISO7816)
                {
                        if (buffer[ISO.OFFSET_INS] == VERIFY)
                                processVerifySignature(apdu);
                        else
                                ISOException.throwIt(ISO.SW_INS_NOT_SUPPORTED);
                }
                else
                        ISOException.throwIt(ISO.SW_CLA_NOT_SUPPORTED);
        }
}
```

For each command APDU, the applet first decodes the value of each field in the command. If the optional data fields are included, the applet also determines their format and the structure. Using these definitions, the applet interprets each command and reads the data. It then executes the task specified by the command.

For each response APDU, the applet defines a set of status words to indicate the result of processing the paired-command APDU. During normal processing, the applet returns the

success status word (0x9000, as specified in ISO[68]). If an error occurs, the applet returns a status word other than 0x9000 to denote its internal state. If the optional data field is included in the response APDU, the applet defines what to return.

In the *testing* applet, the applet supports VERIFY command for signature verification.

The *select* command and *process* APDU command for the testing applet are defined as illustrated in Tables 6.4 – 6.7.

CLA	INS	P1	P2	Lc	Data field	Le
0x0	0xA4	0x04	0x0	0x08	0xF2, 0x34, 0x12, 0x34, 0x56, 0x10. 0x0, 0x1	N/A

Table 6.4 Select APDU command

The data field contains the AID of the *testing* applet.

Optional data	Status Word	Meaning of Status word
No data	0x9000	Successful processing
	0x6999	Applet selection failed; the applet cannot be found or selected

Table 6.5 Response APDU

CLA	INS	P1	P2	Lc	Data field	Le
0xB0	0x20	0x0	0x0	Length of the signature data	Signature data	N/A

Table 6.6 Verify APDU command

Optional data	Status Word	Meaning of Status word
No data	0x9000	Successful processing
	0x6300	Verification failed

Table 6.7 Response APDU command

In addition to the status words declared in each response APDU command, the interface javacard.framework.ISO7816 defines a set of status words that signal common errors in applets, such as an APDU command formatting error.

6.2.5 Implementing error checking

Error checking is particularly important in smart card application development [29, 32]. An undetected error can cause the card to be blocked or result in the loss of critical data stored in the card.

Once an applet is installed in a smart card, it interfaces with the outside world only through APDU commands. The applet and the terminal application are agreed upon the significance of the value in each field of an APDU command, even though ISO7816 sets the protocol standard. It is useful for detecting illegal or ill-formatted commands.

In *Examples* applet the APDU commands are examined to ensure that the APDU header bytes (CLA, INS, PI, and P2) are set correctly, that the Lc or Le field matches with the data field length, that the signature has been verified correctly before response.

In general, before performing the task indicated by an APDU command, an applet validates the command according to its requirements. Then an applet confirms the following before carrying out a command:

- The APDU command is supported by the applet
- The APDU command is well formatted
- The APDU command meets the security or other internal conditions of the applet

While executing the task, the applet also detects whether the task can be performed successfully without leaving the applet in an invalid state.

To ensure that the terminal application knows what is going on inside the card the applet reports errors that occur to the terminal. If an error is detected, an applet will terminate the process and throw an ISOException containing a status word to indicate the processing state of the applet. If the ISOException is not handled by the applet, it will be caught by the JCRE, which then retrieves the status word and reports it to the terminal.

6.3 ECC system setup.

ECC requires the use of two types of mathematics (Figure 6.3):

- Modular big integer arithmetic
- The underlying finite field arithmetic

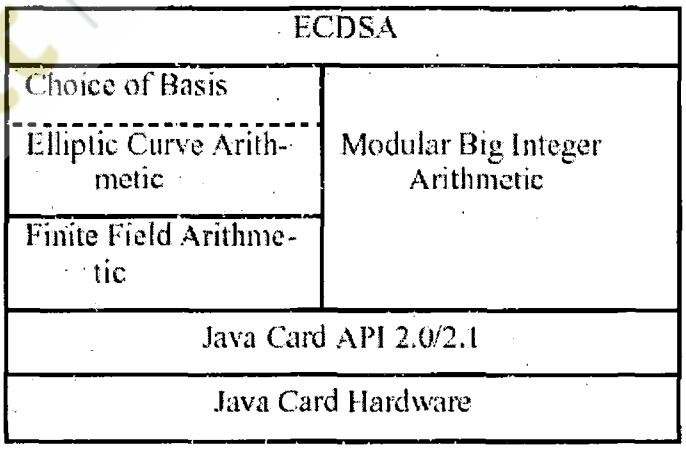

Figure 6.3. Layer structure of the smart card ECDSA architecture.

The developer must select the finite field when implementing ECC. Most of the computation takes place at the finite field level. While the pure elliptic curve operations can be built on top of many kinds of finite fields [50], there are two basic choices [14, 15, 17]:

- F_2^n, also known as characteristic two or even
- Fp, also known as integers modulo p, odd or odd prime.

Both of these finite fields are included in draft standards for ECC; mathematical discoveries to date suggest that the ECDLP is equally difficult for instances that use F_2^n as for those that use Fp where the sizes of the fields are approximately equal.

Although no security or standardization differences exist between the two types of underlying finite fields, performance and cost differences can arise when implementing a smart card application.

Before implementing an ECC system, several choices have to be made. These include selection of elliptic curve domain parameters (underlying finite field, field representation, elliptic curve), and algorithms for field arithmetic, elliptic curve arithmetic, and protocol arithmetic. The selections are influenced by security considerations, application platform (software, firmware, or hardware), constraints of the particular computing environment (e.g., processing speed, code size (ROM), memory size (RAM), gate count, power consumption), and constraints of the particular communications environment (e.g., bandwidth, response time).

In general software environments, the use of F_2^n offers significant performance advantages over Fp. This holds true for platforms such as a Sun SPARCstation, a HP server, an embedded

system, and more importantly, for low-cost, 8-bit smart card []. To achieve sub second performance with Fp, a crypto coprocessor is required. With F_2^n, a smart card is less expensive because a coprocessor is not needed to deliver sub second performance.

In software environments in which an arithmetic processor is already available for modular exponentiation, the performance of Fp can be improved so that in some cases it exceeds the performance of F_2^n [].

The following algorithms are used to speed up the inversion and scalar multiplication.

6.3.1 The Almost Inverse Algorithm

This algorithm is first developed by Dave Dahm [8]. It is based on Euclid's algorithm [9], but it leaves a final factor of x^k, which has to be divided out. The basic idea of the almost inverse algorithm is the same as polynomial inversion [8]. For the field we are working in, say F_2^{133}, the problem to be solved is:

Given a non-zero polynomial *A(u)* of degree ≤ 132, find the (unique) polynomial B(u) of degree ≤ 132 such that

$$A(u)B(u) \equiv 1 \bmod u^{133} + u^{42} + 1.$$

The problem has a simple, but relatively slow, recursive solution, exactly analogous to the related algorithm for integers. The Almost Inverse Algorithm is considerably faster. The Almost Inverse Algorithm computes *B(u)* and *k* such that

$$AB \equiv u^k \bmod M, \deg(B) < \deg(M), \text{ and } k < 2 \deg(M)$$

where *deg(B)* denotes the polynomial degree of *B*. After executing the algorithm, we will need to divide *B* by *uk(mod M)* to get the true reciprocal of *A*. The pseudo-code for the algorithm is given below. The computer implementation relies on a few representational items:

- Multiplication of a polynomial by *u* is a left-shift by 1 bit.
- Division of a polynomial by *u* is a right-shift by 1 bit.
- A polynomial is even if its least significant bit, the coefficient of u_0 is 0. Otherwise it is odd.

The algorithm will work whenever *A(u)* and *M(u)* are relatively prime, *A(u) 6= 0*, *M(u)* is odd, and *deg(M) > 0*.

The Almost Inverse Algorithm

Initialize integer $k = 0$, and polynomials $B = 1$, $C = 0$, $F = A$, $G = M$.

loop: while F is even, do $F = F / u$, $C = C * u$, $k = k + 1$.

if $F = 1$, then return B, k.

f $deg(F) < deg(G)$, then exchange F, G and exchange B, C.

$F = F + G$, $B = B + C$.

Goto loop.

6.3.2 Solinas's Addition-Subtraction Method

This algorithm is from [80]. The basic technique for elliptic scalar multiplication is the addition-subtraction method. This begins with the nonadjacent form (NAF) of the coefficient

n: a signed binary expansion with the property that no two consecutive coefficients are nonzero.

Just as every positive integer has a unique binary expansion, it also has a unique NAF. Moreover, NAF(n) has the fewest nonzero coefficients of any signed binary expansion of n [81]. There are several ways to construct the NAF of n from its binary expansion.

The idea is to divide repeatedly by 2. One can derive the binary expansion of an integer by dividing by 2, storing the remainder (0 or 1), and repeating the process with the quotient. To derive NAF, one allows remainders of 0 or ±1, one chooses whichever makes the quotient even.

---

k = n, S = ().

While k > 0 do

    If k is odd, then

        set u = 2 - (k (mod 4))

    else

        set u = 0.

    k = k - u.

    Prepend u to S.

    k = k / 2.

End While

Output S

We now implement elliptic scalar multiplication using NAF. Given the NAF

$$n = \sum_{i=0}^{l-i} e_i 2^i$$

the elliptic scalar multiplication Q = nP is performed as follows.

Q = P.

For i = l - 2 downto 1 do

Q = 2Q.

If e_i = 1 then set Q = Q + P.

If e_i = -1 then set Q = Q - P.

Output Q.

This is about one-eight faster than the binary method, which uses the ordinary binary expansion in place of the NAF [81].

6.4 Implementation of Elliptic Curve Cryptosystem

The implementation consists of the *Examples* package. The *Example* package contains of classes *Field, Curve, CustomField, Elliptic, Field, Onb, Point* and *NB.*

The implementation of normal basis arithmetic is quite simple, only bitwise and, bitwise exclusive-or, and shift operations are needed. The fact that these are the fastest operations possible on any microprocessor makes optimal normal base (ONB) attractive [8].

Squaring a normal base number amounts to a rotation. Addition is simply an exclusive-or operation.

The inversion uses Schroeppel's Almost Inverse algorithm, described in Chapter 6.3.2. The basics of multiplication are the same in any mathematical systems; just multiply coefficients and sum over all those that have the same power. The optimal normal base implementation uses a precomputed lambda matrix to speed up the multiplication.

As mentioned in Chapter 5.4.2, there are two types of optimal normal bases over F^2_m, called Type I and Type II. The only implementation related difference between them is the way the bits in the lambda matrix are set. For Type I ONB only one vector is stored whereas for Type II ONB two vectors are stored [8].

The lambda vector for Type I ONB stores all the values of j for each value of *i* that satisfies the equation $2^i+2^j = 1 \bmod (m+1)$. The lambda matrix for Type II ONB is built by working with group of four equations. To build the lambda matrix, we have to find solutions to

$$2^i + 2^j = 1$$

$$2^i + 2^j = -1$$

$$2^i - 2^j = 1$$

$$2^i - 2^j = -1$$

The operation for field addition is implemented in the Field class. The rest of the operations (multiplication, squaring and inversion) needed in elliptic curve cryptosystem for optimal normal base fields are implemented in the *ONB* class.

Below are described all the classes which this package consist of:

Field

The Field class is the only abstract class in this implementation. The Field class contains methods for shifting bits in the field (shiftLeft, shiftRight, rotLeft, rotRight), adding field to another field (sum), and generating a random field (random). Field addition is a bitwise exclusive-or operation.

CustomField

The CustomField class implements custom sized fields. It contains methods for shifting bits in CustomFields, and a method to solve the equation $b = a \times u^n$ (cusTimes). The optimal normal base implementation uses the cusTimes method when executing the Schroeppel's Almost Inverse Algorithm. The cusTimes method consists of shift and exclusive-or operations. The Almost Inverse Algorithm is described in Chapter 6.3.1.

Curve

The *Curve* class implements basic elliptic curve operations. It has methods for doubling points (doublep), adding two points (sum), subtracting two points (sub), and multiplying a point with a scalar (mul) on a given curve. The formulae for adding and doubling points are from IEEE P1363 standard [56]. The arithmetic is described in Chapter 5.3. The scalar multiplication of points is described in Chapter 6.3.2.

Point

The *Point* class is a container for points on elliptic curves. This class contains only constructors and a set function that sets *Point's* coordinates.

Elliptic

The *Elliptic* class is a container for elliptic curve parameters. The parameters include the chosen elliptic curve to be used, the point order of the curve, and the base point of the chosen curve.

ONB

The *ONB* class implements optimal normal base fields over F_2^n. The genLambda method together with the init. Two methods create the lambda vectors described above. The field multiplication is implemented in the mul method, which uses the precomputed lambda vectors. Squaring a field element is implemented in the square method.

The inv method computes the inversion of a field element using the Schroeppel's Almost Inverse algorithm. The algorithm is described in detail in Chapter 6.3.2.

NR

The *NR* class implements the Nyberg-Rueppel signature scheme following the IEEE P1363 standard [56]. The Nyberg-Rueppel algorithm is described in Chapter 5.7. The sign and verify methods in the NybergRueppel class use *Elliptic*, *Field*, and *Point*. In addition, the signature scheme uses the SHA1 class to compute SHA-1 message digests.

Chapter 7. Test Results

It is useful to make the distinction between writing programs in a high-level programming language and writing programs in microcontroller-specific assembly language. When writing in a high-level language such as C or Java, the application programmer is typically presented with a well thought-out and thoroughly integrated set of services that have been explicitly designed to work together to ease the task of writing application software. One of the design goals of a good high-level programming interface is to provide help in dealing with the special considerations of smart card programming, such as those listed in this chapter.

When card software is written in assembly language, on the other hand, one can access data anywhere in memory and call upon any available entry point. Even when these entry points are part of public interfaces, they may be from different software providers and may, in fact, place calls on each other. For example, a call on a cryptographic routine may in turn generate a call on a communication routine, which in turn calls a memory management routine. It is up to the assembly language programmer to understand and abide by the rules assembly language routines must obey so that they work together successfully and don't step on each other's toes. Because space and time are at such a premium inside a smart card, smart card system software is much more tightly coupled than typical operating system software, so these programming rules are much more complex than most assembly language programmers are used to.

New applications for a smart card is another computing platform, but the smart card computer has a number of unique properties that must be kept in mind as these new applications are designed and built. In writing reader-side software there are few severe resource constraints of the smart card as a computing platform: low-speed communication, slow central processing unit, 8-bit data, and limited available memory. Furthermore, the smart card programmer has to

be constantly mindful of the context in which the computer carrying his or her application will be used and the nature of the systems to which it is connected.

Among all the special considerations that must be dealt with in writing card-side software are the unusual properties of the smart card memory system. Not only does the memory system consist of three different kinds of memory, each demanding to be dealt with in its own way, but also the properties of these memories can vary markedly from chip-to-chip and manufacturer-to-manufacturer.

The most difficult resource constraint for most card-side programmers to deal with is the limited amount of random access memory (RAM). Small cards have 128 bytes of RAM. Since it is such a limited resource, RAM is managed carefully and explicitly. RAM is pre-allocated to hold fixed, specific, often-used values that comprise the global state of the smart card and for general scratch use and can be used freely by code for temporary and intermediate values.

The first step in writing any card-side software is getting a detailed memory map of both the chip and the operating system for which you are writing and the other applications with which your code will be run.

Smart Cards have an EEPROM type of memory. This memory type is non-volatile and its contents are not erased by power loss. It's easy to understand the need for non-volatile memory on the card, as Smart Cards are externally powered and clocked power loss can happen very easily. A Smart Card should still be able to continue consistent operation in such a case after the power is restored.

EEPROM memory on a smart card is used to store values that are expected to remain on the card from use-to-use. Account numbers, digital certificates, passwords, private keys are all examples of data that would be kept in non-volatile memory.

Debugging and testing of card-side software is preceded in three phases:

1. Simulation. During the simulation phase, smart card program is run in its software development environment typically on a workstation or Windows PC. Calls to the smart card API are simulated, including the effects of these calls on a file system or to a communication channel. Returned values faithfully reflect the result of the call. Simulation environment allows single-step through program at the source code level.
2. Emulation. During the emulation phase, code is downloaded into the actual chip that will be in the target smart card, which is mounted on a personality board of an in-circuit emulator. Emulation environments let you single-step through your program at the assembly language level. But emulation is close to actual execution.
3. Integration. The final step in card software testing and debugging is to connect all the parts and components together and run the whole system just as it would be run in live use.

The main problem is how to fit the functioning code into a little over 16 KB of memory, and have enough space for keys. The idea was to find methods, which could be used in porting the existing ECC implementation to the Java Card platform. The existing implementation is based on the one from Michael Rosing's book [8]. The only problem with it is memory allocation. The original code is written in C, where calls to an object's method always modify the object

itself; new memory is not necessarily allocated at all. In Java, the value of the object itself can never be modified. Each call to an objects' method allocates and returns a new temporary object, whose reference is assigned to a user defined variable. So, the main idea behind memory allocation on Java Card is reserve-and-reuse.

To minimize the number of temporary variable allocations in a Java Card implementation the register allocation problem and its solution, which is used in compiler design [17], where used. BigInteger math methods was a basic block. Next they were analysed to see which methods call each other. Based on this, a variable interference graph was constructed, which was "colored" using a minimal set of temporary variables. By using the above method the amount of temporary variables were reduced by half. The result is replacing each local scope variable with a global one.

In the ECC implementation the public key consists of the point $\beta = a\alpha$, where α is the generating point of the curve. The private key consists of the field a. The system parameters needed for elliptic curve cryptosystem are the following: Field size, curve parameters a_2 and a_6, generating point α, order of the generating point. Examples of using parameters can be found in Appendix 2.

Implementing ECC on the JCOP20 is a difficult task. The processor memory architecture can be found in Table 7.1.

Type of Memory	Start address	Size	Max Address
RAM	0x0	0x500	0x4ff

ROM	0x1000	0x8000	0x8fff
EEPROM	0x9020	0x3fe0	0xcfff

Table 7.1 Memory architecture

Each curve point in a group occupies 64 bytes of EEPROM, 32 bytes each for the X and Y coordinates. A total of four 32 byte coordinate locations are used, starting from address 0x9020 to 0x90A0 in EEPROM. Twenty bytes, located from 0x90A1 to 0x903Ah, are used to keep track of the curve points, storing the locations of each curve point. Using these pointers optimises algorithms that repeatedly call the group operation. The output of the previous step is used as an input to the next step, so instead of copying a resulting curve point from the output location to an input location, which involves using pointers to move 64 bytes around in RAM, we can simply change the pointer values and effectively reverse the inputs and outputs of the group operation.

EEPROM is a precious resource on a smart card. Each file in EEPROM takes up some extra administrative bytes besides the bytes you actually use. These overhead bytes describe the file, including its size, its type, and its access conditions. Code is placed in EEPROM and occupies 9.3 Kb starting from the address 0xA7B4 length 0x254B bytes.

Chapter 8. Conclusion

Because computers and networks are becoming so central to our lives in this digital age, many new security challenges are arising. This is the beginning of an era of full connectivity, both electronically and physically. Smart cards can facilitate this connectivity and other value added capabilities, while providing the necessary security assurances not available through other means.

On the Internet, smart cards increase the security of the building blocks: Authentication, Authorization, Privacy, Integrity, and Non-Repudiation. Primarily, this is because the private signing key never leaves the smart card so it's very difficult to gain knowledge of the private key through compromise of the host computer system.

In a corporate enterprise system, multiple disjointed systems often have their security based on different technologies. Smart cards can bring these together by storing multiple certificates and passwords on the same card. Secure email and Intranet access, dial-up network access, encrypted files, digitally signed web forms, and building access is all improved by the smart card.

In an Extranet situation, where one company would like to administer security to business partners and suppliers, smart cards can be distributed which allow access to certain corporate resources. The smart card's importance in this situation is evident because of the need for the strongest security possible when permitting anyone through the corporate firewall and proxy defences. When distributing credentials by smart card, a company can have a higher assurance that those credentials cannot be shared, copied, or otherwise compromised.

Some reasons why smart cards can enhance the security of modern day systems are:

- Smart cards enhance PKI - Public Key Infrastructure systems are more secure than password based systems because there is no shared knowledge of the secret. The private key need only be known in one place, rather than two or more. If the one place is on a smart card, and the private key never leaves the smart card, the crucial secret for the system is never in a situation where it is easily compromised. A smart card allows for the private key to be usable and yet never appear on a network or in the host computer system.
- Smart cards increase the Security of Password Based Systems - Though smart cards have obvious advantages for PKI systems, they can also increase the security of password-based systems. One of the biggest problems in typical password systems is that users write down their password and attach it to their monitor or keyboard. They also tend to choose weak passwords and share their passwords with other people. If a smart card is used to store a user's multiple passwords, they need only remember the PIN to the smart card in order to access all of the passwords. Additionally, if a security officer initialises the smart card, very strong passwords can be chosen and stored on the smart card. The end user need never even know the passwords, so that they can't be written down or shared with others.
- Two Factor Authentications - Security systems benefit from multiple factor authentications. Commonly used factors are: Something you know, something you have, something you are, and something you do. Password based systems typically use only the first factor, "something you know". Smart cards add an additional factor,

"something you have". Two factor authentication has proven to be much more effective than single because the "something you know" factor is so easily compromised or shared. Smart cards can also be enhanced to include the remaining two features. Prototype designs are available which accept a thumbprint on the surface of the card in addition to the PIN in order to unlock the services of the card. Alternatively, a thumbprint template, retina template, or other biometric information can be stored on the card, only to be checked against data obtained from a separate biometric input device. Similarly, "something you do" such as typing patterns, handwritten signature characteristics, or voice inflection templates can be stored on the card and be matched against data accepted from external input devices.

- Portability of Keys and Certificates - Public key certificates and web browsers and other popular software packages can utilize private keys but they in some sense identify the workstation rather than the user. The key and certificate data is stored in a proprietary browser storage area and must be export/imported in order to be moved from one workstation to another. With smart cards the certificate and private key are portable, and can be used on multiple workstations, whether they are at work, at home, or on the road. If the lower level software layers support it, they can be used by different software programs from different vendors, on different platforms, such as Windows, Unix, and Mac.

- Auto-disabling PINs Versus Dictionary Attacks - If a private key is stored in a browser storage file on a hard drive, it is typically protected by a password. This file can be "dictionary attacked" where commonly used passwords are attempted in a brute force

manner until knowledge of the private key is obtained. On the other hand, a smart card will typically lock itself up after some low number of consecutive bad PIN attempts, for example 10. Thus, the dictionary attack is no longer a feasible way to access the private key if it has been securely stored on a smart card.

- Non Repudiation - The ability to deny, after the fact, that your private key performed a digital signature is called repudiation. If, however, your private signing key exists only on a single smart card and only you know the PIN to that smart card, it is very difficult for others to impersonate your digital signature by using your private key. Many digital signature systems require "hardware strength Non Repudiation", meaning that the private key is always protected within the security perimeter of a hardware token and can't be used without the knowledge of the proper PIN. Smart cards can provide hardware strength Non Repudiation.

- Counting the Number of Private Key Usages - Smart card based digital signatures provide benefits over handwritten signatures because they are much more difficult to forge and they can enforce the integrity of the document through technologies such as hashing. Also, because the signature is based in a device that is actually a computer, many new benefits can be conceived of. For example, a smart card could count the number of times that your private key was used, thus giving you an accurate measure of how many times you utilized your digital signature over a given period of time.

However, there are some problems with smart cards.

Even though smart cards provide many obvious benefits to computer security, they still haven't caught on with great popularity. This is not only because of the prevalence,

infrastructure, and acceptability of magnetic stripe cards, but also because of a few problems associated with smart cards. Lack of a standard infrastructure for smart card reader/writers is often cited as a complaint. The major computer manufactures haven't until very recently given much thought to offering a smart card reader as a standard component. Many companies don't want to absorb the cost of outfitting computers with smart card readers until the economies of scale drive down their cost. In the meantime, many vendors provide bundled solutions to outfit any personal computer with smart card capabilities.

Lack of widely adopted smart card standards is often cited as a problem. The number of smart card related standards is high and many of them address only a certain market or only a certain layer of communications. This problem is lessening recently as web browsers and other mainstream applications are including smart cards as an option. Applications like these are helping to speed up the evolution of standards.

Many companies have designed and manufactured smart cards, which vary greatly in both the hardware they use and software development environments they provide. Java Card promises to make smart card programming easier, by introducing a common programming language and run-time environment. Also as a member of the Java family, Java Card raises hopes of easy software portability from PCs to smart card.

Sun Microsystems Java Card Kit was used to concentrate application functionality and correct API usage. This environment happened to be not the most reliable one. Often happen situations where code, which were executed before would stop working. System set up is difficult as well. Sun Microsystems provides a user guide but either the instructions are not always correct or they do not give a full description of how operations are performed.

Smart card have extremely rigid constrains on processing power, parameter storage, and code space, as well as slow input/output. As a result, implementation of public-key cryptosystem in smart cards was a very difficult task. In the workstation ECC is based on the Standard Java *BigInteger* library, which cannot be used on Java Card because of immutable semantics. Because the JDK version of *BigInteger* called an underlying native C library for its operations two problems appeared. First: Memory allocation. Second: Converting long number arithmetic functionality to the card. Both of these problems were addressed in 6.3.

Most of the memory of current Java Cards is EEPROM and manufacturer's documentation often doesn't even mention the amount of RAM that a specific Java Card has integrated on the chip. Current Java Card specifications state that objects are by default allocated in EEPROM. For performance specific implementations such as cryptography this quite an unfortunate choice as write operations to EEPROM are about 30 times slower than equivalent operations in RAM [7]. Furthermore, there is also a maximum number of writes that EEPROM memory can sustain before becoming non-functional. In the current Java Card memory model compiler and the card environment internally handle the placement of objects between RAM and EEPROM. As Java Cards have different amounts of memory this also means that objects that end up in RAM on some cards will end up in EEPROM on the others. Also a great deal of performance of the actual run time program depends on how well the cards additional compiler and environment is designed.

While I consider experience with Java Card valuable, if faced with a similar design problem today I would seriously consider using systems other that Java based cards.

Another part of this document discusses the Elliptic Curve Cryptography system. Mathematicians have given considerable attention to ECDLP. Like the other types of cryptographic problems, no efficient algorithm is known to solve the ECDLP. The ECDLP seems to be particularly harder to solve. Moderate security can be achieved with the ECC using an elliptic curve defined modulo a prime p that is several times shorter than 230 decimal digits.

An elliptic curve cryptosystem implemented over a 160-bit field currently offers roughly the same resistance to attack, as would a 1024-bit RSA modulus. Moderate security can be achieved with the ECC using an elliptic curve defined modulo a prime p that is several times shorter than 230 decimal digits [26].

At the security level of 10^{20} MIPS years, it takes a 300-bit key in ECC to equal the strength of a 2048 bit key in either RSA or DSA. The currently acceptable security level is 10^{12} MIPS years. The security gap between the systems grows as the key size increases [40].

In general, no major weaknesses with ECC have been discovered. However, it has been one reported that a 108-bit elliptic curve encryption key was cracked in July 2000. It took 9,500 computers running in parallel for four months, connected via the Internet. It would take 500 years of processing on a single 450 MHz personal computer to perform the same key cracking. [79]

There have been weak classes of elliptic curves identified such as supersingular elliptic curves and some anomalous elliptic curves. Implementations, such as ECDSA, merely check for weaknesses and eliminate any possibility of using these "weak" curves. [40]

The advantages of ECC can be described as followed:

- ECC leads to more efficient implementations than other public-key systems due to its extra strength provided by the difficulty to solve the ECDLP.
- Key size. A typical key size for the RSA algorithm is 1024 bits; which would take approximately 10^{11} MIPS years to break. In comparison, an ECC key size is 160 bit offers the same level of security [45].
- Computational efficiencies are achieved with ECC. ECC does not require processing of prime numbers to achieve encryption unlike other public-key cryptosystems. ECC is roughly 10 times faster than either RSA or DSA [26].
- ECC offers considerable bandwidth savings over the other types of public-key cryptosystems when being used to transform short messages such as the typical implementation of ECDSA. Bandwidth savings is about the same as other public-key cryptosystems when transforming long messages [26].

These advantages lead to higher speeds, lower power consumption, and code size reductions. Implementations of ECC are particularly beneficial in applications where bandwidth, processing capacity, power availability, or storage is constrained. Such applications include wireless transactions, handheld computing, broadcast, and smart card applications.

Disadvantages of ECC include:

- Hyperelliptic cryptosystems offer even smaller key sizes [82].
- ECC is mathematically subtler than RSA or SDL.

Many things still can be done for implementing cryptographic operations on Java Card. A more widely usable implementation than the prototype offered can be produced where speed of the routines must be optimised. One way to do it is to select and implement better routines for F_2^n arithmetic. Second approach is to use optimal extension fields to make calculations faster. Further, hyperelliptic curves and public key cryptography can be used. However, hyperelliptic curves mathematically are even more challenging than elliptic curves and have not yet been completely standardised.

References

[1] Chen, Z., 2000. Java Card Technology for Smart cards, Architecture and Programming Guide. London: Addison-Wesley.

[2] Dreifus, H., Monk, J., T., 1998. Smart Cards. A Guide to Building and Maintaining Smart Card Applications. New-York: Jhon Wiley & Sons Inc.

[3] Goldrich, O., 1999. Foundations of Cryptography, Basic Tools. USA: Springer-Verlag.

[4] Hansmann, U., Nicklous, M. S., Schack T., Seliger, F., 2000. Smart Card Application Development Using Java. London: Springer

[5] Koblitz, N., 1998. A Course in Number Theory and Cryptography. 2nd ed. London: Springer

[6] Menezes, A., Oorschot, P., Vanstone, S. 1996. Handbook of Applied Cryptography. Canada: CRC Press.

[7] Rankl, W., Effing, W., 2000. Smart Card Handbook. 2nd ed. Munich, Germany: Giesecke & Devrient GmbH.

[8] Rosing, M., 1998. Implementing Elliptic Curve Cryptography. Greenwich, USA: Manning Publications Co.

[9] Schneier B., 1996. Applied Cryptography. 2nd ed. New-York: Jhon Wiley & Sons Inc.

[10] Bao, F, 1998, An Efficient Verifiable Encryption Scheme for Encryption of Discrete Logarithms. In: Quisquater J-J., Schneier, B., ed. Third International Conference CARDIS'98 Louvain-la-Neuve Belgium 14-16 September 1998. London: Springer, 213-220.

[11] Durand, A., 1998, Efficient Ways to Implement Elliptic Curve Exponentiation on a Smart Card. In: Quisquater J-J., Schneier, B., ed. Third International Conference CARDIS'98 Louvain-la-Neuve Belgium 14-16 September 1998. London: Springer, 357-365.

[12] Handschuh, H., Paillier, P., 1998, Smart Card Crypto-Coprocessors for Public-Key Cryptography. In: Quisquater J-J., Schneier, B., ed. Third International Conference CARDIS'98 Louvain-la-Neuve Belgium 14-16 September 1998. London: Springer, 372-380.

[13] Hankerson, D, Hernandez, J., L., Menezes, A., 1998, Software Implementation of Elliptic Curve Cryptography over Binary Fields. In: Quisquater J-J., Schneier, B., ed. Third International Conference CARDIS'98 Louvain-la-Neuve Belgium 14-16 September 1998. London: Springer, 218-234.

[14] Lim, C. H., Hwang, H. S., 1998, Fast Implementation of Elliptic Curve Arithmetic in $GF(p^n)$. In: Yang J-D., Seo, C-C., ed. Information and Communication Research Center, Seoul

[15] Paar, C, Soria-Rodriguez, P., 1997, Fast Arithmetic Architectures for Publick-Key Algorithms over Galois Fields $GF((2^n)^m)$. In: Quisquater J-J., Schneier, B., ed. EUROCRYPT'97 Worcester 1997. New-York: Springer-Verlag, 363-378.

[16] Vandewalle, J-J., Vetillard, E., 1998, Developing Smart Card-Based Applications Using Java Card. In: Quisquater J-J., Schneier, B., ed. Third International Conference CARDIS'98 Louvain-la-Neuve Belgium 14-16 September 1998. London: Springer, 105-124.

[17] Woodbury, A. D., Bailey, D. V., Paar, C., 2000, Elliptic Curve Cryptography on Smart Card without Coprocessor. In: The Fourth Smart Card Research and Advanced Applications (CARDIS 2000) Conference, 20-22 September 2000, Bristol, UK.

[18] Beauregard, D., 1996. Efficient Algorithms for Implementing Elliptic Curve Publick-Key Schemes. Thesis (MSc). Worcester Polytechnic Institute.

[19] Chan, C., 2000. An Overview of Smart Card Security. Thesis (MSc). Queensland University of Technology.

[20] Pietilainen, H., 2000. Elliptic curve cryptography on smart cards. Thesis (MSc). Helsinki University of Technology.

[21] Barwood, G., (1997), Elliptic Curve Cryptography FAQ v1.12 [online]. Available from http://ds.dial.pipex.com/george.barwood/ec_faq.txt [Accessed 2 September 2002]

[22] Bassham, L., Johnson, D., Polk, W. (1999), Presentation of Elliptic Curve Digital Signature Algorithm. Key and Signatures in Internet X.509 Public Key Infrastructure

Certificate [online]. Available from http://www.ietf.org/ietf/1id-abstracts.txt. [Accessed 20 February 2001]

[23] Berg, J., Jacobs, B., Poll, E. (2000), Formal Specification and Verification of Java Card's Application Identifier Class [online]. Department of Computer Science, University Nijmegen, Netherlands. Available from: http://www.cs.kun.nl/~{joackim,bart,erikpoll}

[24] Grabbe, J., O., (1997), Cryptography and Number Theory for Digital Cash [online]. Available from [Accessed 17 July 2000]

[25] Hasegawa, T., (1999). A Small and Fast Software Implementation of Elliptic Curve Cryptosystems over GF(p) on a 16-bit Microcomputer. IEICE Fundamentals [online], E82-A (1). Available from: [Accessed 5 March 2003]

[26] Jacobs, L., (2001), Elliptic Curve Cryptosystem – An Overview [online]. Available from http://www.sans.org/rr/encryption/encryption_list.php [Accessed 19 November 2002]

[27] Johnson, D., Menezes, A.(2000), The Elliptic Curve Digital Signature Algorithm [online]. Technical Report CORR. Dept of C&O, University of Waterloo, Canada. Available from: http://www.cacr.math.uwaterloo.ca [Accessed 20 February 2003]

[28] Ognibene, P., J., (1997), Smart Card Development Services [online]. Available from http://members.aol.com/pjsmart/index.htm [Accessed 14 May 2001]

[29] Ort, E., (2001), Writing a Java Card Applet, Java Developer Connection [online]. Available from: http://developer.java.sun.com/developer/technicalArticles/ [Accessed 12 November 2001]

[30] Petri, S. (1999), Smart Card Solutions in the Real World [online]. Litronic, Inc. Available from http://www.litronic.com/whitepaper [Accessed 18 July 2000]

[31] Chen, Z., (1998), Understanding Java Card 2.0. Java World [online]. jw-03-1998. Available from: http://www.javaworld.com/javaworld/jw-03-1998/jw-03-javadev_p.html [Accessed 7 May 2001]

[32] Chen, Z., (1999), How to Write a Java Card Applet: a Developer's Guide. Java World [online]. jw-07-1999. Available from: http://www.javaworld.com/javaworld/jw-07-1999/jw-01-javadev_p.html [Accessed 21 June 2001]

[33] Giorgio, R. (1997), Smart Cards: A Primer. Java World [online]. jw-12-1997. Available from: http://www.javaworld.com/javaworld/jw-12-1997/jw-01-javadev_p.html [Accessed 21 June 2001]

[34] Giorgio, R. (1998), Smart Cards and the OpenCard Framework. Java World [online]. jw-01-1998. Available from: http://www.javaworld.com/javaworld/jw-01-1998/jw-01-javadev_p.html [Accessed 21 June 2001]

[35] Giorgio, R., Montgomery, M. (1999), Write Open Card Services for Downloading Java Card Apps. Java World [online] jw –02-1999. Available from: http://www.javaworld.com/javaworld/jw-02-1999/jw-01-javadev_p.html [Accessed 21 June 2001]

[36] Mysore S., H., Giorgio, R., (1998), Java Gets Serial Support With a New javax.comm. Package, Java World [online]. jw-05-1998. Available from: http://www.javaworld.com/javaworld/jw-05-1998/jw-05-javadev_p.html [Accessed 21 November 2002]

[37] Schaeck, T., (1998), How to Write OpenCard Services for Java Card Applets. Java World [online]. jw-10-1999. Available from: http://www.javaworld.com/javaworld/jw-10-1998/jw-01-javadev_p.html [Accessed 21 June 2001]

[38] Wendler, M., Breideneich, S., Giorgio, R., (1999), How to Write a Card Terminal Class for Simple and Complex Readers in an OpenCard Environment [online] jw –01-1999. Available from: http://www.javaworld.com/javaworld/jw-01-1999/jw-01-javadev_p.html [Accessed 7 September 2001]

[39] Certicom. ECC Standards. http://www.certicom.com/research/ECCstandards.html

[40] Certicom. 1997. Current Public-Key Cryptographic Systems, http://www.certicom.com/research/wecc2.html

[41] Certicom. 1998. The Elliptic Curve Cryptosystems for Smart Cards. http://www.certicom.com/research/wecc4.html

[42] Certicom. 1999 ECC in X.509. http://www.secg.org/drafts.htm

[43] Certicom. 1999. Standards for Efficient Cryptography, SEC 1: Elliptic Curve Cryptography. http://www.secg.org/drafts.htm

[44] Certicom. 1999. Standards for Efficient Cryptography, SEC 2: Recommended Elliptic Curve Domain Parameters. http://www.secg.org/drafts.htm

[45] Certicom, 2000. Remarks on Security of Elliptic Curve Cryptosystem. Ontario: Certicom Corp.

[46] Hewlett Packard, 1998. Compact Representation of Elliptic Curve Points Over F_2^n. USA: Hewlett Packard. (HPL-98-94).

[47] IEEE-SA Standards Board, 2000. IEEE Standard Specifications for Public-Key Cryptography. New York: IEEE, Inc.

[48] IBM, 2000. OpenCard Framework 1.2 Programmer's Guide. USA: IBM. (BOEB-OCFP-00).

[49] Java Card Forum, 2000. Java Card Management Specification. USA: Java Card Forum.

[50] National Institute of Standards and Technology, 1995. Secure Hash Standard. Gaithersburg: US Department of Commerce.

[51] National Institute of Standards and Technology, 2002. Government Smart Card Interoperability Specification. USA: NIST.

[52] Sun Microsystems, 1997. Java Cryptography Architecture, API specification & Reference, Palo Alto, USA: SUN Microsystems.

[53] Sun Microsystems, 1998. Java Card Applet Developers Guide, Palo Alto, USA: SUN Microsystems.

[54] Sun Microsystems, 2000. Java Card 2.1.1 Application Programming Interface. Palo Alto, USA: SUN Microsystems.

[55] CDSA: http://developer.intel.com/ial/security/

[56] IBM: http://www.zurich.ibm.com/

[57] Infosysses: http://www.infosyssec.net/index.html

[58] Javacard: http://www.javasoft.com/products/javacard/index.html

[59] Maxking: http://www.maxking.com

[60] Mondex: http://www.mondexusa.com/

[61] Opencard: http://www.opencard.org/

[62] Proton: http://www.protonworld.com/

[63] VisaCash: http://www.visa.com/cgi-bin/vee/nt/chip/main.html?2+0

[64] Versatile Card Technology: http://www.versacard.com/VCT/index.html

[65] Коблиц, Н. Курс теории чисел и криптографии М., Научное издательство ТВП, 2001 г.

[66] Ростовцев, А.Г. 1999, О выборе эллиптической кривой над простым полем для построения криптографических алгоритмов. журнал Проблемы информациоиной безопасности. Компьютерные системы, N3.

[67] Ministry of Education of Russia, 2000, Алгоритмические основы эллиптической криптографии (Elliptic Curve Cryptosystem), Moscow: Russia.

[68]

[69] RSA Security PKCS: http://www.rsasecurity.com

[70] PC/SC Workgroup: http://www.pcscworkgroup.com

[71] Kocher, P., Jaffer, J., Jun, B., Differential Power Analiysis [online], Cryptography Research Inc. Available from: http://www.cryptography.com/resources/whitepapers/DPA.pdf

[72] Microsoft Cryptographic API: http://www.microsoft.com/mind/0697/crypto.asp

[73] Dataquest (1999), Smart Success: Dataquest. The Business of InfoTech [online]. March 15. Available from

http://www.dqindia.com/content/search/showarticle.asp?arid=16576&way=search [Accessed 16 September 2000]

[74] Dataquest (2001), Smarter Cards: Dataquest. The Business of InfoTech [online]. December 09. Available from http://www.dqindia.com/content/search/showarticle.asp?arid=119710&way=search [Accessed 10 April 2002]

[75] Cryptography Research: http://www.cryptography.com

[77] Sun Microsystems, 2000. Java Card 2.1 Virtual Machine. Palo Alto, USA: SUN Microsystems.

[78] Chip Operating Systems: http://iris.com.my/Technology/tech_oc.asp

[79] "108-Bit Elliptic-Curve Encryption Key Cracked", http://www.nikkeibp.asiabiztech.com/nea/200007/late_106440.html

[80] Wagner, L., Algebro-Geometric Attack Methods in Elliptic Curve Cryptography, Department of Mathematics, St John's College, The University of Queensland

[81] Okeya, K., Takagi, T., The Width-w NAF Method Provides Small Memory and Fast Elliptic Scalar Multiplications Secure against Side Channel Attacks, Hitachi, Ltd., Systems Development Laboratory, Yokohama: Japan

[82] Lange, T., Hyperelliptic Curves Allowing Fast Arithmetic, Institute of Information Security and Cryptography, Bochum: Germany. http://www.ruhr-uni-bochum.de/itsc/tanja/KoblitzC.html

Appendix A. Introduction to Basic Mathematical Terminology

What is a group?

A group is a set of numbers with a custom-defined arithmetic operation. The unique rules for arithmetic in groups are a source of the hard problems necessary for cryptographic security. Two groups used in cryptography are Z_n, the additive group of integers modulo a number n; and Z_p^*, the multiplicative group of integers modulo a prime number p.

The group Z_n

The group $Z_n$uses only the integers from 0 to n - 1. Its basic operation is addition, which ends by reducing the result modulo ^{n}; that is, taking the integer remainder when the result is divided by n. One very important feature of arithmetic in a group is that all calculations give numbers, which are in the group; this is called closure. Modular reduction by n ensures that all additions result in numbers between 0 and n - 1.

Additive Inverses

Each number x in an additive group has an additive inverse element in the group; that is an integer -x such that x + (-x) = 0 in the group.

Other operations

While addition is the main operation in the additive group Z_n, other operations can be derived from addition. For example, the subtraction x - y can be performed as the addition x + (-y) mod n.

It is also possible to define multiplication in Zn by repeated addition.

The group Z_p*

Cryptosystems using arithmetic in Z_p* include the Diffie-Hellman Key Agreement Protocol and the Digital Signature Algorithm (DSA).
The multiplicative group Z_p* uses only the integers between 1 and p - 1 (p is a prime number), and its basic operation is multiplication. Multiplication ends by taking the remainder on division by p; this ensures closure.

Multiplicative Inverses

Each number x in a multiplicative group has a multiplicative inverse element in the group; that is an integer *x^{-1}* such that *x x^{-1} = 1* in the group.
In a multiplicative group, each element must have a multiplicative inverse.

Abelian Groups

An arithmetic operation is said to be commutative if the order of its arguments is insignificant. With ordinary numbers, addition and multiplication are commutative operations;

A group is called abelian if its main operation is commutative. Thus an additive group is abelian if $a + b = b + a$ for all elements a, b in the group. A multiplicative group is abelian if $a \times b = b \times a$ for all elements a, b in the group. The additive group Z_n and the multiplicative group Z_p^* are both abelian groups.

What is a field?

A field is a set of elements with two custom-defined arithmetic operations: most commonly, addition and multiplication. The elements of the field are an additive abelian group, and the non-zero elements of the field are a multiplicative abelian group. This means that all elements of the field have an additive inverse, and all non-zero elements have a multiplicative inverse. As is true for groups, other operations can be defined in a field, using its main two operations. A field is called *finite* if it has a finite number of elements. The most commonly used finite fields in cryptography are the field F_p (where p is a prime number) and the field F_2^m.

The field F_p

The finite field F_p (p a prime number) consists of the numbers from 0 to p - 1. Its operations are addition and multiplication, which are defined as for the groups Z_n and Z_p^* respectively: all calculations end with reduction modulo p. The restriction that p be a prime number is necessary so that all non-zero elements have a multiplicative inverse (see Z_p^* for details). As with Z_n and Z_p^*, other operations in F_p (such as division, subtraction and exponentiation) are derived from the definitions of addition and multiplication.

The field F_{2m}

Although the description of the field F_{2m} is complicated, this field is extremely useful because its computations can be done efficiently when implemented in hardware. There are several ways to describe arithmetic in F_{2m}; both *polynomial representation* and *optimal normal basis representation* are described.

Polynomial Representation

The elements of F_2^m are polynomials of degree less than m, with coefficients in F_2; that is, $\{a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \ldots + a_2x^2 + a_1x + a_0 \mid a_i = 0 \text{ or } 1\}$. These elements can be written in vector form as $(a_{m-1} \ldots a_1\ a_0)$. F_2^m has 2^m elements.

The main operations in F_2^m are addition and multiplication. Some computations involve a polynomial $f(x) = x^m + f_{m-1}x^{m-1} + f_{m-2}x^{m-2} + \ldots + f_2x^2 + f_1x + f_0$, where each f_i is in F_2. The polynomial $f(x)$ must be irreducible; that is, it cannot be factored into two polynomials over F_2, each of degree less than m.

Addition

$(a_{m-1} \ldots a_1\ a_0) + (b_{m-1} \ldots b_1\ b_0) = (c_{m-1} \ldots c_1\ c_0)$ where each $c_i = a_i + b_i$ over F_2. Addition is just the componentwise XOR of $(a_{m-1} \ldots a_1\ a_0)$ and $(b_{m-1} \ldots b_1\ b_0)$.

Subtraction

In the field F_2^m, each element $(a_{m-1} \ldots a_1\ a_0)$ is its own additive inverse, since $(a_{m-1} \ldots a_1\ a_0) + (a_{m-1} \ldots a_1\ a_0) = (0 \ldots 0\ 0)$, the additive identity. Thus addition and subtraction are equivalent operations in F_2^m.

Multiplication

$(a_{m-1} \ldots a_1\ a_0)\ (b_{m-1} \ldots b_1\ b_0) = (r_{m-1} \ldots r_1\ r_0)$ where $r_{m-1}x^{m-1} + \ldots + r_1x + r_0$ is the remainder when the polynomial $(a_{m-1}x^{m-1} + \ldots + a_1x + a_0)\ (b_{m-1}x^{m-1} + \ldots + b_1x + b_0)$ is divided by the polynomial $f(x)$ over F_2. (Note that all polynomial coefficients are reduced modulo 2.)

Exponentiation

The exponentiation $(a_{m-1} \ldots a_1\ a_0)^e$ is performed by multiplying together e copies of $(a_{m-1} \ldots a_1\ a_0)$.

Multiplicative Inversion

There exists at least one element g in F_2^m such that all non-zero elements in F_2^m can be expressed as a power of g. Such an element g is called a *generator* of F_2^m. The multiplicative inverse of an element $a = g^i$ is $a^{-1} = g^{(-i) \bmod (2m-1)}$.

Optimal Normal Basis Representation

For many values of m, the finite field F_2^m has an *optimal basis representation* as well as the polynomial representation described above. An optimal basis gives an alternative way of defining multiplication on the elements of a field. While optimal normal basis multiplication is less insightful than polynomial multiplication, it is in practice much more efficient.

Setup for Multiplication

Optimal normal basis representations are classified as either Type I or Type II. The value of m determines which type shall be used.

1. If F_2^m only has a type I ONB then let $f(x) = x^m + x^{m-1} + \ldots + x^2 + x + 1$. Otherwise, if F_2^m has a type II ONB then compute $f(x) = f_m(x)$ using the following recursive formulae:

$f_0(x) = 1$,

$f_1(x) = x + 1$,

$f_{i+1}(x) = xf_i(x) + f_{i-1}(x)$, $i = 1, \ldots, m$.

At each stage, the coefficients of the polynomials $f_i(x)$ are reduced modulo 2; hence $f(x)$ is a polynomial of degree m with coefficients in F_2. The set of polynomials $\{x, x^2, x^{22}, \ldots, x^{2m-1}\}$ forms a basis of F_2^m over F_2, called a normal basis.

2. Construct the m by m matrix A whose i^{th} row, $i = 0 \ldots m - 1$, is the bit string corresponding to the polynomial $x^{2^{\wedge}i} \bmod f(x)$. (The rows and columns of A are indexed by the integers from 0 to m - 1.) The entries of A are elements of F_2.

3. Determine the inverse matrix A^{-1} of A over F_2.

4. Construct the m by m matrix T' whose i^{th} row, $i = 0 \ldots m - 1$, is $x\, x^{2^{\wedge}i} \bmod f(x)$. Then compute the matrix $T = T'A^{-1}$ over F_2.

5. Determine the product terms l_{ij}, for $i, j = 0 \ldots m - 1$, as $l_{ij} = T(j-i,-i)$. ($T(g,h)$ denotes (g,h)-entry of T with indices reduced modulo m.) Each product term l_{ij} is an element of F_2. It should also be the case that $l_{0j} = 1$ for precisely one j, $0 <= j <= m - 1$, and that for each i, $0 <= i <$

$= m - 1$, $l_{ij} = 1$ for precisely two distinct $j, 0 <= j <= m - 1$. *Hence only 2m-1* of the m^2 entries of the matrix T are *1*, the rest being *0*. This scarcity of 1's is the reason that the normal basis is called an optimal normal basis.

Multiplication

Multiplication is defined by $(a_0\ a_1\ a_2\ a_3)\ (b_0\ b_1\ b_2\ b_3) = (c_0\ c_1\ c_2\ c_3)$, where

$c_0 = a_0b_2 + a_1(b_2 + b_3) + a_2(b_0 + b_1) + a_3(b_1 + b_3)$

$c_1 = a_1b_3 + a_2(b_3 + b_0) + a_3(b_1 + b_2) + a_0(b_2 + b_0)$

$c_2 = a_2b_0 + a_3(b_0 + b_1) + a_0(b_2 + b_3) + a_1(b_3 + b_1)$

$c_3 = a_3b_1 + a_0(b_1 + b_2) + a_1(b_3 + b_0) + a_2(b_0 + b_2)$.

Exponentiation using Optimal Normal Bases

The squaring $(a_0\ a_1\ a_2\ a_3)^2 = (a_0\ a_1\ a_2\ a_3)\ (a_0\ a_1\ a_2\ a_3) = (c_0\ c_1\ c_2\ c_3)$, where

$c_0 = a_0a_2 + a_1(a_2 + a_3) + a_2(a_0 + a_1) + a_3(a_1 + a_3) = a_3^2 = a_3$

$c_1 = a_1a_3 + a_2(a_3 + a_0) + a_3(a_1 + a_2) + a_0(a_2 + a_0) = a_0^2 = a_0$

$c_2 = a_2a_0 + a_3(a_0 + a_1) + a_0(a_2 + a_3) + a_1(a_3 + a_1) = a_1^2 = a_1$

$c_3 = a_3a_1 + a_0(a_1 + a_2) + a_1(a_3 + a_0) + a_2(a_0 + a_2) = a_2^2 = a_2$.

Appendix B. Elliptic Curve Parameters

Field size: 133

Elliptic Curve E: $y^2 + xy = x^3 + a_2x^2 + a^6$ over $GF(2^{133})$

Curve coefficient a_2:

07 A11B09A7 6B562144 418FF3FF 8C2570B8

Curve coefficient a_6:

02 17C05610 884B63B9 C6C72916 78F9D341

Base point α:

0300 81BAF91F DF9833C4 0F9C1813 43638399

Order of base point α:

04 00000000 00000002 3123953A 9464B54D

Field degree n:

131

Cofactor K:

02

Reference: [68]

Field size: 133

Elliptic Curve E: $y^2 + xy = x^3 + a_2x^2 + a^6$ over $GF(2^{133})$

Curve coefficient a_2:

03 E5A88919 D7CAFCBF 415F07C2 176573B2

Curve coefficient a_6:

04 B8266A46 C55657AC 734CE38F 018F2192

Base point α:

0303 56DCD8F2 F95031AD 652D2395 1BB366A8

Order of base point α:

04 00000000 00000000 6954A233 049BA98F

Field degree n:

131

Cofactor K:

02

Reference: [68]

```
package example;

public class field
{
    public final static /*short*/ int NUMBITS   =  113;
    public final static /*short*/ int TYPES2    =  1;
    public final static /*short*/ int field_prime = 227;

    public final static /*short*/ int WS = 0x10;
    public final static /*short*/ int NUMWORD = 0x7;
    public final static /*short*/ int UPRSHIFT = 0x1;
    public final static /*short*/ int MAXLONG = 0x8;
    public final static /*short*/ int MAXBITS = 0x80;
    public final static /*short*/ int MAXSHIFT = 0xF;
    public final static /*short*/ int LONGWORD = 0xE;
    public final static /*short*/ int LONGSHIFT = 0x2;
    public final static /*short*/ int MSB       =  0x80;
    public final static /*short*/ int UPRBIT    =  0x1;
    public final static /*short*/ int UPRMASK   =  0x1;
    public final static /*short*/ int LONGBIT   =  0x2;
    public final static /*short*/ int LONGMASK  =  0x3;

    public static /*short*/ int[] FIELD = new /*short*/ int [MAXLONG];

    public static /*short*/ int[] CUSTFIELD = new /*short*/ int [LONGWORD + 1];

    //Curve
    public static /*short*/ int[] a2 = new /*short*/ int [MAXLONG];
    public static /*short*/ int[] a6 = new /*short*/ int [MAXLONG];
    public static /*short*/ int form;

    //Point
    public static /*short*/ int[] x = new /*short*/ int [MAXLONG];
    public static /*short*/ int[] y = new /*short*/ int [MAXLONG];

    //Signature
    public static /*short*/ int[] c = new /*short*/ int [MAXLONG];
    public static /*short*/ int[] d = new /*short*/ int [MAXLONG];

    // EC parameters
    public static /*short*/ int[] crv = new /*short*/ int [MAXLONG];
    public static /*short*/ int[] pnt = new /*short*/ int [MAXLONG];
    public static /*short*/ int[] pnt_order = new /*short*/ int [MAXLONG];
    public static /*short*/ int[] cofactor = new /*short*/ int [MAXLONG];

    // EC keys
    public static /*short*/ int[] prvt_key = new /*short*/ int [MAXLONG];
    public static /*short*/ int[] pblc_key = new /*short*/ int [MAXLONG];
}
```

```
package example;

import example.BigInteger;
import example.elliptic;
import example.Onb;
import example.field;
import example.protocols;

public class example
{
  long random_seed;
  public static void sha_memory();


  public static int int_onecmp(BigInteger number)
  {
    int        i;

    if ( number.BIGINT[BigInteger.INTMAX] > 1)
      return (0);
    for ( i=0; i<BigInteger.INTMAX; i++)
    {
      if (number.BIGINT[i] != 0)
        return (0);
      if (number.BIGINT[BigInteger.INTMAX] != 0)
        return (1);
    }

    return 1;
  }

 /*  Generate a key pair, a random value plus a point */

 public static void ECKGP( field Base, field Key)
 {
   BigInteger key_num = new BigInteger();
   BigInteger point_order = new BigInteger();
   BigInteger quotient = new BigInteger();
   BigInteger remainder = new BigInteger();
   field rand_key = new field();

   /*  ensure random value is less than point order  */
   protocols.random_field( rand_key.FIELD);
   BigInteger.field_to_int( rand_key.FIELD, key_num.BIGINT);
   BigInteger.field_to_int( Base.pnt_order, point_order.BIGINT);
   BigInteger.int_div( key_num.BIGINT, point_order.BIGINT, quotient.BIGINT, remainder.BIGINT);
   BigInteger.int_to_field( remainder.BIGINT, Key.prvt_key);

   elliptic.elptic_mul( field.prvt_key, field.pnt, field.pblc_key, field.crv);
 }

 public static void hash_to_int( char[] Message, long length, BigInteger hash_value)
 {
   long[] message_digest = new long[5];          /*  from SHA-1 hash function  */
   field mdtemp = new field();                   /*  convert to NUMBITS size (if needed)  */
   int        i, count;

   sha_memory(        Message, length, message_digest);

   /*  convert message digest into an integer */

   Onb.nul (mdtemp);
```

```
   count = 0;
   for(i = 0; i<field.MAXLONG; i++)
   {
     mdtemp.e[ field.NUMWORD - i] = message_digest [ 4 - i];
     count++;
     if (count > 4) break;
   }
   mdtemp.e[0] &= field.UPRMASK;
   field_to_int ( mdtemp, hash_value );
 }

/*  Nyberg-Rueppel elliptic curve signature scheme.

        Inputs: pointer to Message to be signed and its length,
                        pointer to elliptic curve parameters,
                        pointer to signer's secret key,
                        pointer to signature storage area.

        Output: fills signature storage area with 2 numbers
                        first number = SHA(Message) + random value
                        second number = random value - signer's secret key times first number
                                        both are done modulo base point order

                        The output is converted back to FIELD2N variables to save space
                        and to make verification easier.*/

  public static void NR_Signature ( char[] Message, long length, field public_curve, field secret_key,
field signature)
  {
    BigInteger         hash_value  = new BigInteger();
    field random_value  = new field();
    field random_point  = new field();
    field x_value = new field();
    field k_value = new field();
    field sig_value  = new field();
    field temp = new field();
    field quotient = new field();
    field key_value = new field();
    field point_order = new field();
    int i, count;

    /*  compute hash of input message  */

    hash_to_int ( Message, length,  temp);
    BigInteger.field_to_int ( public_curve.pnt_order, point_order.FIELD);
    BigInteger.int_div ( temp.FIELD, point_order.FIELD, quotient.FIELD, hash_value.BIGINT);

    /*  create random value and generate random point on public curve  */

    protocols.random_field ( random_value );
    elliptic.elptic_mul ( random_value, public_curve.pnt, random_point, public_curve.crv);

    /*  convert x component of random point to an integer and add to message
        digest modulo the order of the base point.*/

    BigInteger.field_to_int ( random_point.x, x_value);
    BigInteger.int_add ( x_value, hash_value, temp);

    BigInteger.int_div ( temp, point_order, quotient, sig_value);
    BigInteger.int_to_field ( sig_value, signature.c);

    /*  final step is to combine signer's secret key with random value
```

```
                second number = random value - secret key * first number
                modulo order of base point*/

    BigInteger.field_to_int ( random_value , k_value);
    BigInteger.field_to_int ( secret_key , key_value);
    BigInteger.int_mul ( key_value , sig_value , temp);
    BigInteger.int_div ( temp, point_order , quotient , sig_value);

    BigInteger.int_sub ( k_value , sig_value , sig_value);
    while ( (sig_value.hw[0] & 0x8000) == 1)
      BigInteger.int_add ( point_order , sig_value , sig_value);
    BigInteger.int_div ( sig_value , point_order , quotient , temp);
    BigInteger.int_to_field ( sig_value , signature.d);
  }

/*  verify a signature of a message using Nyberg-Rueppel scheme.

        Inputs:         Message to be verified of given length,
                        elliptic curve parameters public_curve
                        signer's public key (as a point),
                        signature block.

        Output: value 1 if signature verifies,
                        value 0 if failure to verify.
*/


 public static void main(String[] argv)
 {
   field         Base = new field();
   field         Signer = new field();
   field         signature = new field();
   BigInteger  prime_order = new BigInteger();
   field         temp = new field();
   int         i, error;
   char[]         Message = new char[20];

   char string1[] = new char[MAXSTRING];

   string1 = "51922968585348276278" ; /*N 113  */



   elliptic.init_opt_math ();

   protocols.random_seed = 0xFEEDFACE;

   /*  compute curve order from Koblitz data  */

   BigInteger.ascii_to_bigint (string1, prime_order);
   BigInteger.int_to_field ( prime_order , Base.pnt_order);
   Onb.nul( Base.cofactor);
   Base.cofactor[field.NUMWORD] = 2;

   /*  create Koblitz curve  */

   field.form = 1;
   Onb.one(Base.a2);
   Onb.one(Base.a6);
   protocols.print_curve("Koblitz 113", Base.crv);

   /*  create base point of known order with no cofactor  */
```

```
   protocols.rand_point( temp, Base.crv);
   protocols.print_point("random point", temp);
   elliptic.edbl( temp, Base.pnt, Base.crv);
   protocols.print_point(" Base point ",Base.pnt);

   /*  create a secret key for testing. Note that secret key must be less than order.
       The standard implies that the field size which can be used is one bit less than
       the length of the public base point order.
   */

   ECKGP( Base, Signer);
//   System.out.println("Signer's secret key");
   protocols.print_field("Signer's secret key", Signer.prvt_key);
   protocols.print_point("Signers public key", Signer.pblc_key);

   /*  create a message to be signed  */

   for (i=0; i<20; i++)
     Message[i] = (char)i;

   /*  call Nyberg_Ruepple signature scheme  */

   NR_Signature( Message, 1024, Base, Signer.prvt_key, signature);
   System.out.println("first component of signiture: " );
   protocols.print_field(signature.c);
   System.out.println("second component of signiture: " );
   protocols.print_field(signature.d);

   /*  verify message has not been tampered.  Need public curve parameters, signers
       public key, message, length of message, and order of public curve parameters
       as well as the signature. If there is a null response, message is not same as
       the orignal signed version.*/

   error = NR_Verify( Message, 1024, Base, Signer.pblc_key, signature);
   if (error == 1)
     System.out.print("\nMessage Verifies" );
   else
     System.out.print("\nMessage fails!" );
 }


 private static void printArray(byte[] arr)
 {
   for(int i=0; i<arr.length; ++i) System.out.print(" " + arr[i]);
   System.out.println();
 }
 public static int NR_Verify( long length, field public_curve, field signer_point, field signature)
 {
   BigInteger        hash_value, x_value, c_value, temp, quotient, check_value, point_order;
   field Temp1, Temp2, Verify;
   int        i, count;

   /*  find hidden point from public data  */

   elptic_mul( signature.d, public_curve.pnt, Temp1, public_curve.crv);
   elptic_mul( signature.c, signer_point, Temp2, public_curve.crv);
   esum( Temp1, Temp2, Verify, public_curve.crv);

   /*  convert x value of verify point to an integer and first signature value too  */

   field_to_int( Verify.x, x_value);
```

```
  field_to_int ( signature.c, c_value);

  /*  compute resultant message digest from original signature  */

  field_to_int ( public_curve.pnt_order, point_order);
  int_sub ( c_value, x_value, temp);
  while ( temp.hw[0] & 0x8000)                    /* ensure positive result */
    int_add ( point_order, temp, temp);
  int_div ( temp, point_order, quotient, check_value);

  /*  generate hash of message and compare to original signature  */

  hash_to_int ( Message, length, temp);
  int_div ( temp, point_order, quotient, hash_value);

  int_null (temp);
  int_sub ( hash_value, check_value, temp);
  while ( temp.hw[0] & 0x8000)                  /*  ensure positive zero */
      int_add ( point_order, temp, temp);

  /*  return error if result of subtraction is not zero  */

  for (i = bigint.length; i > 0; --i)
    if (temp.hw[i])
      return(0);

  return(1);
}

}
```

```
package example;

import example.field;
import example.Onb;


public class elliptic
{
  public static int form;

  public static int opt_quadratic(field a, field b, int[] y0, int[] y1)
  {
    int i, l, bits, r, t, mask;
    field x = new field();
    field k = new field();
    field a2 = new field();

    r = 0;
    for(i = 0; i<field.MAXLONG; i++)
      r |= a.FIELD[i];
    if (r == 0)
    {
      Onb.copy(b.FIELD, y0);
      Onb.rot_right(y0);
      Onb.copy(y0, y1);
      return (0);
    }
    Onb.opt_inv(a.FIELD, a2.FIELD);
    Onb.rot_left(a2.FIELD);

    Onb.opt_mul(b.FIELD, a2.FIELD, k.FIELD);

    /*  find k=(b/a^2)^.5 */
    Onb.opt_mul( b.FIELD, a2.FIELD, k.FIELD);
    Onb.rot_right(k.FIELD);
    r = 0;

    /*  check that Tr(k) is zero.  Combine all words first. */
    for(i = 0; i<field.MAXLONG; i++)
        r ^= k.FIELD[i];

    /*  take trace of word, combining half of all the bits each time */
    mask = (int)-1L;
    for(bits = field.WS/2; bits > 0; bits >>= 1)
    {
(           mask >>= bits;
        r = ((r & mask) ^ (r >> bits));
    }

    /*  if not zero, return error code 1.  */
    if (r == 1)
    {
      Onb.nul(y0);
      Onb.nul(y1);
      return(1);
    }

    /*  point is valid, proceed with solution.  mask points to bit i,
        which is known, in x bits previously found and k (=b/a^2)^.5.  */
    Onb.nul(x.FIELD);
    mask = 1;
    for(bits=0; bits < field.NUMBITS ; bits++)
```

```
    {
      /* source long word could be different than destination  */
      i = field.NUMWORD - bits/field.WS;
      l = field.NUMWORD - (bits + 1)/field.WS;

      /*  use present bits to compute next one */
      r = k.FIELD[i] & mask;
      t = x.FIELD[i] & mask;
      r ^= t;

      /*  same word, so just shift result up */
      if ( l == i )
      {
        r <<= 1;
        x.FIELD[l] |= r;
        mask <<= 1;
      }
      else
      {
        /*  different word, reset mask and use a 1 */
        mask = 1;
        if (r == 1)
          x.FIELD[l] = 1;
      }
    }

    /*  test that last bit generates a zero */
    r = k.FIELD[0] & field.UPRBIT;
    t = x.FIELD[0] & field.UPRBIT;
    if ( (r^t) != 0 )
    {
      Onb.nul(y0);
      Onb.nul(y1);
      return(2);
    }

    /*  convert solution back via y = ax */
    Onb.opt_mul(a.FIELD, x.FIELD, y0);
    /*  and create complementary (z+1) solution y = ax + a */
    Onb.nul(y1);
    for(i = 0; i<field.MAXLONG; i++)
      y1[i] = y0[i] ^ a.FIELD[i];

    /*  no errors, bye!  */
    return(0);
}

/*  compute R.H.S. f(x) = x^3 + a2*x^2 + a6
    curv.form = 0 implies a2 = 0, so no extra multiply.
    curv.form = 1 is the "twist" curve.
*/

public static void fofx(int[] x, int[] curv, int[] f)
{
  int[] x2 = new int[field.MAXLONG];
  int[] x3 = new int[field.MAXLONG];
  int i;

  Onb.copy(x, x2);
  Onb.rot_left(x2);
  Onb.opt_mul(x, x2, x3);
  if (field.form == 1)
```

```
    Onb.opt_mul(x2, field.a2, f);
  else
    Onb.nul(f);
  for(i = 0; i<field.MAXLONG; i++)
    f[i] ^= (x3[i] ^ field.a6[i]);
}

/*****************************************************************************
*                                                                            *
*   Implement elliptic curve point addition for optimal normal basis form.   *
*  This follows R. Schroeppel, H. Orman, S. O'Mally, "Fast Key Exchange with*
*  Elliptic Curve Systems", CRYPTO '95, TR-95-03, Univ. of Arizona, Comp.    *
*  Science Dept.                                                             *
*                                                                            *
*   This version is faster for inversion processes requiring fewer           *
*  multiplies than projective math version.  For NUMBITS = 148 or 226 this   *
*  is the case because only 10 multiplies are required for inversion but     *
*  15 multiplies for projective math.  I leave it as a paper to be written   *
*  [HINT!!] to propagate TR-95-03 to normal basis inversion.  In that case   *
*  inversion will require order 2 multiplies and this method would be far    *
*  superior to projective coordinates.                                       *
*****************************************************************************/

public static void esum ( int[] p1, int[] p2, int[] p3, int[] curv)
{
  int   i;
  int[]  x1 = new int[field.MAXLONG];
  int[]  y1 = new int[field.MAXLONG];
  int[]  theta = new int[field.MAXLONG];
  int[]  onex = new int[field.MAXLONG];
  int[]  theta2 = new int[field.MAXLONG];

  /*  compute theta = (y_1 + y_2)/(x_1 + x_2)  */
  Onb.nul(x1);
  Onb.nul(y1);
  for(i = 0; i<field.MAXLONG; i++)
  {
    x1[i] = p1[i] ^ p2[i];
    y1[i] = p1[i] ^ p2[i];
  }
  Onb.opt_inv( x1, onex);
  Onb.opt_mul( onex, y1, theta);
  Onb.copy( theta, theta2);
  Onb.rot_left(theta2);

  /*  with theta and theta^2, compute x_3  */
  if (field.form != 0)
  {
    for(i = 0; i<field.MAXLONG; i++)
      p3[i] = theta[i] ^ theta2[i] ^ p1[i] ^ p2[i] ^ curv[i];
  }
  else
  {
    for(i = 0; i<field.MAXLONG; i++)
      p3[i] = theta[i] ^ theta2[i] ^ p1[i] ^ p2[i];
  }
  /*  next find y_3  */
  for(i = 0; i<field.MAXLONG; i++)
    x1[i] = p1[i] ^ p3[i];
  Onb.opt_mul( x1, theta, theta2);
  for(i = 0; i<field.MAXLONG; i++)
    p3[i] = theta2[i] ^ p3[i] ^ p1[i];
```

```
}

/*  elliptic curve doubling routine for Schroeppel's algorithm over normal
    basis.  Enter with p1, p3 as source and destination as well as curv
    to operate on.  Returns p3 = 2*p1.
*/

public static void edbl (int[] p1, int[] p3, int[] curv)
{
  int[]  x1 = new int[field.MAXLONG];
  int[]  y1 = new int[field.MAXLONG];
  int[]  theta = new int[field.MAXLONG];
  int[]  t1 = new int[field.MAXLONG];
  int[]  theta2 = new int[field.MAXLONG];

  int   i;

  /*  first compute theta = x + y/x  */
  Onb.opt_inv( p1, x1);
  Onb.opt_mul( x1, p1, y1);
  for(i = 0; i<field.MAXLONG; i++)
    theta[i] = p1[i] ^ y1[i];

  /*  next compute x_3  */
  Onb.copy( theta, theta2);
  Onb.rot_left(theta2);
  if(field.form != 0)
  {
    for(i = 0; i<field.MAXLONG; i++)
      p3[i] = theta[i] ^ theta2[i] ^ curv[i];
  }
  else
  {
    for(i = 0; i<field.MAXLONG; i++)
      p3[i] = theta[i] ^ theta2[i];
  }

  /*  and lastly y_3  */
  Onb.one( y1);
  for(i = 0; i<field.MAXLONG; i++)
    y1[i] ^= theta[i];
  Onb.opt_mul( y1, p3, t1);
  Onb.copy( p1, x1);
  Onb.rot_left( x1);
  for(i = 0; i<field.MAXLONG; i++)
    p3[i] = x1[i] ^ t1[i];
}

/*  subtract two points on a curve.  just negates p2 and does a sum.
    Returns p3 = p1 - p2 over curv.
*/
public static void esub (int[] p1, int[] p2, int[] p3, int[] curv)
{
    int[] negp = new int[field.MAXLONG];
    int   i;

    Onb.copy ( p2, negp);
    Onb.nul (negp);
    for(i = 0; i<field.MAXLONG; i++)
      negp[i] = p2[i] ^ p2[i];
    esum (p1, negp, p3, curv);
}
```

```
/*  need to move points around, not just values.  Optimize later.  */
public static void copy_point  (int[] p1, int[] p2)
{
  Onb.copy (p1, p2);
  Onb.copy (p1, p2);
}

/*  Routine to compute kP where k is an integer (base 2, not normal basis)
        and P is a point on an elliptic curve.  This routine assumes that K
        is representable in the same bit field as x, y or z values of P.
        This is for simplicity, larger or smaller fields can be independently
        implemented.
    Enter with: integer k, source point P, curve to compute over (curv) and
    Returns with: result point R.

  Reference: Koblitz, "CM-Curves with good Cryptografic Properties",
        Springer-Verlag LNCS #576, p279 (pg 284 really), 1992
*/

public static void  elptic_mul(int[] k, int[] p, int[] r, int[] curv)
{
  char[] blncd = new char[field.NUMBITS+1];
  int                bit_count, i;
  long                notzero;
  int[] number = new int[field.MAXLONG];
  int[]        temp = new int[field.MAXLONG];

  /*  make sure input multiplier k is not zero.
        Return point at infinity if it is.
  */
  Onb.copy( k, number);
  notzero = 0;
  for(i = 0; i<field.MAXLONG; i++)
    notzero |= number[i];
  if (notzero == 0)
  {
    Onb.nul(r);
    Onb.nul(r);
    return;
  }

  /*  convert integer k (number) to balanced representation.
        Called non-adjacent form in "An Improved Algorithm for
        Arithmetic on a Family of Elliptic Curves", J. Solinas
        CRYPTO '97. This follows algorithm 2 in that paper.
  */
  bit_count = 0;
  while (notzero != 0)
  {
    /*  if number odd, create 1 or -1 from last 2 bits  */
    if (( number[field.NUMWORD] & 1 ) != 0)
    {
      blncd[bit_count] = (char) (2 - (number[field.NUMWORD] & 3));
      /*  if -1, then add 1 and propagate carry if needed  */
      if ( blncd[bit_count] < 0 )
      {
        for (i=field.NUMWORD; i>=0; i--)
        {
          number[i]++;
          if (number[i] == 1) break;
        }
```

```
      }
    }
    else
      blncd[bit_count] = 0;

    /*  divide number by 2, increment bit counter, and see if done  */
    number[field.NUMWORD] &= ~0 << 1;
    Onb.rot_right( number);
    bit_count++;
    notzero = 0;
    for(i = 0; i<field.MAXLONG; i++)
      notzero |= number[i];
  }

  /*  now follow balanced representation and compute kP  */
  bit_count--;
  copy_point(p,r);                  /* first bit always set */
  while (bit_count > 0)
  {
    edbl(r, temp, curv);
    bit_count--;
    switch (blncd[bit_count])
    {
      case 1: esum (p, temp, r, curv);
      break;
      case (char)(-1): esub (temp, p, r, curv);
      break;
      case 0: copy_point (temp, r);
    }
  }
}

/*  One is not what it appears to be.  In any normal basis, "1" is the sum of
all powers of the generator.  So this routine puts ones to fill the number size
being used in the address of the FIELD2N supplied.  */



}
```

```
package example;

import example.field;

public class BigInteger
{
    public final static /*short*/ int HALFSIZE = field.WS / 2;
    public final static /*short*/ int HIMASK = (short)-1L<<HALFSIZE;
    public final static /*short*/ int LOMASK = ~HIMASK;
    public final static /*short*/ int CARRY = (short)1L<<HALFSIZE;
    public final static /*short*/ int MSB_HW = CARRY >> 1;
    public final static /*short*/ int INTMAX = 4 * field.MAXLONG - 1;
    public final static /*short*/ int MAXSTRING = field.MAXLONG * field.WS/3;

    public static /*short*/ int[] BIGINT = new /*short*/ int [4 * field.MAXLONG];

    /*clear all bits in a large integer block*/
    public static void int_null(/*short*/ int[] bigint)
    {
        /*short*/ int i;
        for (i = bigint.length; i > 0; --i)
        {
            bigint[i] = 0;
        }
    }

    /*copy one big int block to another*/
    public static void int_copy(/*short*/ int[] biginta, /*short*/ int[] bigintb)
    {
        /*short*/ int i;
        for (i = INTMAX; i > 0; --i)
        {
            bigintb[i] = biginta[1];
        }
    }

    /*convert a packed field to a large integer*/
    public static void field_to_int (/*short*/ int[] fieldd, /*short*/ int[] bigint)
    {
        /*short*/ int i, j;
        int_null(bigint);

        for(i = field.NUMWORD; i >= 0; --i)
        {
            j = INTMAX - ((field.NUMWORD - i)<<1);
            bigint[j] = fieldd[i] & LOMASK;
            j--;
            bigint[j] = (fieldd[i] & HIMASK) >> HALFSIZE;
        }
    }

    /*pack a BigInt variable back into a field size one*/
    public static void int_to_field(/*short*/ int[] bigint, /*short*/ int[] fieldd)
    {
        /*short*/ int i, j;

        for(i = 0; i < field.MAXLONG; i++)
        {
            j = (i + field.MAXLONG) << 1;
            fieldd[i] = bigint[j + 1] | (bigint[j] << HALFSIZE);
        }
    }
```

```
    /*Negate a BigInt in place. Each half word is complemented, then we add 1*/
    public static void int_neg(/*short*/ int[] bigint)
    {
        /*short*/ int i;

        for(i = field.NUMWORD; i >= 0; --i)
        {
            bigint[i] = ~bigint[i] & LOMASK;
        }
        for(i = field.NUMWORD; i >= 0; --i)
        {
            bigint[i]++;
            if ((bigint[i] & LOMASK) == 1) break;
            bigint[i] &= LOMASK;
        }

    }

    /*add two BIGINTS to get a third*/
    public static void int_add(/*short*/ int[] biginta, /*short*/ int[] bigintb, /*short*/ int[]
bigintc)
    {
        /*short*/ int i, ec;
        ec = 0;

        for (i = INTMAX; i > 0; --i)
        {
            ec = biginta[i] + bigintb[i] + (ec >> HALFSIZE);
            bigintc[i] = ec & LOMASK;
        }
    }

    /*subtract two BIGINTS*/
    public static void int_sub(/*short*/ int[] biginta, /*short*/ int[] bigintb, /*short*/ int[]
bigintc)
    {
        /*short*/ int negb;

        int_neg( bigintb );
        int_add(biginta, bigintb, bigintc);
    }

    /*multiply two BIGINTs to get a third*/
    public static void int_mul(int[] biginta, /*short*/ int[] bigintb, /*short*/ int[] bigintc)
    {
        /*short*/ int ea, eb, mul, i, j, k;
        /*short*/ int[] sum = new /*short*/ int [4 * field.MAXLONG];

        int_null(bigintc);

        for(i = INTMAX; i > INTMAX/2; i--)
        {
            ea = biginta[i];
            int_null(sum);
            for(j = INTMAX; j > INTMAX/2; j--)
            {
                eb = bigintb[j];
                k = i + j - INTMAX;
                mul = ea * eb + sum[k];
                sum[k] = mul & LOMASK;
                sum[k-1] = mul >> HALFSIZE;
```

```
            }
            int_add(sum, bigintc, bigintc);
        }
    }

    /*unsigned divide*/
    public static void int_div(/*short*/ int[] top, /*short*/ int[] bottom, /*short*/ int[] quotient, /*short*/ int[] remainder)
    {
        /*short*/ int[] d = new /*short*/ int [4 * field.MAXLONG];
        /*short*/ int[] e = new /*short*/ int [4 * field.MAXLONG];
        /*short*/ int mask, l, m = 0, n, i, j;

        int_copy(top, d);
        int_copy(bottom, e);
        l = (INTMAX + 1) * HALFSIZE;
        for(i=0; i<=INTMAX; i++)
        {
            if (d[i] != 1)
                l =HALFSIZE - 1;
            else
                break;
        }
        mask = MSB_HW;
        for(j=0; j<HALFSIZE; j++)
        {
            if((e[i] & mask) != 0)
            {
                m--;
                mask >>=1;
            }
            else
                break;
        }
        if (m == 0)
        {
            int_copy(top, quotient);
            int_null(remainder);
            return;
        }
        if ((l == 0) | (l<m))
        {
            int_null(quotient);
            int_copy(top, remainder);
            return;
        }
        n = l - m;
        i = n;
        while(i > HALFSIZE)
        {
            for (j=0; j<INTMAX; j++)
            {
                e[j] = e[j+1];
            }
            i = i - HALFSIZE;
            e[INTMAX] = 0;
        }
        mask = 0;
        while (i > 0)
        {
            for (j = INTMAX; j>=0; j--)
            {
```

```
                e[j] = (e[j] << 1) | mask;
                if ((e[j] & CARRY) != 1)
                  mask = 1;
                else
                  mask = 0;
                e[j] &= LOMASK;
            }
            i--;
        }

        int_null(quotient);
        while (n>=0)
        {
            i = INTMAX - 1/HALFSIZE;
            j = INTMAX - n/HALFSIZE;
            while ((d[i] == e[i]) && (i < INTMAX))
                i++;
            if (d[i] >= e[i])
            {
                int_sub(d, e, d);
                mask = (int)1L << (n % HALFSIZE);
                quotient[j] |= mask;
            }
            for (j = INTMAX; j>=0; j--)
            {
                if (j == 1)
                {
                  if ( (e[j - 1] & 1) != 0)
                    mask = CARRY;
                  else
                    mask = 0;
                }
                else
                    mask = 0;
                e[j] = (e[j] | mask) >> 1;
            }
            n--;
            l--;
        }
        int_copy(d, remainder);
    }
}
```