NTNU

Norwegian University of
Science and Technology

RFID implementation and performance analysis of a short MQQ digital signature

Kamran Saleem Soomro

Master of Telematics - Communication Networks and Networked Services (2 year)
Submission date: June 2010
Supervisor: Danilo Gligoroski, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem Description

Recently an algorithm called MQQ-SIG for producing short and very fast digital signatures was proposed. As the RFID based contactless smart cards are becoming more popular and they have limited computational and storage resources, it is important to have good performance of digital signature in that technology too.

Assignment given: 11. January 2010
Supervisor: Danilo Gligoroski, ITEM

Abstract

Contactless smart cards (RFID cards) have been widely used for many applications such as epassport, ebanking, transit fare payment and access control systems. These cards have limited resources for performing arithmetic and logical operations and storing data along with program code. As asymmetric cryptographic algorithms, performs time consuming complex operations and demands more resources therefore these operations are performed in the special co-processors inside smart cards. The implementation of these extra co-processors increase the cost of smart cards.

Recently a new algorithm Multivariate Quadratic Quasigroup (MQQ) has been proposed for asymmetric cryptography and it is claimed that decryption oprations are faster than already existing algorithms (RSA, ECC) [17]. Eventually, a digital signature scheme based on MQQ has been proposed and it is named as MQQ-SIG [28]. In original MQQ public key algorithm the size of private and public key was quite large in (KBytes). The size of private key has been significantly reduced in MQQ-SIG scheme. Due to this improvement in the private key size, it becomes possible to implement signing procedure of MQQ-SIG inside contactless smart card. The fast signing speed and simple operations performed in signing makes MQQ-SIG an appealing choice for smart cards which has constrained resources comparatively to other devices such as mobile cell phones and personal computers (PC).

In this thesis we have implemented the digital signature part of MQQ-SIG algorithm in Java for the 8-bit contactless smart card from the NXP family JCOP 41 V2.2.1. These cards have Java Card Virtual Machine (JCVM) which enables limited features of Java. This is a completely original work and as far as we know there are no other Java implementations of MQQ-SIG digital signature.

Key generation part of MQQ is quite time consuming and therefore can not be implemented inside smart cards. Similarly, verification part of MQQ-SIG utilize public key for verification of signed message. The public key of MQQ scheme is quite large in hundreds of KBytes and therefore can not be stored inside smart cards. These two parts of MQQ-SIG has been implemented on desktop computers and are not part of our Master thesis.

Preface

This thesis has been accomplished and submitted for the final requirement in the Master Degree of Technology program at the Norwegian University of Science and Technology (NTNU). Thesis work has been performed at the Department of Telematics in the Spring semester 2010 under the supervision of Danilo Gligoroski.

The topic of this thesis represents implementation of a new fast digital signature algorithm along with programming skills required for the implementation inside java based contactless smart card.

Acknowledgements

I would like to thanks my supervisor Danilo Gligoroski for providing me all the motivation, technical help and friendly environment throughout the duration of Master thesis. Without his kind help it would have not been possible to achieve the required objectives.

I also would like to thanks my wife and children for making me feel a perfect and happy man of this world. This gives me enormous amount of energy and dedication for the successful completion of Master thesis.

Kamran Saleem Soomro

Trondheim, June 14, 2010

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Digital signature has been utilizing in many applications like e-banking, e-procurement and e-approval for user authentication and message integrity. With the increasing legal validity of digital signature in many countries including USA and Germany, it has been becoming widely used in signing electronic mails and orders. Digital signature offers businesses more secure, paperless and efficient working environment by reducing cost and time. Smart cards provide ideal environment to store user's private key securely and to generate digital signature inside the card without any external interference. As digital signature uses PKI (Public Key Infrastructure) which uses different key for encryption and decryption and is much slower than symmetric cryptography which uses same key for both encryption and decryption. Smart cards have limited resources unlike PC (Personal Computer). Therefore, special co-processor is required in smart cards to perform complex operations of digital signature which increases the cost of smart card. Recently an ultrafast new MQQ (Multivariate Quadratic Quasigroup) algorithm has been developed which has 10,000 times faster signing speed and 17,000 times more verification speed than other existing Asymmetric cryptographic algorithm (RSA and ECC) [1]. MQQ-SIG is digital signature scheme based on MQQ algorithm and we want to observe how MQQ-SIG digital signature part performs on 8-bit contactless smart card such as NXP

JCOP 41 V2.2.

1.2 MQQ

In 2008, a new algorithm MQQ (Multivariate Quadratic Quasigroups) was proposed for asymmetric cryptography by Gligoroski, Markovski and Knapskog which was based on a new trapdoor function of multivariate quadratic polynomials obtained by quasigroups and quasigroup string transformations. The security of MQQ has been analyzed by Perret using Groebner basis approach, and Emam M. who has also analyzed MQQ using MutantXL (an improved variant of XL algorithm). Original MQQ is considered broken, but by removing certain percentage of the public key authors claimed that they have a signature scheme MQQ-SIG which is not vulnerable on the previous successful attacks on the full MQQ scheme. MQQ and MQQ-SIG are considered very fast and highly parallelizable public key algorithm compare to RSA and ECC. It is claimed that they are around 10,000 times faster than RSA [16, 17, 28].

1.2.1 MQQ-SIG Digital Signature

MQQ signature variant is called MQQ-SIG [28] and it is constructed by removing one quarter of equations from original MQQ public key algorithm. This signature scheme is believed to be secure against Groenber bases approach which could not solve the remaining known equations. Some characteristics of MQQ-SIG signature scheme is given below:

- It does not provide message expansion therefore it is digital signature with appendix.
- It has length of n bits where $n = 160, 192, 224, 256$.
- Its speculated security is $2^{n/2}$.
- Its verification speed is almost equivalent to other multivariate quadratic public key algorithms.
- In software implementation, signing speed is 500 to 5000 times faster than other popular public key algorithms.

- In hardware implementation, signing speed is around 10,000 times faster than other popular public key algorithms.

Due to all above characteristics, it is very tempting alternative for producing digital signature on smart cards and RFIDs.

1.3 Different Technologies inside Plastic cards

In our daily life, we have been exposed by different plastic cards for accessing academics and its resources, while traveling in bus, tram and metro, withdrawal of money from bank ATM or doing shopping using debit or credit cards. All these applications uses different technologies according to their required parameters of processing power, storage capacity, level of security, types of card accessing interface, and amount of cost. Here these technologies have been discussed.

1.3.1 Smart Cards

These cards are most advanced cards providing many features to high demanding applications. These cards include processor and different types of memories: RAM, ROM and EEPROM and optionally co-processor for performing complex operations of cryptography. ROM contains the operating system for the smart card, RAM stores volatile data required for processing inside processor and EEPROM is non-volatile and contains program specific to application requirement. Many applications like e-passport, bank credit and debit cards and access control system for government and corporate workers demands smart cards. These cards also provide resistance against known attacks and are very difficult to forge or copy and also provide standard encryption protocols (RSA, ECC, SHA-1, 3DES and AES) for data security and privacy.

1.3.2 Contactless Smart Cards

These cards are used in applications demanding more security and privacy just like human identification and electronic payment. These RF-enabled

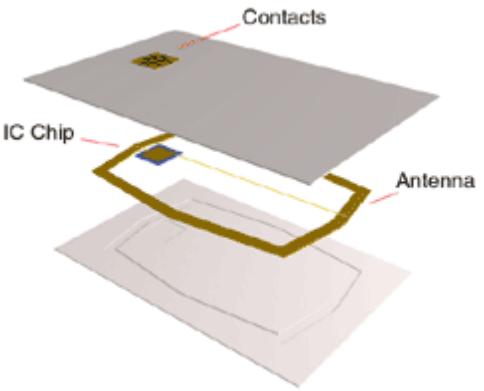
smart cards are almost equivalent to contact smart cards that we use in our daily life in order to withdraw money from bank ATM by physically inserting smart card into reader. Only difference is the mechanism of accessibility which takes advantage of RF interface in case of contactless smart card and no physical connection is required between card and reader. Contactless smart cards provide more advanced security features along with short range (under 4 inches) of accessibility from RFID reader. Nowadays, Many emerging applications are taking benefit of this technology including Transit fare payment systems, electronic payment cards, electronic passport and visas, government personal identity verification systems, and confidential building access systems. Contactless smart card contains microcontroller, read/write memory and RF interface for contactless communication with RFID reader and performs complex operations (encryption and authentication). Like contact smart card it does not include power supply rather it takes power from RFID reader. As with the increasing use of this technology, international standards for contactless smart cards (ISO/IEC 14443) has been set up and followed by all contactless smart card applications.

1.3.3 Dual Interface or Combi Smart Cards

Dual Interface cards, also called Combi Cards (Combination cards), provides dual interfaces for contact and contactless access to reader. These both interfaces use same Integrated Circuit and hence provide more flexibility for applications. Figure 1.1(a) describe Combi cards combines two interfaces: contact and contactless in one card. Contacts are provided for contact access and Antenna is provided for contactless access and both interfaces are using same IC Chip. Combi cards adhere to standards ISO/IEC 14443 for contactless interface and ISO/IEC 7816 for contact interface. In our implementation dual interface java card has been used as shown in figure 1.1(b).

1.4 Security features of contactless smart cards

As we have mentioned above contactless smart card exploit RF technology for wireless communication with the reader and operate at a lower distance



(a) Dual Interface Card Components

(b) Dual Interface Java Card

Figure 1.1: Dual Interface or Combi Cards

around 4 inches. Contactless smart cards and reader conform to international standards ISO/IEC 14443 and can include many standard cryptographic protocols such as RSA, ECC, 3DES and AES. Many applications can use these security protocols in order to maintain mutual authentication, confidentiality, integrity and privacy of data either stored inside card or data transmitted between card and reader. These security features are further explained below:

1.4.1 Mutual Authentication

This feauture provides very important security functionality for both card and reader before taking place any real transaction. It is important to know

for both parties involved in a transaction about each other. This feature avoids any duplicate card or reader to take part in transaction. Both card and reader authenticate each other by providing some secret that is only known by them individually.

1.4.2 Data Security

This feature provides protection of communication between contactless smart card and reader against eavesdropping by encrypting the data. This feature provides complete protection of data stored in card or communicated through air. Digital signature can be used to authenticate the card and information provided by card along with integrity of information and cryptographically random number generators can be used to produce keys inside card in order to prevent replay attack.

1.4.3 Contactless smart card physical security

This feature in contactless smart card is equivalent to contact smart card and it is quite difficult to make a duplicate card. The chip inside card has intelligent logic to prevent all tampering attempts and it has sensors to detect thermal and UV light attacks and special circuitry for preventing differential power analysis attack.

1.5 Benefits of Contactless Technology

Contactless technology is expanding with the passage of time in many applications like access control, e-passport and e-payment due to its following benefits:

1.5.1 Saving of time and Easy to Use

Some applications require fast identification of users especially in transit payment systems and e-passport where many people stay in a queue at a

time. Contactless technology offers fast access without swapping the card inside the reader and it is also possible to bring the card within close proximity of reader without taking the card outside of purse due to RF technology used for contactless access.

1.5.2 Dependability

Some applications as mentioned earlier have heavy usage all day long and in that case both cards and readers are utilized more frequently causing damage to reader and card. Contactless cards and readers are properly covered and that's why provide more durability are reliability against heavy usage along with humidity, dirt, cold and other harsh weather elements.

1.6 Digital signature for contactless smart card

Signing a document is not a new concept and has been used since many decades in order to authenticate the identity of the signer. Nowadays, many applications have adopted new technology to secure and speed up the process. One of them is digital signature, it is based on same concept of signing a document and provide authentication of signer but provide more security because it is hard to forge digital signature than hand-made signature. Digital signature is used to provide authentication of signer and integrity of message. It offers benefit of authentication along with non-repudiation so that once the document is signed then any effort of editing document will lead to failure in verification process of digital signature. Digital signature can be generated inside a contactless smart card. This will bind the specific user to a particular contactless smart card. All the transactions made through the contactless smart card then can be authenticated and their integrity can also be verified with the help of digital signature.

1.7 Applications of contactless smart card

The most popular contactless smart card applications are Access Control, National Identity cards, E-passports and E-payments. The overview of each application is described below:

1.7.1 Access Control Systems

These applications are getting popularity both in government and in private sector. Contactless smart card stores enough information in order to show legitimate identity of user to access control system so that system can verify the real user. Sometimes, contactless smart cards also contain biometric data of user to provide more security.

USA government has taken steps to standardize the personal identity verification for their federal employees and contractors which provide security related specifications required for smart card in order to implement more secure access control system [3]. In this regard, FIPS 201 (Federal Information Processing Standards 201) is published by NIST (National Institute for Standards and Technology) in order to satisfy the requirements of HSPD-12 (Homeland Security Presidential Directive 12) regarding PIV (Personal Identity Verification). It is approved by secretary of commerce and issued on 25th February 2005.

Other identity verification smart cards in US are: CAC-C (Common Access Card with Contactless) which is being issued by DOD (Department of Defense), USA, for the identity verification of on-duty military personnel [4] and TWIC (Transportation Worker Identification Credential) issued by the Transportation Security Administration and FRAC (First Responder Authentication Card) issued by Department of Homeland Security . FRAC, an identity management system, provides first responders easy access to government buildings and emergency areas in case of disaster.

1.7.2 National Identity Cards

These are also being implemented in contactless smart cards. Several countries are planning to adopt this technology including Germany. Malaysia is the first country in the world that deployed National ID smart card (MyKad) for their citizens in 2001. Now china is becoming a largest implementer of contactless National ID card [5].

1.7.3 E-Passport

This is one of the most crucial and growing applications of contactless smart cards in identity verification. Many countries have already take benefit of this technology for passport. E-passport complies with technical standards published by the ICAO (International Civil Aviation Organization) [6]. In addition to guidelines provided for MRTD (Machine Readable Travel Documents) that are mentioned in ISO 7501. ICAO is planning that all countries will convert their existing non-digital passport into digital version (e-passport) that stores encrypted biometric data on a contactless smart card. Earlier e-passports were implementing BAC (Basic Access control) scheme that protects owner's data including facial image stored inside e-passport contactless smart card being accessed without holder's consent. A shared key is derived from the data: passport number, expiry date and holder's birthday obtained by OCR (Optical Character Recognition) reader and then the generated key is used to access the data stored inside contactless smart card. In order to provide data integrity data inside contactless card is digitally signed by home country. BAC also provides protection against eavesdropping and skimming attacks. EU (European Union) countries have implemented EAC (Extended Access Control) [7] scheme that includes additional biometric data (fingerprints) in the e-passport. EAC uses PKI (Public Key Infrastructure) for participating parties and provide mutual authentication.

1.7.4 E-payment

This is another growing application of contactless smart card which provides time saving and easy to use. Major financial entities have already

adopted contactless payment solution including American Express (ExpressPay), MasterCard (PayPass) and Visa (payWave) in their credit and debit cards. In Hong Kong, Octopus cards, implemented with Sony Felica technology which is similar but not exactly comply with ISO 14443. Octopus cards, rechargeable payment cards, are issued in 1997 for travel payments and hence are considered as most mature implementation of contactless smart card. Nowadays, these cards are also used in stores, supermarkets, restaurants, parking garages and other point of sale applications. The popularity of Octopus card in can be examined by following facts: utilized by 95% population of Hong Kong, 9 million Octopus cards along with 150,000 smart watches have been issued; 7 million daily transactions are made worth 6.5 million US dollar [8].

In the UK (United Kingdom), the Barclays' OnePulse card , developed with Visa, provides multi-application facilities including transit and payment. OnePulse card is three in one and can be used as London travel card (Oyster card), contactless payment card with limited purchasing option within 10 British pounds using Visa payWave and chip-and-pin contact credit and debit card similar like Barclay bank card. This card complies to ISO 14443 standards for contactless communication which is specified in EMV (Europay, MasterCard and Visa) specification.

1.8 Thesis Structure

Brief Description of the rest of thesis:

Chapter 2: (Backgroud) This chapter provides background information for used elements in our thesis. These elements are MQQ-SIG algorithm used for signing of message, RFID tags used for contactless access of smart cards, other used cryptographic algorithms (RSA, ECDSA) and hash function (SHA1) and some known attacks on smart cards are also discussed in the end.

Chapter 3: (Java Cards Technology) In order to implement a software over a system, it is necessary to have knowledge about the system architecture and its limitations. This chapter provides some basic and necessary information required to write a software over used contactless smart card. Here the

protocol (APDU) used for cummunication between contactless smart card and reader is also discussed.

Chapter 4: (Implementation) This chapter provides information about the specifications of used contactless smart card and reader. It also discuss about the tool used for programming the signing algorithm for contactless smart card. Here we have discussed the implementation details of MQQ-SIG signing algorithm along with left and right parastrophes of quasigroup.

Chapter 5: (Evaluation and Discussion of Results) This chapter provides obtained results of our implementation

Chapter 6: (Conclusion and Future Work) This chapter provides conclusion of our Master thesis and Future Work

Appendix A: (Program Code for MQQ-SIG Digital Signature) Appendix A includes the programmig code of MQQ-SIG signing algorithm which is developed for contactless smart card.

Appendix B: (Program Code for RSA Digital Signature) Appendix B includes the programming code for 1024 bits RSA-CRT digital signature. RSA-CRT algorithm is already implemented inside the co-processor of used contactless smart card. The only version of Secure Hash Function given inside used contactless smart card for RSA-CRT is SHA1.

Appendix C: (Program Code for ECDSA Digital Signature) Appendix C includes the programming code for 163 bits ECDSA digital signature. RSA-CRT algorithm is already implemented inside the co-processor of used contactless smart card. The only version of Secure Hash Function given inside used contactless smart card for ECDSA is SHA1.

Chapter 2

Background

This chapter provides background information about used cryptographic algorithms in our implementation and RFID tags used for contactless smart card and digital signature.

2.1 Quasigroups and MQQ

Here we give a brief overview of quasigroups and quasigroup string transformations. A more detailed explanation is found in [27, 26].

Definition 1. A quasigroup $(Q, *)$ is a groupoid satisfying the law

$$(\forall u, v \in Q)(\exists! x, y \in Q) \quad u * x = v \ \& \ y * u = v. \tag{2.1}$$

This implies the cancelation laws $x*y = x*z \Longrightarrow y = z,\ y*x = z*x \Longrightarrow y = z$ and the equations $a * x = b,\ y * a = b$ have unique solutions $x,\ y$ for each $a, b \in Q$.

Given a quasigroup $(Q, *)$ five so called parastrophes (or conjugate operations) can be adjoint to $*$, and here we will use only two of them, denoted by $\setminus$ and $/$ and defined by

$$x * y = z \iff y = x \setminus z \iff x = z/y \tag{2.2}$$

Then $(Q, \backslash)$ and $(Q, /)$ are quasigroups too and the algebra $(Q, *, \backslash, /)$ satisfies the identities

$$x \backslash (x * y) = y, \quad (x * y)/y = x, \quad x * (x \backslash y) = y, \quad (x/y) * y = x \tag{2.3}$$

Conversely, if an algebra $(Q, *, \backslash, /)$ with three binary operations satisfies the identities (2.3), then $(Q, *)$, $(Q, \backslash)$, $(Q, /)$ are quasigroups and (2.2) holds.

To define a multivariate quadratic PKC for our purpose, we will use the following Lemma:

Lemma 1. For every quasigroup $(Q, *)$ of order 2^d and for each bijection $Q \to \{0, 1 \ldots, 2^d - 1\}$ there are a uniquely determined vector valued Boolean functions $*_{vv}$ and d uniquely determined $2d$-ary Boolean functions $f_1, f_2, \ldots, f_d$ such that for each $a, b, c \in Q$

$$\begin{gathered} a * b = c \iff \\ *_{vv}(x_1, \ldots, x_d, y_1, \ldots, y_d) = (f_1(x_1, \ldots, x_d, y_1, \ldots, y_d), ..., \\ f_d(x_1, \ldots, x_d, y_1, \ldots, y_d)). \end{gathered} \tag{2.4}$$

Recall that each k-ary Boolean function $f(x_1, \ldots, x_k)$ can be represented in a unique way by its algebraic normal form (ANF), i.e., as a sum of products

$$ANF(f) = \alpha_0 + \sum_{i=1}^{k} \alpha_i x_i + \sum_{1 \le i < j \le k} \alpha_{i,j} x_i x_j + \sum_{1 \le i < j < s \le k} \alpha_{i,j,s} x_i x_j x_s + \ldots, \tag{2.5}$$

where the coefficients $\alpha_0, \alpha_i, \alpha_{i,j}, \ldots$ are in the set $\{0, 1\}$ and the addition and multiplication are in the field $GF(2)$. The ANFs of the functions f_i give us information about the complexity of the quasigroup $(Q, *)$ via the degrees of the Boolean functions f_i. It can be observed that the degrees of the polynomials $ANF(f_i)$ rise with the order of the quasigroup. In general, for a randomly generated quasigroup of order 2^d, $d \ge 4$, the degrees are higher than 2. Such quasigroups are not suitable for our construction of multivariate quadratic PKC.

Definition 2. A quasigroup $(Q, *)$ of order 2^d is called Multivariate Quadratic Quasigroup (MQQ) of type $Quad_{d-k}Lin_k$ if exactly $d - k$ of the polynomials f_i are of degree 2 (i.e., are quadratic) and k of them are of degree 1 (i.e., are linear), where $0 \le k < d$.

In [17] first sufficient conditions were given one quasigroup to be MQQ and an algorithm for finding MQQs up to the order of 2^5 was given there. That work was later extended in [29] for constructing MQQs of order 2^d for any d. The common characteristic of MQQs produced by those two methods is that the quasigroups are bilinear. Namely, for a given multivariate quadratic quasigroup $(Q, *)$ given by the equations:

$$*_{vv}(x_1, \ldots, x_d, y_1, \ldots, y_d) = (f_1(x_1, \ldots, x_d, y_1, \ldots, y_d), ..., f_d(x_1, \ldots, x_d, y_1, \ldots, y_d))$$

we can rewrite those equations in the following form:

$$\mathbf{A_1} \cdot (y_1, \ldots, y_d)^T + \mathbf{b_1} \equiv \mathbf{A_2} \cdot (x_1, \ldots, x_d)^T + \mathbf{b_2} \tag{2.6}$$

where $\mathbf{A_1} = [f_{ij}]_{d \times d}$ is a $d \times d$ matrix and $\mathbf{b_1} = [u_i]_{d \times 1}$ is a $d \times 1$ vector of linear Boolean expressions of the variables $x_1, \ldots, x_d$, while $\mathbf{A_2} = [g_{ij}]_{d \times d}$ is a $d \times d$ matrix and $\mathbf{b_2} = [v_i]_{d \times 1}$ is a $d \times 1$ vector of linear Boolean expressions of the variables $y_1, \ldots, y_d$.

The description of the algorithm for producing bilinear MQQs of order 2^d given in [29] can be described shortly by the following expression:

$$\mathbf{x} * \mathbf{y} \equiv \mathbf{B} \cdot \mathbf{U}(\mathbf{x}) \cdot \mathbf{A_2} \cdot \mathbf{y} + \mathbf{B} \cdot \mathbf{A_1} \cdot \mathbf{x} + \mathbf{c} \tag{2.7}$$

where $\mathbf{x} = (x_1, \ldots, x_d)$, $\mathbf{y} = (y_1, \ldots, y_d)$, the matrices $\mathbf{A_1}$, $\mathbf{A_2}$ and $\mathbf{B}$ are nonsingular in $GF(2)$, of size $d \times d$, the vector $\mathbf{c}$ is a random d-dimensional vector with elements in $GF(2)$ and all of them are generated by a uniformly random process. The matrix $\mathbf{U}(\mathbf{x})$ is an upper triangular matrix with all diagonal elements equal to 1, and the elements above the main diagonal are linear expressions of the variables of $\mathbf{x} = (x_1, \ldots, x_d)$. It is computed by the following expression:

$$\mathbf{U}(\mathbf{x}) = I + \sum_{i=1}^{d-1} \mathbf{U}_i \cdot \mathbf{A_1} \cdot \mathbf{x}, \tag{2.8}$$

where the matrices $\mathbf{U}_i$ have all elements 0 except the elements in the rows from $\{1, \ldots, i\}$ that are strictly above the main diagonal. Those elements can be either 0 or 1.

Once we have a multivariate quadratic quasigroup

$$*_{vv}(x_1, \ldots, x_d, y_1, \ldots, y_d) = (f_1(x_1, \ldots, x_d, y_1, \ldots, y_d), ..., f_d(x_1, \ldots, x_d, y_1, \ldots, y_d))$$

we will be interested in those quasigroups that will satisfy the following condition:

$$Rank(\mathbf{B}_{f_i}) \geq 2d - 4, \tag{2.9}$$

where matrices $\mathbf{B}_{f_i}$ are $2d \times 2d$ Boolean matrices defined from the expressions f_i as

$$\mathbf{B}_{f_i} = [b_{j,k}],\ b_{j,8+k} = b_{8+k,j} = 1,\ \text{iff } x_j y_k \text{ is a term in } f_i. \tag{2.10}$$

Proposition 1. A multivariate quadratic quasigroup that satisfies the conditions (2.7), ..., (2.10) can be encoded in a unique way with 81 bytes.

2.2 Description of the MQQ-SIG algorithm

A generic description for our scheme can be expressed as a typical multivariate quadratic system: $\mathbf{S} \circ P' \circ \mathbf{S}' : \{0,1\}^n \to \{0,1\}^n$ where $\mathbf{S}' = \mathbf{S} \cdot \mathbf{x} + \mathbf{v}$ (i.e. $\mathbf{S}'$ is a bijective affine transformation), $\mathbf{S}$ is a nonsingular linear transformation, and P' is a bijective multivariate quadratic mapping on $\{0,1\}^n$.

First we will describe how the mapping $P' : \{0,1\}^n \to \{0,1\}^n$ is defined by the algorithm described in Table 2.1.

$P'(\mathbf{x})$
Input: A vector $\mathbf{x} = (f_1, \ldots, f_n)$ of n linear Boolean functions of n variables. We implicitly suppose that a multivariate quadratic quasigroup $*$ is previously defined, and that $n = 32k$, $k = 5, 6, 7, 8$ is also previously determined.
Output: 8 linear expressions $P'_i(x_1, \ldots, x_n), i = 1, \ldots, 8$ and $n - 8$ multivariate quadratic polynomials $P'_i(x_1, \ldots, x_n), i = 9, \ldots, n$
1. Represent a vector $\mathbf{x} = (f_1, \ldots, f_n)$ of n linear Boolean functions of n variables $x_1, \ldots, x_n$, as a string $\mathbf{x} = X_1 \ldots X_{\frac{n}{8}}$ where X_i are vectors of dimension 8; 2. Compute $\mathbf{y} = Y_1 \ldots Y_{\frac{n}{8}}$ where: $Y_1 = X_1$, $Y_{j+1} = X_j * X_{j+1}$, for even $j = 2, 4, \ldots$, and $Y_{j+1} = X_{j+1} * X_j$, for odd $j = 3, 5, \ldots$ 3. Output: $\mathbf{y}$.

Table 2.1: Definition of the nonlinear mapping $P' : \{0,1\}^n \to \{0,1\}^n$ [28]

The algorithm for generating the public and private key is defined in the Table 2.2.

Algorithm for generating Public and Private key for the MQQ-SIG scheme
Input: Integer n, where $n = 32 \times k$ and $k = \{5, 6, 7, 8\}$.
Output: Public key $\mathbf{P}$: $n - \lfloor \frac{n}{4} \rfloor$ multivariate quadratic polynomials $P_i(x_1, \ldots, x_n)$, $i = 1 + \lfloor \frac{n}{4} \rfloor, \ldots, n$, Private key: Two permutations of the numbers $\{1, \ldots, n\}$, and 81 bytes for encoding a quasigroups $*$
1. Generate an MQQ $*$ according to equations (2.7) ... (2.10). 2. Generate a nonsingular $n \times n$ Boolean matrix $\mathbf{S}$ and affine transformation $\mathbf{S}'$ according to equations (2.11), ..., (2.14). 3. Compute $\mathbf{y} = \mathbf{S}(P'(\mathbf{S}'(\mathbf{x})))$, where $\mathbf{x} = (x_1, \ldots, x_n)$. 4. Output: The public key is $\mathbf{y}$ as $n - \lfloor \frac{n}{4} \rfloor$ multivariate quadratic polynomials $P_i(x_1, \ldots, x_n)$ $i = 1 + \lfloor \frac{n}{4} \rfloor, \ldots, n$, and the private key is the tuple $(\sigma_1, \sigma_k, *)$.

Table 2.2: Algorithm for generating the public and private key [28]

Algorithm for digital signature with the private key $(\sigma_1, \sigma_k, *)$
Input: A document M to be signed.
Output: A signature $\mathbf{sig} = (x_1, \ldots, x_n)$.
1. Compute $\mathbf{y} = (y_1, \ldots, y_n) = H(M)\|_n$, where M is the message to be signed, $H()$ is a standardized cryptographic hash function such as SHA-1, or SHA-2, with a hash output of not less than n bits. The notation $H(M)\|_n$ denotes the least significant n bits from the hash output $H(M)$. 2. Set $\mathbf{y}' = \mathbf{S}^{-1}(\mathbf{y})$. 3. Represent $\mathbf{y}'$ as $\mathbf{y}' = Y_1 \ldots Y_{\frac{n}{8}}$ where Y_i are Boolean vectors of dimension 8. 4. By using the left and right parastrophes $\setminus$ and $/$ of the quasigroup $*$ compute $\mathbf{x}' = X_1 \ldots X_{\frac{n}{8}}$, such that: $X_1 = Y_1$, $X_j = X_{j-1} \setminus Y_j$, for even $j = 2, 4, \ldots$, and $X_j = Y_j / X_{j-1}$, for odd $j = 3, 5, \ldots$. 5. Compute $\mathbf{x} = \mathbf{S}^{-1}(\mathbf{x}') + \mathbf{v} = (x_1, \ldots, x_n)$. 6. A digital signature of the document M is the vector $\mathbf{sig} = (x_1, \ldots, x_n)$.

Table 2.3: Algorithm for signing [28]

The algorithm for signing by the private key $(\sigma_1, \sigma_k, *)$ is defined in Table 2.3.

The algorithm for signature verification with the public key $\mathbf{P} = \{P_i(x_1, \ldots, x_n) \mid i = 1 + \lfloor \frac{n}{4} \rfloor, \ldots, n\}$ is given in Table 2.4.

2.2.1 Nonsingular Boolean matrices in MQQ-SIG

In order to sign a message in MQQ-SIG scheme non-singular matrices are required. The procedure is mentioned in [28].

Algorithm for signature verification with a public key $\mathbf{P} = \{P_i(x_1, \ldots, x_n) \mid i = 1 + \lfloor \frac{n}{4} \rfloor, \ldots, n\}$
Input: A document M and its signature $\mathbf{sig} = (x_1, \ldots, x_n)$.
Output: TRUE or FALSE.
1. Compute $\mathbf{y} = (y_{1+\lfloor \frac{n}{4} \rfloor}, \ldots, y_n) = H(M)\vert_{n-\lfloor \frac{n}{4} \rfloor}$, where M is the signed message, $H()$ is a standardized cryptographic hash function such as SHA-1, or SHA-2, with a hash output of not less than n bits, and the notation $H(M)\vert_{n-\lfloor \frac{n}{4} \rfloor}$ denotes the least significant $n - \lfloor \frac{n}{4} \rfloor$ bits from the hash output $H(M)$. 2. Compute $\mathbf{z} = (z_{1+\lfloor \frac{n}{4} \rfloor}, \ldots, z_n) = \mathbf{P}(\mathbf{sig})$. 3. If $\mathbf{z} = \mathbf{y}$ then return TRUE, else return FALSE.

Table 2.4: Algorithm for verification [28]

In general, if we just simply generate a nonsingular Boolean matrix $\mathbf{S}$ of size $n \times n$ to store such a matrix or its inverse $\mathbf{S}^{-1}$ (that we need for the process of signing), we would need n^2 bits. Having in mind that our target sizes for the size of the digital signatures are $n = 160, 192, 224, 256$ bits (i.e. $n = 32 \times k$ and $k = \{5, 6, 7, 8\}$), for storing $\mathbf{S}^{-1}$ we would need 3.125 Kbytes, 4.5 Kbytes, 6.125 Kbytes or 8.0 Kbytes.

In order to reduce the private information for the linear and affine transformations that is perform in MQQ-SIG scheme, nonsingular matrices $\mathbf{S}$ are defined by the following expression [28]:

$$\mathbf{S}^{-1} = \sum_{i=1}^{k} I_{\sigma_i}, \tag{2.11}$$

where $I_{\sigma_i},\ i = \{1, \ldots, 5, 6, 7, 8\}$ are permutation matrices of size $n = 32 \times k$ and where permutations σ_i are permutations on n elements. They are defined by the following expressions:

$$\begin{cases} \sigma_1 & - & \text{random permutation on } \{1, 2, \ldots n\} \text{ satisfying the condition (2.13)}, \\ \sigma_2 & = & RotateLeft(\sigma_1, 32) \text{ satisfying the condition (2.13)}, \\ \sigma_3 & = & RotateLeft(\sigma_2, 64) \text{ satisfying the condition (2.13)}, \\ \sigma_j & = & RotateLeft(\sigma_{j-1}, 32), \text{for} j = 4, \ldots, k-1, \text{ satisfying the condition (2.13)}, \\ \sigma_k & - & \text{random permutation on } \{1, 2, \ldots n\} \text{ satisfying the condition (2.13)} \end{cases} \tag{2.12}$$

$$\sigma_\nu = \begin{pmatrix} 1 & 2 & \ldots & 8 & 9 & n-1 & n \\ s_1^{(\nu)} & s_2^{(\nu)} & \ldots & s_8^{(\nu)} & s_9^{(\nu)} & s_{n-1}^{(\nu)} & s_n^{(\nu)} \end{pmatrix}, \ \{s_1^{(\nu)}, s_2^{(\nu)}, \ldots, s_8^{(\nu)}\} \bigcap \{1, 2, \ldots, 8\} = \emptyset \tag{2.13}$$

There is required an additional condition to be fulfilled by the permutations $\sigma_1, \ldots, \sigma_k$:

$$L = \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \vdots \sigma_{k-1} \\ \sigma_k \end{bmatrix}, \text{ is a Latin Rectangle.} \tag{2.14}$$

Once a nonsingular matrix $\mathbf{S}^{-1}$ is obtained then its inverse is obtained by computing

$$\mathbf{S} = (\mathbf{S}^{-1})^{-1}$$

and from there the affine transformation is obtained

$$\mathbf{S}'(\mathbf{x}) = \mathbf{S} \cdot \mathbf{x} + \mathbf{v}, \tag{2.15}$$

where the vector $\mathbf{v}$ is n–dimensional Boolean vector defined from the values of the permutation σ_k by the following expression:

$$\mathbf{v} = (v_1, v_2, \ldots, v_n), \text{ where } v_i = \left(\frac{s^{(k)}_{\lceil \frac{i}{4} \rceil}}{2^{i \bmod 4}} \right) \bmod 2. \tag{2.16}$$

In words: the bits of the vector $\mathbf{v}$ are constructed by taking the four least significant bits of the first $\frac{n}{4}$ values in the permutation σ_k.

Proposition 2. The linear transformations $\mathbf{S}^{-1}$ can be encoded in a unique way with $2n$ bytes.

Simple Power Analysis (SPA), Differential Power Analysis (DPA) and High Order DFA

2.3 RFID from Beginning till Today

RFID (Radio Frequency Identification) is a technology that is used to track or identify different kind of objects like animals, goods in supply chain, and weapons in military. The simplest RFID system can consist of transponder (electronic tag) for storing information required for tracking objects and transceiver (reader) for receiving response sent by transponder. It is said

that necessity is the mother of invention. During World War II there was a need to identify the coming airplanes status as friend or foe and this necessity invent the first RFID system called IFF (Identification of Friend or Foe) under the heading of Sir Robert Alexander Watsen-Watt, a Scottish physicist, who also discovered radar in 1935. A California entrepreneur Charles Walton obtained copyrights for passive transponder used for accessing the building without key in 1973. Automated toll payment system was commercialized in mid 1980 with the contribution of scientists of Los Alamos National Laboratory, New Mexico, who have also developed passive transponder based systems that used UHF (Ultra-High Frequency) in order to track cows for vaccination purpose. Eventually, the cow tracking system is replaced by a LF (Low Frequency) (125 KHz) small transponder. These transponders are being used today into access cards of building and also injected in cows after encapsulating in glass. Now days, companies are shifted towards HF (High Frequency) (13.56 MHz) because this frequency provides more data rate than previously used 125 KHz and is unregulated. These systems are being used today in many applications like payment systems (Mobile Speedpass), access control and contactless smart cards, which is the one we are going to use in our thesis [9].

2.4 RF-Enabled Tecnology

RF-enabled technology offers different services to fulfill the specific requirements of applications like storage capacity, operational frequency, computational power, access range, security and privacy of data. Some applications such as animal identification and inventory tracking requires wide range of communication between reader and transponder but less necessity of security and privacy while other applications like human identity verification and payment systems demands more security and privacy and less communication distance. Following are examples of RF-enabled technologies:

2.5 RFID tag

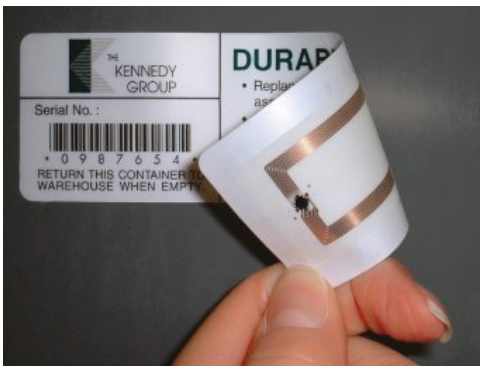
It contains a chip along with an antenna which receives signal from RFID reader and returns back signal with data stored (tag Id number) inside the tag. It is inserted in an object which is supposed to be track or identify and it has minimal support for data security and privacy. RFID Tags are divided into following two types:

2.5.1 Passive RFID tag

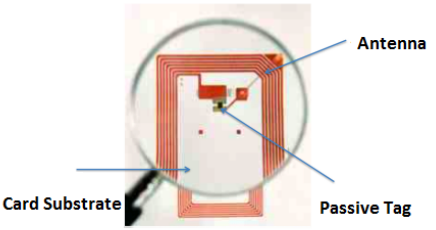
It does not contain a power supply. As an alternative, It generates power from the electromagnetic waves of RFID reader when tag is exposed in it and then tag circuit is activated and transmits back the stored information in memory. The size of passive RFID tag is very small as shown in Figure 2.1(b), sometime equal to a grain of rice or thinner like a paper, due to absence of power supply. Figure 2.1(a) shows the RFID attached in a label that in turn can be attached to the tracking item. Figure 2.1(b) shows RFID passive tag and antenna in a substrate. The RFID reader can access passive RFID tags from the length of 10 millimeters to 6 meters. Many applications utilize passive RFID tags because of low cost related to them.

2.5.2 Active RFID tag

This variant of RFID tag, on the other hand includes an internal power supply, which helps to maintain a longer access range from RFID reader along with large memory to hold more data. These tags can also store information transmitted by RFID reader and also are bigger in size, around the size of a coin, because of presence of power supply. The accessibility of these tags is extended many meters and as far as their battery life time is concern, It can be used around 10 years. Active RFID tags provide many benefits such as precision, dependability, and higher performance in adverse environments, such as humid or harsh



(a) RFID Tag in Label



(b) RFID Tag Components

Figure 2.1: RFID Passive Tags

2.6 Digital Signature

In order to use digital signature in broad spectrum, it is required to store private key of signer in safe place to sign a message and legal framework that bind all parties involved in digital signature application. Smart card provide has potential to store private key and with the help of co-processor a digital signature can also be generated inside the card. In 1995, Utah state in USA has developed legal framework for the digital signature and many other countries including Germany signature legislation adopted it as a guideline [10].

2.6.1 Standards for digital signature

Due to open usage of digital signature, it is required to follow standards for all commercial applications. ISO/IEC 7816-4 provides guidelines for general smart card commands, ISO/IEC 7816-8 provide guidelines for commands required to generate digital signature in smart cards. Standards for smart card plastic body, electrical properties and data transmission are also provided in ISO/IEC 7816. Authentication process between smart card and rest of the world is provided in ISO/IEC 9796-2. Basic methods and mechanisms for digital signature are provided in ISO/IEC 14888. The structure and coding mechanism of PKI certificate is provided in X.509.

2.6.2 Signing and Verification of Digital Signature

After signing a message, the digital signature can be used as a security token that provide user authentication and data integrity anywhere in the world to verify the sender of the message and the originality of the message. This becomes possible because of asymmetric cryptography that provides different private and public keys to a user. User can sign a message with own private key and any one can verify the origin of message using user's public. Figure 2.2 shows signing process of typical digital signature. Plain text message is supplied to hash algorithm that return short hash of message typically 20 bytes. The hash of message provides defense mechanism against message modification with fraudulent intention and also provides short data to complex, time consuming cryptographic operations. Sender's private key is used to decrypt the hash of the message which in turn provide authentication of the sender. Finally the decrypted hash (digital signature) plus plain message are sent to receiver.

Figure2.3 shows verification process of typical digital signature. The received message can be verified with the help of digital signature. Hash of the plain message will be calculated and then digital signature will be encrypted using public key of the sender. After encryption the hash of the digital signature will be compared with the hash of plain message. If both hashes matches with each other then we can say that message is arrived from the original sender and no one has modified the original message otherwise if both hashes do not match then message is fake.

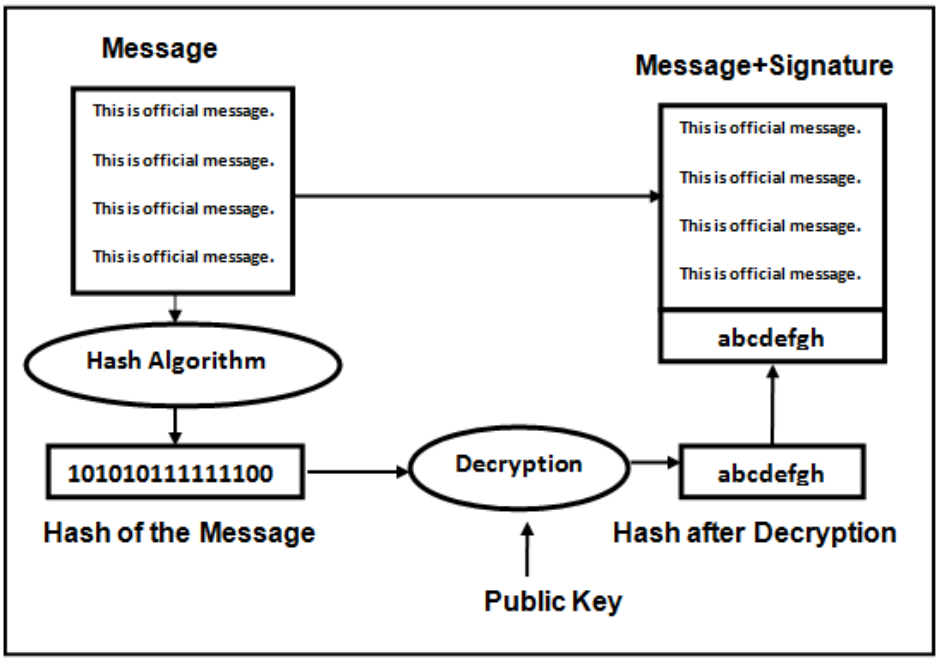


Figure 2.2: Signing Process of Digital Signature

2.6.3 Certificates

It can be observed from figure 2.3 that public key of sender is used to verify that origin of message. It is therefore, necessary to make sure about the authenticity of the public key used in verification of digital signature. For this purpose, Certification Authority (CA) sign the public key of the user and act as a third party which can authenticate that public key really belongs to the original user. Certificate contains public key of the user, name of the user, signature of user's public key signed by CA and other parameters according to X.509 standards. As smart card has limited resources therefore public key certificate for smart card typically has size of 1 kilo Byte [10].

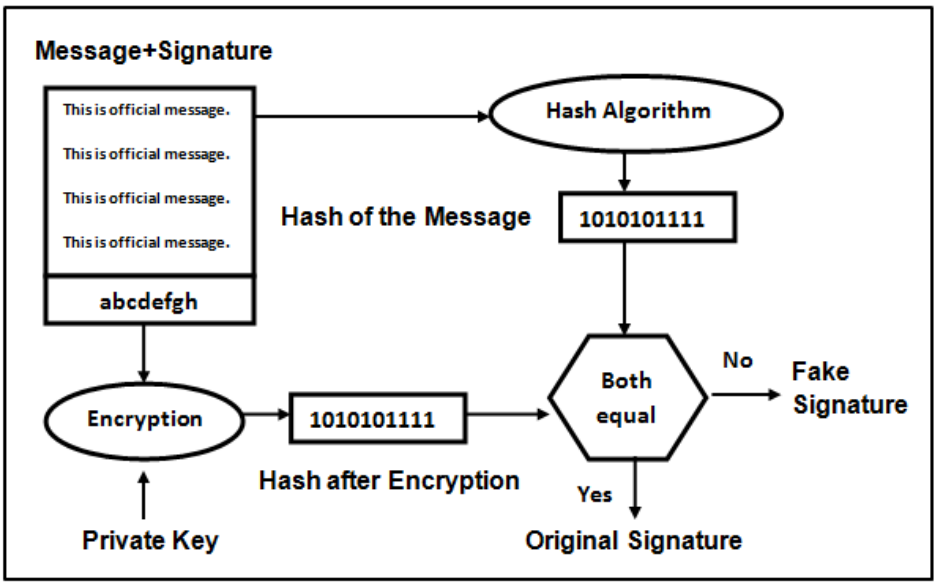


Figure 2.3: Verification Process of Digital Signature

2.7 Cryptographic Algorithms

The strength of underlying cryptographic algorithm relied on the solution of hard mathematical problems like integer factorization and discrete logarithm. Based on these mathematical problems different cryptographic algorithms have been realized today.

2.7.1 RSA

It is the first public key cryptographic algorithm published in 1978 [14]. Its security strength is based on hardness of factorizing large integer numbers and therefore secret key of RSA should be large enough to be broken by modern computers processing power and advanced factorization algorithms.

Encrytion and Decryption

In order to perform encryption on message M, RSA cryptosystem uses following formula:

$$C = M^e \; mod \; N$$

where C = Cipher text, M = plaint text message, e = public exponent, N= length of bits in RSA cryptosystem. While decryption is performed using following formula:

$$M = C^d \; mod \; N$$

where M = Plain text message, C = cipher text, d = private exponent and N = length of bits in RSA cryptosystem.

Key Pair Generation

Key generation process in RSA has been performed in following way:

- Two different large prime numbers p and q are being selected.
- Both prime numbers p and q are being multiplied to generate another number n $(n = p * q)$ equivalent to the length of RSA cryptosystem. For example 1024 bit RSA cryptosystem should contain 1024 bit after multiplication of both selected prime numbers p and q. This number will be publicly available and is used for both encryption and decryption.
- Euler Tortient function of n will be calculated $(\Phi(n) = (p-1)(q-1))$.
- Public exponent, e, will be selected such as e should be relatively prime to Euler Tortient function calculated in previous step or greatest common divisor of e and $\Phi(n)$ should be equal to one. $gcd(\Phi(n), e) = 1$. The value of e should be greater than 1 and less than $\Phi(n)$. $(1 < e < \Phi(n))$. Public exponent e will be available publicly and it is used along with n (e, n) for encryption. Typical values used for public exponent are 3 and $2^{16} + 1$ because both values have only two one bits which helps in fast calculation of encryption.

- Private exponent, d, will be calculated by performing modulo inverse multiplication over public exponent e. Such as $d = e^{-1}\ (mod\ \Phi(n))$. Therefore, $ed = 1\ (mod\ \Phi(n))$. Like e, d should be relatively prime to $\Phi(n)$ and d will be present securely only to the person who keeps private key. Private exponent, d, has typically same length of bits as n.

- Public key is: $PU = (e, n)$

- Private key is: $PR = (d, n)$

Digital Signature

In order to sign a message user utilize her own private key and in verification process public key of the user has been used. Following equations explain RSA signing action:

$$S = M^d\ mod\ N$$

Where S is the signature, M is plain text message, d is private exponent and N is length of cryptosystem constructed by prime number p and q. Following is the verification process of RSA signature:

$$M = S^e\ mod\ N$$

where M is plain text, S is signature, e is public exponent and N is length of RSA cryptosystem.

In order to sign a message, modular exponentiation is performed over message which is quite complex and large operation due to the length of private exponent, d. therefore it is quite hard to implement an modular exponentiation algorithm in constrained environment of smart card. In JCOP smart card RSA signature is generated using CRT (Chinese Remainder Theorem) which is four times faster than a typical modular exponentiation algorithm.

Chinese Remainder Theorem (CRT)

CRT speed up signing process is achieved using following operations:

$$Sp = (M \ mod \ p)^{(dmod(p-1))} \ mod \ p$$
$$Sq = (M \ mod \ q)^{(dmod(q-1))} \ mod \ q$$

Then, combining both calculated Sp and Sq, signature S can be obtained:

$$S = aSp + bSq \ mod \ n$$

where

$$a = 1(mod \ p) \text{ and } b = 0(mod \ p)$$
$$a = 0(mod \ q) \text{ and } b = 1(mod \ q)$$

Alternatively, RSA signature can be generated using CRT scheme as:

$$S = Sq + ((Sp - Sq)q^{-1} \ mod \ p) * q$$

Here, each one of two exponentiations in CRT $(d \ mod \ (p-1))$ and $(d \ mod \ (q-1))$ takes eight times faster than modular exponentiation, d, which is twice in length than both exponentiations of CRT. The disadvantage of CRT scheme is to store pre-calculated more information $(p, q, d \ mod \ (p-1), d \ mod \ (q-1), q^{-1} \ mod \ p)$ in the smart card than only (d, n) in case of modular exponentiation. In JCOP smart card, following variables represent all five components of RSA_CRT scheme:

- $P = p$ (prime factor of n)
- $Q = q$ (prime factor of n)
- $DP1 = d \ mod \ (p-1)$ (CRT exponent 1)
- $DQ1 = d \ mod \ (q-1)$ (CRT exponent 2)
- $PQ = q^{-1} \ mod \ p$ (CRT co-efficient)

Padding Scheme

It is not considered secure to simply apply RSA operations on plain text. Message should be padded with some value or should be converted into

their equivalent hash form. There are two popular schemes for providing padding with RSA signature:

- RSASSA-PSS (Signature with appendix based on Probabilistic Signature Scheme)
- RSASSA-PKCS1-V1_5 (Signature with appendix based on version 1.5 of PKCS#1)

In our implementation of RSA-CRT 1024 bit digital signature, RSASSA-PKCS1-V1_5 padding scheme is used which combines RSASP1 and RSAVP1 primitives with the EMSA-PKCS1-V1_5 encoding method along with SHA1 hash function. In RSASP1, private key is quintuple of (p, q, dP, dQ, qInv) and in RSAVP1, public key is (n, e) [15].

Encoding method

It converts a message into encoded message of specific length. In our case, 1024 bit RSA_CRT_SHA1, 20 bytes (160 bits) hash of message will be converted to 128 bytes (1024 bits) encoded message before performing signing operation. Following is the details of encoding method:

- K = 1024 bits (intended length of encoded message for RSA 1024 bits key)
- M = Plain text message
- H = SHA1(M) (hash of message using SHA1 hash function)
- EM = 00 || 01 || PS || 00 || T

Where EM is encoded message, || is concatenation operator, T is an ASN.1 value of type DigestInfo encoded using the DER (Distinguished Encoding Rules) and is combination of hash function identifier along with hash value of message, PS is a string of octets consisting of values 0xff and the length of PS is K - (length of T +3).

The value of T with hash function SHA1:

T = hash function identifier || hash

Hash function identifier for SHA1 is following 15 Octets in hexadecimal format:

SHA1 = 30 21 30 09 06 05 2b 0e 03 02 1a 05 00 04 14

This will be combined with 20 byte hash value of message:

T = 30 21 30 09 06 05 2b 0e 03 02 1a 05 00 04 14 || H

The length of T is 15 + 20 = 35 bytes and the length of PS is 128 - (35+3) = 90 bytes.

2.7.2 ECDSA

In 1985, Neal Koblitz and Victor Miller had invented Elliptic Curve Cryptosystems (ECC) and security of ECC is based on computational difficulty of Elliptic Curve Discrete Logarithm Problem (ECDLP). In 1992, Scott Vanstone proposed ECDSA against the National Institute of Standards and Technology (NIST) proposal for Digital Signature Standards (DSS). In 1998, ECDSA had been included in International standards Organization (ISO) as ISO 18888-3. In 1999, ECDSA had been included in American National Standards Institute (ANSI) as ANSI X9.62 and in 2000; it is included in Institute of Electrical and Electronic Engineering (IEEE) as IEEE 1363-2000 and also included in U.S. Government Federal Information Processing Standards (FIPS) as FIPS 186-2.

Advantage of ECC over RSA is smaller key size with equivalent security to large key size of RSA. Smaller key size reflects less time in the processing of complex crypto operations and also smaller certificates for public key authentication can be used. Hence ECC provides faster execution and require less storage medium for keys. These benefits make ECDSA more suitable choice for environments where processing power and storage is constrained like smart cards. Elliptic curves are defined over finite fields F_q where q is the order of finite field F. There are Prime Finite Fields F_p where q=p and Binary Finite Field F_2^m where $q = 2^m$. Binary finite fields have two basis representations: Polynomial Basis Representation and Normal Basis Representation. In our case we have utilized elliptic curve over binary finite

field F_2^m with polynomial basis for element representation of chosen finite field.

Domain Parameters

It is important that all parties involved in ECDSA signature generation and verification should comply with similar set of domain parameters. These domain parameters define the elliptic curve over finite field, order of finite field and a base point G over elliptic curve finite field. There are seven ECDSA domain parameters: D = (q, FR, a, b, G, n, h)

- Q: It is field size of elliptic curve; possible values of q can be p (an odd prime) or 2^m. In our case $q = 2^m$, here m is extension degree of binary field F_2^m.
- FR: It is field representation of the elements of finite field F_2^m; possible values can be polynomial representation and normal representation. In our case FR = polynomial representation.
- A, B: These two parameters are field elements in F_q and these defines the equation of elliptic curve over F_q.
- G: It defines a base point (x_G, y_G) over chosen elliptic curve.
- N: It defines the order of base point G.
- H: the co-factor.

In our case of ESDSA, we implemented SHA-1 as hash function and NIST recommended Koblitz curve K-163 with polynomial basis representation as elliptic curve [11]. Details of K-163 curve's parameters and their corresponding values are mentioned below:

Key Pair Generation

ECDSA keys are generated by using a specific set of domain parameters defined in previous section $D = (q, FR, a, b, G, n, h)$. A certificate can be used to authenticate the used domain parameters or mutual agreement of all involving parties on similar set of domain parameters is required. Key generation in ECDSA has following steps:

Curve Parameters	Values of Parameters
m	163
FR	Polynomial basis $f(x) = x_{163} + x_7 + x_6 + x_3 + 1$
n	5846006549323611672814741753598448348329118574063
h	2
a2,b2	1,1
xG	0x 2 fe13c053 7bbc11ac aa07d793 de4e6d5e 5c94eee8
yG	0x 2 89070fb0 5d38ff58 321f2e80 0536d538 ccdaa3d9

Table 2.5: NIST Recommended K-163 Curve Values [11]

- Randomly choose an integer number d within the range $[1, n-1]$.
- Calculate $Q = dG$.

Here, ECDSA private key is d and public key is Q.

Signature Generation

In order to sign a message, m, domain parameters along with key pair (Q, d) should be initialized and then following operations are performed.

1. Randomly choose an integer number k within the range $[1, n-1]$.
2. Perform computation $kG = (x_1, y_1)$.
3. Perform computation $r = x_1 \ mod \ n$, if this results zero jump to step 1
4. Perform computation $k^{-1} \ mod \ n$.
5. Perform computation $e = SHA1(m)$. where $SHA1$ is hash function.
6. Perform computation $s = k^{-1}(e + dr) \ mod \ n$. if this results zero jump to step 1.
7. Signature for message m is (r, s).

Signature Verification

Signature verification process requires domain parameters. In order to verify ECDSA signature, receiver will acquire domain parameters $D = (q, FR, a, b, G, n, h)$ along with public key Q. Receiver can authenticate the domain parameters D along with public key and then following operations are performed.

- Verify that r and s are within the range of $[1, n-1]$.
- Perform computation $e = SHA1(m)$.
- Perform computation $w = s^{-1}\ mod\ n$.
- Perform computation $u_1 = ew\ mod\ n$.
- Perform computation $u_2 = rw\ mod\ n$.
- Perform computation $X(x_2, y_2) = u_1G + u_2Q$.
- Perform computation $v = x_2\ mod\ n$.
- Verification successful if $v = r$.

2.7.3 Secure Hash Algorithm SHA1

In our implementation of 20 byte short and fast digital signature, we have used SHA1 as hash function because it generates 20 bytes hash of message and also used by other signatures (based on ECC and RSA), implemented inside special co-processors in JCOP smart card. SHA1 was published by NIST and issued by FIPS 180-1 in 1995. SHA-1 hash function converts a message having maximum length less than of 2^{64} bits into short hash value also called message digest and the length of message digest is 160 bits (20 bytes). In digital signature, hash function (SHA1) provides data integrity and compression of message which results fast generation of digital signature. It is computationally hard to find collision, two messages having same hash value, therefore if an adversary try to change signed message then verification will fail due to change in computed hash value on received message. Birth day attack on message digest improves the chances of collision to $2^{n/2}$ hash operations where n is length of message digest in bits

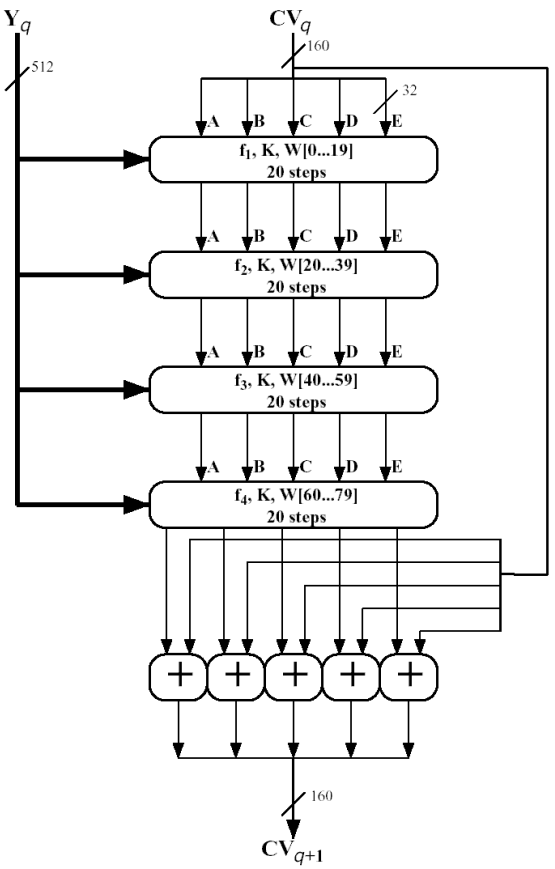


Figure 2.4: SHA-1 module [13]

$(n = 160)$. Another attach published by Wang in 2005 in which a collision of SHA1 is found by performing 2^{69} hash operations [12].

Operations of SHA1

Figure 2.4 depicts SHA-1 architecture having 4 rounds; each round has 20 steps, and total 80 steps are processed in order to generate 160-bit message digest from input message. Input message is divided into exactly 512-bit blocks, M_n, where number of blocks, n, depends on size of input message. The block is further divided into W_n words, where n = 16 and each word is 32 bit long.

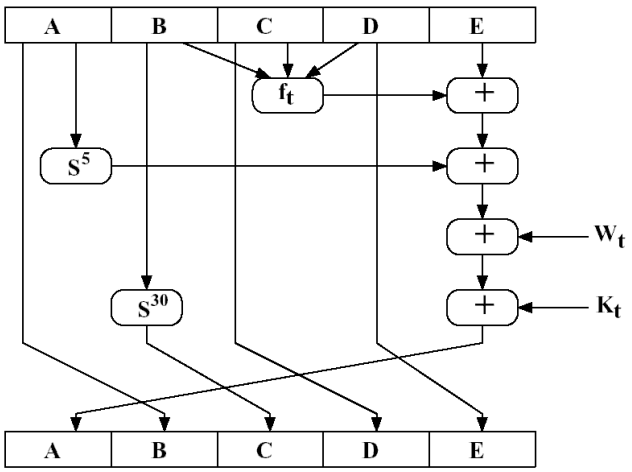


Figure 2.5: SHA-1 Single Step Operation [13]

Padding is required in order to make last block of message equivalent to 512 bits block. The length of original message is also added in the end of last block using 2 words and space between end of message and length of message is filled by 1 followed by zeros. For example: last block contains only two words and last two words are filled with the length of message then remaining 12 words (16 - 2 - 2) are filled with 1 followed by zeros (10000000...).

Each block, M_i is repetitively processed 80 times $(0 \leq t \leq 79)$. Figure 2.5 shows operation performed on a single step of SHA-1 out of 80 steps. The operations performed on single block, Mi, are explained in following 5 steps:

1. Divide M_i in 16 words $W_0, W_1, ..., W_15$.
2. For t = 16 to 79 let $W_t = S1(W_{t-3}\ XOR\ W_{t-8}\ XOR\ W_{t-14}\ XOR\ W_{t-16})$
3. Let $A = H_0, B = H_1, C = H_2, D = H_3, E = H_4$.
4. For t = 0 to 79 do

   $TEMP = S^5(A) + f_t(B; C; D) + E + W_t + K_t$

   $E = D$; $D = C$; $C = S^{30}(B)$; $B = A$; $A = TEMP$

5. Make

   $H_0 = H_0 + A; H_1 = H_1 + B; H_2 = H_2 + C; H_3 = H_3 + D; H_4 = H_4 + E$

Steps 1 and 2

In step1 block of message is divided into 16 words.

In step2 each word from $W_16\ to\ W_79$ is calculated from previous words of block.

Step 3

In step 3 five variables A, B, C, D, E are initialized with H_0, H_1, H_2, H_3, H_4, each one is 32 bit word and initialized with following values:

H0 = 0x67452301

H1 = 0xEFCDAB89

H2 = 0x98BADCFE

H3 = 0x10325476

H4 = 0xC3D2E1F0

Step 4

In Step 4 operations performed, depicted in Figure 2.5 are mentioned. It takes input A, B, C, D, E, W_t, K_t and after processing gives output A, B, C, D, E which are used as input for next step of SHA1. Operations performed in this step are explained below:

$S^n(A)$ is circular left shift operation performed on word A. It shift n bits of A towards left and already existing bits on occupied left side will be shifted toward right.

$F_t(B, C, D)$ is a sequence of logical operations performed on words B, C and D:

f_t(B,C,D) = (B AND C) OR ((NOT B) AND D), $(0 \leq t \leq 19)$

f_t(B;C;D) = B XOR C XOR D, $(20 \leq t \leq 39)$

f_t(B;C;D) = (B AND C) OR (B AND D) OR (C AND D), $(40 \leq t \leq 59)$

f_t(B;C;D) = B XOR C XOR D, $(60 \leq t \leq 79)$

K_t is constant word and its values are:

K_t = 5A827999, $(0 \leq t \leq 19)$

K_t = 6ED9EBA1, $(20 \leq t \leq 39)$

K_t = 8F1BBCDC, $(40 \leq t \leq 59)$

K_t = CA62C1D6, $(60 \leq t \leq 79)$

Step 5

Finally, In Step5, 160 bits message digest $(H_0, H_1, H_2, H_3, H_4)$ is generated by adding the values of words (A, B, C, D, E) generated in the last step with the initial values of the words $(H_0, H_1, H_2, H_3, H_4)$ in respective order.

2.8 Attacks on smart cards

Smart cards are different than other platforms by means of limited computation power, memory and they are dependent on the reader which provides power to smart cards. As smart cards are also implementing co-processor for encryption algorithm in order to provide secure communication. It is necessary to know the relevant attacks on smart cards. Smart card attacks can be categorized in to two major classes.

2.8.1 Invasive Attacks

In this attack, smart card chip is removed from the plastic card and then it is placed in a test bed where micro probing is applied on the chip sur-

face in order to extract secret information. In modern smart cards, applying invasive attacks is extremely expensive and difficult due to more metal layers and having features below the wave length of visible light. In order to apply invasive attacks on these cards, one needs extremely expensive tools like Ion Beam Testers and Electron Beam Testers [18].

2.8.2 Non-invasive Attacks

These attacks do not damage the smart card chip or its plastic body like Invasive attacks. These attacks are more harmful because user of the card has no idea when the secret information has been stolen and this information can be used for bad intentions. Once the security of smart card is compromised then It is also easy to implement these kind of attacks on large scale because of low cost involved. Some examples of Non-invasive attacks are mentioned below:

Timing Analysis

In this attack, time taken by a specific unit to perform a sequence of operations is measured carefully and then this information can be helpful for cryptanalysts in order to obtain the secret key. In smart cards, this attack is considered more effective as more precisely timings can be obtained by using a proprietary reader. In [19], this attack is implemented over RSA signature by carefully measuring the difference in time taken by signatures of various messages. This difference of time is then used to produce secret key.

Power Analysis

Instantaneous power consumption of smart card can be analyzed while it performs cryptographic operations and then this information can be useful in deriving the secret key. This attack can be implemented on smart card by connecting a resistor in series with a smart card and power supply. An oscilloscope is then can be used to analyze the potential difference across the

resistor [20]. Examples of power analysis attacks are Simple Power Analysis (SPA), Differential Power Analysis (DPA) and High Order DFA. Figure 2.6 gives an example of power analysis graph of RSA implementation with square and multiply algorithm.

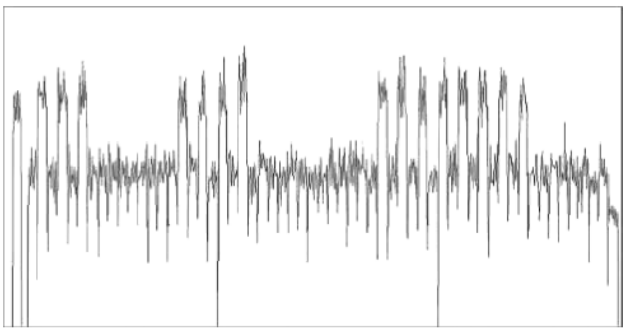


Figure 2.6: The Power Consumption of an RSA with Square and Multiply Algorithm [21]

Chapter 3

Java Cards Technology

Java card is just like any smart card that can execute java applications. Due to the limited processing power and limited memory available on the smart cards only restricted java language features are available on java card language. This implies that both Java Card Virtual Machine (JCVM) and java card API do not have all the functionality of normal Java Virtual Machine (JVM) and Java API. Java cards have been introduced in market first in 1996 by Schlumberger which then transferred the java card specification to the Sun Microsystems.

Java source code contains java programs written by the java developer. The source file is then converter into java class using java compiler. Java class contains byte codes which are independent of underlying machine architecture. Java byte codes are then interpreted into native program code of underlying machine using java virtual machine (JVM). This programming technique offers rapid development of applications without having knowledge of underlying platform complexities and also application program portability due to underlying platform independent java byte codes. All these advantages make java an appropriate choice for smart card applications development.

3.1 JC Application Architecture

Figure 3.1 shows architecture of typical Java Card application comprising of three parts which are discussed below:

Back-end side Applications in back end side provide access to secure information stored in databases or files. Back end applications are connected with host applications residing in reader side

Reader side This part comprises of Host application and Card Acceptance Device (CAD). Host Application provides connectivity among users Java card applet and Back end application.CAD provide power to the java card and also electrical or contactless communication based on RF. CAD can be attached to PC through serial port or it can be a connected to terminal like bank payment terminal along with keypad and screen. Communication between host application and java card applet is performed in Application Protocol Data Unit (APDU).

Card side Java card comprises of java applets, Java card Runtime Environment and Javacard virtual machine and java card operating system. Application functionality resides in java applets. Java card architecture supports multiple applications in the same card.

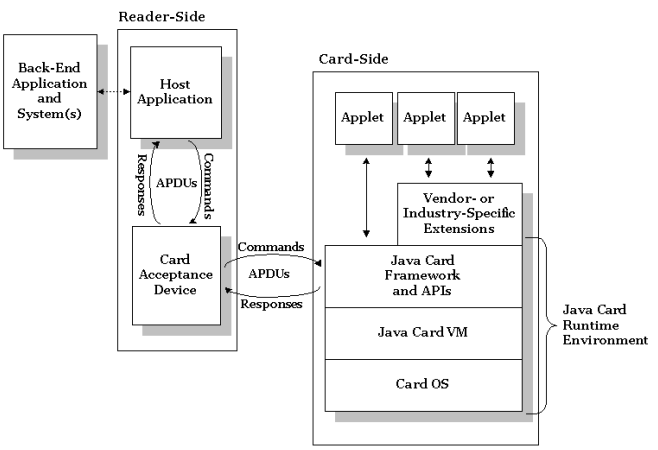


Figure 3.1: Java Card Application Architecture [22]

3.2 Architecture of Java Card

Figure 3.2 discuss details of Java card architecture. Each element of java card architecture is discussed in detail.

3.2.1 Applets

An applet is a java program that conforms the rule defined by Java Card Runtime Environment (JCRE) and is not similar to java applet that run in a web browser. Applets can be stored and updated in a java card even after manufacturing process of the card which is not the case in other embedded systems where applications are burned in ROM during manufacturing process. Applet class javacard.framework.Applet is the super class of all applets placed in java card and it is necessary that all applet classes in java card extend this super class. JCRE supports multi-applications on the same java card which implies that many applets can reside inside the card. As shown in the Figure 3.2 there are loyalty applet, wallet applet and authentication applet. Each applet application can be initialized as a different object. For example authentication applet can be used as an instance user authentication for banking payment and as an instance of user authentication for transit payment.

A typical structure of an applet is defined in figure 3.3 which exhibits important elements of java card applet class (MyApplet). Each applet class must extends from javacard.framework.Applet which contains methods used by JCRE when action from an applet is required. There are five important elements of java card applet class which are discussed below:

install() This method is called by JCRE during applet installation. It creates an instance of Applet subclass.

register() This method is called by JCRE after an applet is installed on java card. It registers the installed applet with JCRE. This method can be called either from install method or from Applet subclass constructor.

select() This method is called by JCRE when a specific applet is need to be selected. This method can also be used for initialization.

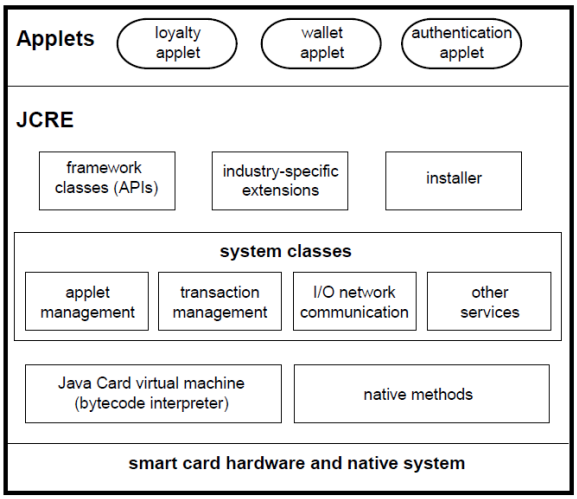


Figure 3.2: Java Card Architecture [23]

deselect() This method is called by JCRE to inform the already selected method that another applet is selected. This method can be used to clean up transient memory occupied by old applet.

process() This method is called by JCRE when functionality implemented inside an applet is called after selecting an applet. This method deals with the transferred command APDU from the reader.

Java applications, other than java card applets, are uniquely identified by Unicode strings based on internet domain naming scheme. Java card applets are uniquely identified and selected by an Application Identifier (AID) on the java card platform. ISO 7816-5 defines standards for AID in order to uniquely identify an applet on java cards. The format is explained in figure 3.4 where Registered Application Provider Identifier (RID) contains 5 bytes which uniquely identify each application provider and other part of AID is Proprietary Application Identifier Extension (PIX). The length of PIX is not fixed but it can be in the range of 0 to 11 bytes. The length of AID can be from 5 to 16 bytes long. In our case the applet AID plus package AID are combined together to form RID. The first four bytes of Applet AID and package AID are similar due to requirement of java card project.

```
import javacard.framework.*
...
public class MyApplet extends Applet {
// Definitions of APDU-related instruction codes
    ...
    MyApplet() {...} // Constructor
    // Life-cycle methods
    install() {...}
    select() {...}
    deselect() {...}
    process() {...}
    // Private methods
    ...
}
```

Figure 3.3: Java Card Applet Structure

Registered Application Provider Identifier (RID)	Proprietary Application Identifier Extension (PIX)
5 bytes	0 to 11 bytes

Figure 3.4: Application Identifier (AID) Format

3.2.2 Java Card Runtime Environment (JCRE)

JCRE manages different functions of java card and is placed in between java applets and hardware of java card as described in figure 3.2. It acts as an operating system and is responsible for applet installation and execution, assigning java card resources to applet, dealing with transactions and network communication between card and reader, and on-card system and applet security. It also provide isolation between underlying technology of smart card and application applets which results portable and easy development of applets for different architectures of smart cards. Industry-specific extensions provide additional services such as Visa open platform

for secure financial services. Framework classes (API) of JCRE provide application programming interface and it consists of API packages for applet development and cryptographic functions. These API packages are compact and customized for smart card application development.

Java Card virtual machine JCVM executes byte code by interpreting them regarding to native programming codes. Native methods provide support to JCVM for low level communication to smart card hardware. The functionalities provided by native methods to JCVM are low level communication protocols, memory management and access to cryptographic units.

3.2.3 Java Card Virtual Machine (JCVM)

Figure 3.5 exhibits Java Card Virtual Machine (JCVM) architecture which separates JCVM into two parts: Off-card VM and On-card VM. If we look into normal java application execution process starting from source code where source code is first converted into byte code and then byte codes are executed by JCVM. In java card, applet source code is written and compiled by the off-card VM which can reside in a desktop PC or workstation and then it is installed and executed by the on-card VM which reside inside java card. Following are steps that describe in detail process of java card applet from creation to execution.

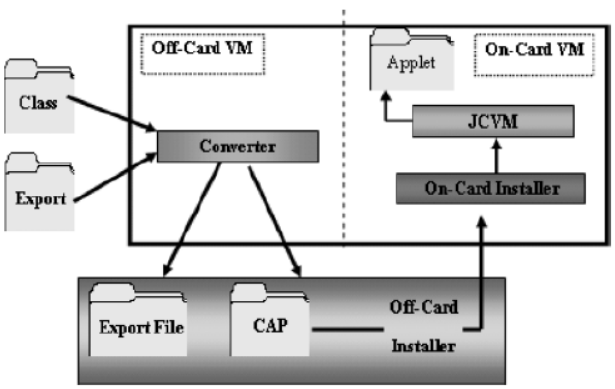


Figure 3.5: Java Card Virtual Machine Architecture

- Developer of java card applet should be careful about limitations of java card API and can write source program in any standard Java IDE.
- Java card applet is compiled in a standard Java compiler that takes source file as input and return an Class file (Java Byte code) along with an Export file (additional information like Header file in C).
- The converter, a component of Off-card VM, takes Class file and Export file as input and return a Converted Applet (CAP) file and Export file. The converter makes sure that input file comply with java card APIs and Java card framework security requirements.
- Finally the off-card installer takes CAP file and Export file as input and together with on-card installer perform all the required steps defined in java card framework to install the applet in java card.
- On-card VM interprets the applet file and executes it.

As JCVM is a subset of JVM therefore it has some limitations described below:

- It does not support data types: char, int, float, double, long.
- It does not support multi-dimensional arrays.
- It does not support garbage collection.
- It does not support threads.
- It does not support arrays having more than 32767 elements.
- It does not support local variables in a method more than 255.

3.3 APDU

In order to perform communication between java card and reader, message-passing is done through the protocol Application Protocol Data Unit (APDU). This is a logical packet that transfers command and response. Information sent from reader to java card is called Command-APDU (C-APDU) and information sent back from java card to reader is called Response-APDU

(R-APDU). The java card acts as a passive entity which responds to the instructions sent by reader which is active entity.

Command APDU

The format of C-APDU is given in figure 3.6. It has two parts one mandatory and one optional. Fields of C-APDU are discussed below:

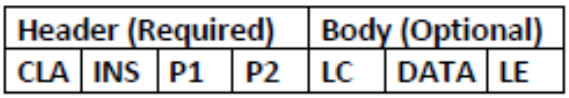


Figure 3.6: Command APDU

CLA It is called class of Instructions and the length of this field is 1 byte. This field is used to define an application specific class of instructions. Standards for valid CLA values have been defined in ISO 7816-4. Application developers of java card should take into account these values which are mentioned below in table 3.1.

CLA Values	**Interpretation**
0x0n, 0x1n	ISO 7816-4 card instructions.
20 to 0x7F	Reserved
0x8n or 0x9n	Customized application-specific instructions.
0xAn	Application- or vendor-specific instructions.
B0 to CF	Customized application-specific instructions.
D0 to FE	Application- or vendor-specific instructions.
FF	Reserved for protocol type selection.

Table 3.1: CLA field Values by ISO 7816-4

INS It is called Instruction Code and the length of this field is 1 byte. This field is used to define instruction against a specific class of instruction set or against a specific value of CLA field. Table 3.2 shows instruction codes

INS Value	Interpretation
0E	Erase Binary
20	Verify
70	Manage Channel
82	External Authenticate
84	Get Challenge
88	Internal Authenticate
A4	Select File
B0	Read Binary
B2	Read Record(s)
C0	Get Response
C2	Envelope
CA	Get Data
D0	Write Binary
D2	Write Record
D6	Update Binary
DA	Put Data
DC	Update Record
E2	Append Record

Table 3.2: INS field Values by ISO 7816-4

for INS field when used against values of CLA field (0x0n, 0x1n) defined as ISO-7816-4 card instructions.

P1, P2 These are called Instruction Parameters and the length of each field is 1 byte. These fields are used to define parameters of a specific instruction.

LC It is called Length of Data field in C-APDU and the length of this field is 1 byte. This field is used to define the length of data sent in Data field in bytes.

Data It is called Data field and the length of this field is not fixed like other fields. The length of Data field can be varying from 0 to 255 bytes. When there is no data to transfer; both LC field and Data field can be empty.

LE It is called Length of Data field in R-APDU and the length of this field is 1 byte. The length of data filed in R-APDU can be controlled using this field.

Response APDU

The format of R-APDU is depicted in figure 3.7 which has one optional and one mandatory part. Fields of each part has been discussed below:

Body (Optional)	Trailer (Required)	
Data	SW1	SW2

Figure 3.7: Response APDU

Data This field contains Data returned in response of java card applet against a query from reader. Normally nature of this field is described in applet codes. It can be empty or it should return some data if LE field in C-APADU contains some value.

SW1, SW2 These fields contain status of processed applet against any C-APDU from reader. Each field has length of 1 byte.

Status codes SW1 and SW2 has been standardized by ISO-7816-4 on generic level and some of which are further explained in detail in table 3.3 where Status column uses following abbreviations.

NP: process completed, normal processing

EE: process aborted, execution error

WP: process completed, warning processing

CE: process aborted, checking error

SW1, SW2	**Status**	**Interpretation**
6281	WP	The returned data may be erroneous.
6282	WP	Fewer bytes than specified by the Le.
6283	WP	The selected file is blocked (invalidated).
6581	EE	Memory error (e.g. during a write operation).
6881	CE	Logical channels not supported.
6882	CE	Secure messaging not supported.
6982	CE	Security state not satisfied.
6983	CE	Authentication method blocked.
6987	CE	Expected secure messaging data objects missing.
6988	CE	Secure messaging data objects incorrect.
6A80	CE	Parameters in the data portion are incorrect.
6A81	CE	Function not supported.
6A82	CE	File not found.
6A83	CE	Record not found.
6A84	CE	Insufficient memory.
6A86	CE	Incorrect P1or P2 parameter.
6A87	CE	Lc inconsistent with P1 or P2.
6A88	CE	Referenced data not found.
6F00	CE	Command aborted with unidentified error.
9000	NP	Command successfully executed.

Table 3.3: Field Values of Processing Status (SW1, SW2)

Chapter 4

Implmentation

Our implementation of MQQ-SIG digital signature is based on NXP JCOP 41 V2.2.1 72K java card and OMNIKEY RFID reader and we have used Eclipse SDK 3.2 as an applet development tool for java card. There are two interfaces in this card; one is contact interface and another is contactless interface. We have used contactless interface for accessing this card from RFID reader.

4.1 NXP JCOP 41 V2.2.1 72K Java Card

The smart card used in our implementation is multi-application (support many applets) and dual interface (contact, contactless) java card. And the name of this smart card is NXP JCOP 41 V2.2.1 72 K. NXP is the manufacturer name of java card. It is a semiconductor company which is founded by Philips in 2006. JCOP is an IBM implementation of Java card 2.2.1 and Global Platform 2.1.1 basic specifications including refinements from VISA international set in the VISA open-platform card implementation guides. All necessary clarification from ISO7816 and EMV 2000 are also incorporated in to the implementation.

The hardware related features of JCOP 41 V.2.2.1 72K has been explained in Philips Semiconductor SmartMX P5CT072 family. The SmartMX family implements the IC architecture of smart card using advanced CMOS

technology having feature size of 0.18 ţm with 5 metal layers. The instruction set is having advanced op-codes and is also compatible with classic 80C51 instruction set. These cards have also included cryptographic co-processors for public and shared secret based encryption such as RSA, ECC, DSA and AES with greater security and low power consumption. All these benefits make these cards ideal choice for applications like e-passport, e-banking, public transportation, payTV and access control.

P5CT072 is a secure PKI smart card controller with temper resistant features. Some features of P5CT072 device are discussed below and also depicted in figure 4.1.

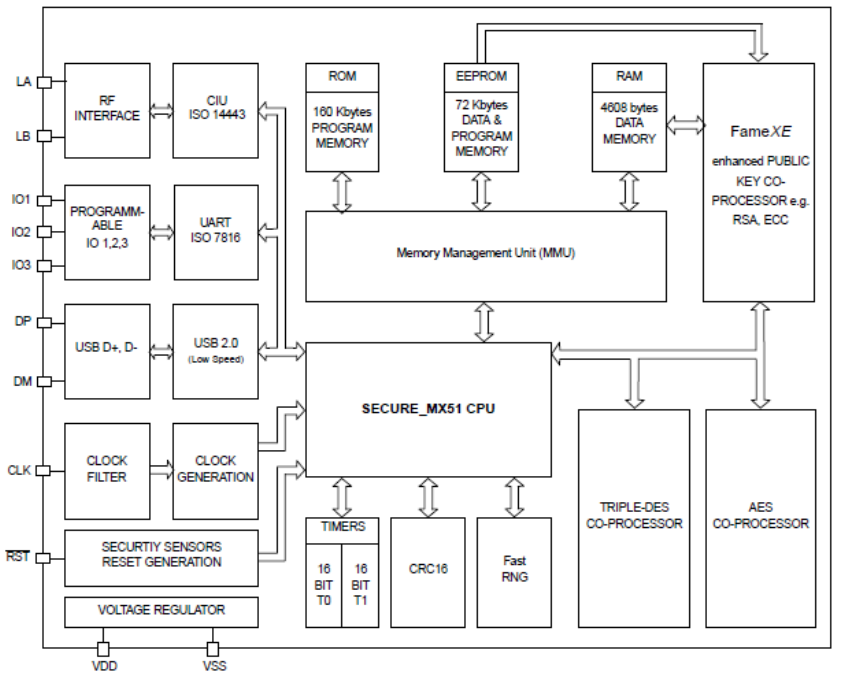


Figure 4.1: Architecture of device P5CT072 [24]

- Three types of memory used. RAM 4608 bytes for volatile storage of data. ROM, 160 Kbytes, burned at manufacturing time and contains OS. EEPROM is non-volatile, 72 Kbytes, for storing applets and data and its data retention time is minimum 20 years.

- Co-processor FameXe for PKI (Public Key Infrastructure) contains RSA and ECC.
- Two Co-processors: one for DES, 3DES and one for AES.
- Dual Interface: Contact interface as defined in ISO/IEC 7816 and Contactless interface as defined in ISO/IEC 14443A, fully supports protocol T=CL as defined in ISO/IEC 14443-4, Data transfer rates: (106, 212, 424) Kbits/sec.
- Power supply required (1.8, 3, 5) volts with maximum external clock frequency of 10MHZ supplied by contact pads or RD-reader in case of contactless access.
- Internal clock frequency is up to 30 MHZ which is independent of external operating frequency of 13.56 MHZ.
- Security features provided are: Low/High clock frequency and temperature sensor. Single Fault Injection (SFI) attack detection. Light Sensors. Memory security (RAM, ROM, and EEPROM) is provided by means of encryption and physical measures.
- Depending on application requirements, three configurations have been provided for P5CT072 device: A, B and B4, Details of these configurations are depicted in figure 4.2. Configuration B and B4 provide compatibility with old MIFARE based infrastructure with 1 Kbytes and 4 Kbytes of memory. In our implementation we have used configuration B.
- Low power Random Number Generator (RNG) implementation in hardware as defined in FIPS 140-2.

4.2 Reader OMNIKEY 5321

We have used OMNIKEY 5321 reader in our implementation which provides support for dual interface: contact and contactless interface for smart card and is connected with PC with USB interface. It operates at 13.56 MHZ frequency for contactless interface. Some features of this reader are discussed below [25]:

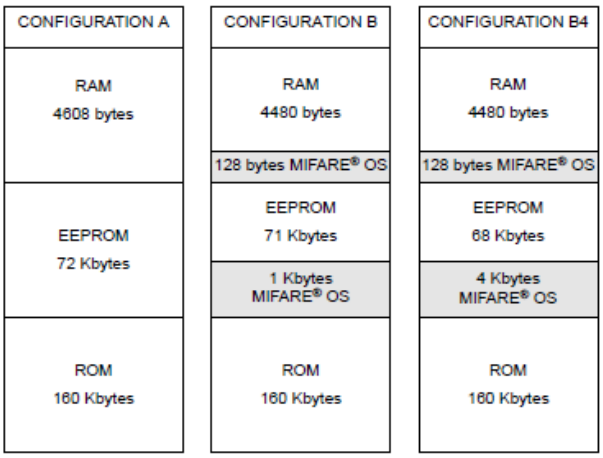


Figure 4.2: Three Memory Configurations of P5CT072 device [24]

- Contactless interface perform read/write operations on 13.56 MHZ. Support for ISO/IEC 14443A and ISO/IEC 14443B standards along with transmission rate up to 848 Kbps.
- Support USB 2.0 with data transmission rate of 12 Mbps.
- Support many OS: Windows, Linux and Mac OS.
- Support API for PC/SC, OCF over PC/SC, CT-API, over PC/SC and Synchronous API.
- Dimensions of reader (L x W x H) = (115 x 96.5 x 25.5) mm and weight of reader is 160 grams.
- Durability is 100,000 insertions and Meantime between Failure (MTBF) is 500,000 hours.
- Support contactless cards from many manufacturers. NXP (MIFARE, DESFire and SMART-MX), ST Micro (x-indent, SR 176, SR 1x 4K), Atmel(AT088RF020).

In our implementation of contactless interface ISO/IEC 14443A standard has been used. As we have used java card that need OpenCard Framework (OCF) and windows 7 operating system which include support for PC/SC driver. Therefore, in our case, Supported API is OCF over PC/SC.

The function of these supported APIs is to provide compatibility between different vendors of smart cards and readers.

4.3 Development Tool IDE (Eclipse SDK 3.2)

We have used Eclipse SDK 3.2 with JCOP tool for writing java program codes for the sake of digital signature implementation on java card because it has been included in the shipment of java card. It supports JRE 1.4.3 or JRE 1.5.x, an old version of Java Runtime Environment (JRE). We have used jre1.5.0_08 which is 5th version with update 8 of JRE. This tool has been installed over Windows 7 operating system and it contains:

- Editor and Debugger for java program.
- Byte code compiler.
- Cap file converter.
- JCOP shell for sending commands from reader to card.

Now, the process of applet installation using JCOP shell will be discussed below:

1. **/card** command connects the reader OMNIKEY 5321 to the JCOP shell.

```
>  /card -a a000000003000000 -c com.ibm.jc.CardManager
resetCard with timeout: 0 (ms)
--Waiting for card...
ATR=3B 8A 80 01 4A 43 4F 50 34 31 56 32 32 31 7F
ATR: T=0, T=1, Hist="JCOP41V221"
 => 00 A4 04 00 08 A0 00 00 00 03 00 00 00 00
 (11910 usec)
 <= 6F 10 84 08 A0 00 00 00 03 00 00 00 A5 04 9F 65
    01 FF 90 00
Status: No Error
```

2. **set-key** command registers the 3DES secret key to the Card Manager which will be used for secure message during authentication. Here default keys have been used.

```
cm>  set-key 255/1/DES-ECB/404142434445464748494a4b4c4d4e4f

255/2/DES-ECB/404142434445464748494a4b4c4d4e4f

255/3/DES-ECB/404142434445464748494a4b4c4d4e4f
```

3. **init-update** command initiate authentication to the Card Manager using Secure Channel Protocol (SCP). In response four fields have been sent back including card challenge for reader and card cryptogram for card authentication.

```
cm>  init-update 255
 => 80 50 00 00 08 09 04 51 35 78 A7 AE 21 00
 (38472 usec)
 <= 00 00 81 29 00 22 37 91 36 54 FF 02 02 12 DC 95
    4D 53 D5 AA 5C E6 00 EF E2 44 4E 8D 90 00
Status: No Error
```

4. **ext-auth** command is used to authenticate reader. Here Plain means no secure messaging is used.

```
cm>  ext-auth plain
 => 84 82 00 00 10 C6 89 04 2D 8A 00 AB 43 59 4D D1
    0C EE 7F D2 97
 (42609 usec)
 <= 90 00
Status: No Error
```

5. After mutual authentication of card and reader installation of applet will be started. First old applet already residing in card is deleted and then package contains applet will be deleted. Here 636172644d is an AID of applet and 6361726441 is an AID of package.

```
cm>  delete 636172644d
 => 80 E4 00 00 07 4F 05 63 61 72 64 4D 00
 (3848464 usec)
 <= 00 90 00
Status: No Error
cm>  delete 6361726441
 => 80 E4 00 00 07 4F 05 63 61 72 64 41 00
 (3734051 usec)
 <= 00 90 00
Status: No Error
```

6. Now, new cap file containing applet and its package will be uploaded into card. The Cap file is uploaded in blocks using data field of C-APDU. Here 523278 μsec is the time taken to upload whole Cap file into card and 2284 bytes is the size of applet in EEPROM of used java card.

```
cm>  upload -d "C:\Eclipse2\DigSigMQQ\bin\packMQQ\javacard\packMQQ.cap"
 => 80 E6 02 00 12 05 63 61 72 64 41 08 A0 00 00 00
    03 00 00 00 00 00 00 00
 (17887 usec)
 <= 00 90 00
Status: No Error
 <= 00 90 00
Status: No Error
Load report:
  5743 bytes loaded in 2.2 seconds
  effective code size on card:
      + package AID       5
```

```
 + applet AIDs       12
 + classes           17
 + methods           1511
 + statics           739
 + exports           0
-----------------------------
   overall           2284  bytes
```

7. After successful loading, applet need to be installed which in turn registers applet with Card Manager.

```
cm>  install -i 636172644d -q C9#() 6361726441 636172644d
 => 80 E6 0C 00 18 05 63 61 72 64 41 05 63 61 72 64
    4D 05 63 61 72 64 4D 01 00 02 C9 00 00 00
 (269405 usec)
 <= 90 00
Status: No Error
```

8. Once applet is installed then it can be selected for execution using its AID.

```
cm>  /select 636172644d
 => 00 A4 04 00 05 63 61 72 64 4D 00
 (9534 usec)
 <= 90 00
Status: No Error
```

9. Finally, the applet can be executed using the appropriate C-APDU, defined inside applet. In our case 80100000 is the C-APDU for executing the digital signature generation process. 20 byte digital signature has been returned along with 2 byte response status word 90 00 which shows successful completion of applet execution. The execution time for the generation of our MQQ based digital signature is shown here (1748615 μs) which is 1.74 seconds approximately.

```
cm>  /send 80100000
 => 80 10 00 00
 (1748615 usec)
 <= E1 8A E7 6C 1C 9E DD 22 82 08 2A BB DA 40 38 C8
    53 F1 BE AC 90 00
Status: No Error
```

4.4 Implementation of MQQ-SIG

We have implemented small MQQ based digital signature (MQQ-SIG) for NXP JCOP 41 V2.2 contactless smart card (java based) using n=160 bits (signature size). Our implementation contains only message signing part of MQQ-SIG inside java card while key pair generation and verification parts of MQQ-SIG are performed on desktop computers. We have been provided

the generated private key and we have stored the private key $(\sigma_1, \sigma_5, *)$ on EEPROM of java card. The length of private key is (160+160+81 = 401) bytes. It should be noted that it is not possible to implement the verification part on java card due to long size of corresponding public key which is 189 Kbytes and we have 72 Kbytes of EEPROM available on used java card. Figure 4.3 shows the MQQ-SIG algorithm for signing message, M.

Algorithm for digital signature with the private key $(\sigma_1, \sigma_k, *)$
Input: A document M to be signed.
Output: A signature $\mathbf{sig} = (x_1, \ldots, x_n)$.
1. Compute $\mathbf{y} = (y_1, \ldots, y_n) = H(M)\|_n$, where M is the message to be signed, $H()$ is a standardized cryptographic hash function such as SHA-1, or SHA-2, with a hash output of not less than n bits. The notation $H(M)\|_n$ denotes the least significant n bits from the hash output $H(M)$. 2. Set $\mathbf{y}' = \mathbf{S}^{-1}(\mathbf{y})$. 3. Represent $\mathbf{y}'$ as $\mathbf{y}' = Y_1 \ldots Y_{\frac{n}{8}}$ where Y_i are Boolean vectors of dimension 8. 4. By using the left and right parastrophes $\backslash$ and $/$ of the quasigroup $*$ compute $\mathbf{x}' = X_1 \ldots X_{\frac{n}{8}}$, such that: $X_1 = Y_1$, $X_j = X_{j-1} \backslash Y_j$, for even $j = 2, 4, \ldots$, and $X_j = Y_j / X_{j-1}$, for odd $j = 3, 5, \ldots$. 5. Compute $\mathbf{x} = \mathbf{S}^{-1}(\mathbf{x}') + \mathbf{v} = (x_1, \ldots, x_n)$. 6. A digital signature of the document M is the vector $\mathbf{sig} = (x_1, \ldots, x_n)$.

Figure 4.3: MQQ-SIG signing algorithm [28]

As we have implemented 160 bits digital signature therefore, in our implementation n=160. For this reason we have selected SHA1 because it generates 160 bit hash of the message, M and this is also the standard hash function used in other digital signature algorithms in java card (RSA, ECDSA). Flow chart 4.4 shows the implementation of the MQQ-SIG algorithm shown in figure 4.3. Implementation details of Steps of figure 4.4 are discussed below:

Step 1

In our case, $y = SHA1(M)$ which is 160 bits hash of the input message, M.

Step 2

Set $y' = S^{-1}(y)$.

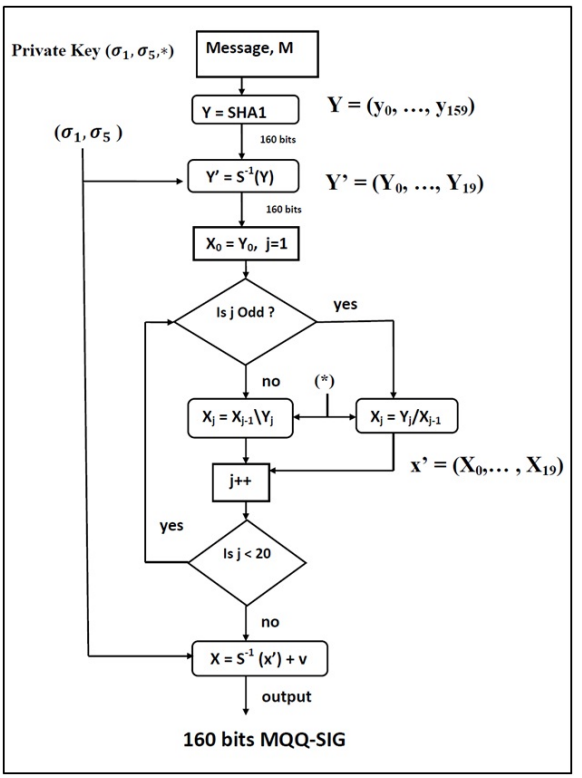


Figure 4.4: Flow Chart of MQQ-SIG signing algorithm Implementation

Here, S^{-1} is non-singular Boolean matrix which is defined by

$$S^{-1} = \sum_{i=1}^{k} I_{\sigma_i}$$

and is discussed in section 2.2.1. In our case k=5 and (σ_1, σ_5) are part of private key. Each one is vector of 160 elements. In our implementation σ_1 (Random Permutation 1) is represented by RP1 and σ_5 (Random Permutation 5) is represented by RP5. We can describe the operations performed in step 2 in words. Each bit of the output in step 1, y (hash of message), is first permutated according to the values of $RP1$ and stored in array $H1$ as shown in figure

$For\ i = 1$ to 20 :

$H2[i] = H2[i] \wedge (H1[i] \wedge H1[(i+4)\%20] \wedge H1[(i+12)\%20] \wedge H1[(i+$

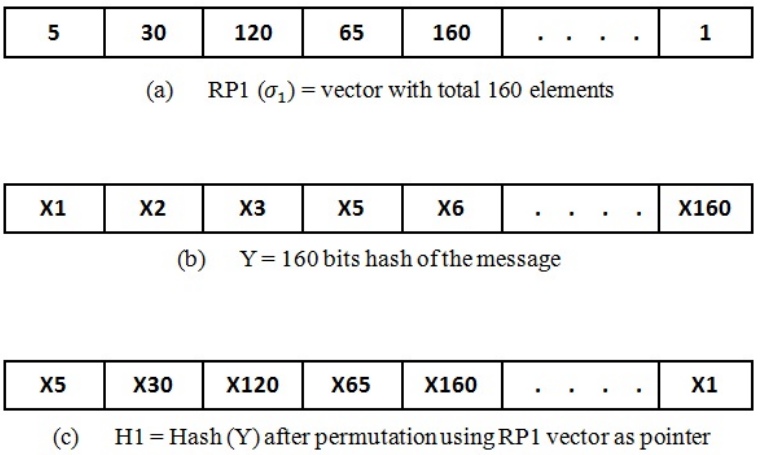


Figure 4.5: Manipulation of the hash using RP1

16)%20])

Where $\wedge$ is exclusive-OR logical operation, and $\%$ is modulus operation.

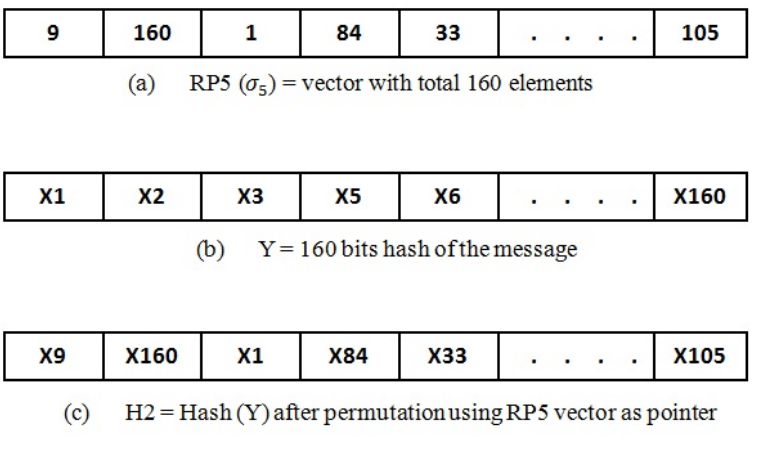


Figure 4.6: Manipulation of the hash using RP5

Step 3

Output of step 2, y', is arranged in array (Y_1 to Y_{20}) where each dimension of array represents a byte.

Step 4

Here, we perform quasigroup operations on the output of step 3 and compute $x' = (X_1 \text{ to } X_{20})$, where $(X_1 = Y_1)$, and other values of $(X_2 \text{ to } X_{20})$ are computed using left and right parastrophes of quasigroup as shown below:

$X_j = X_{j-1} \backslash Y_j$, for even $j = 2, 4, 6, \ldots, 20$

$X_j = Y_j / X_{j-1}$, for odd $j = 3, 5, 7, \ldots, 19$

Now, we will discuss how we have implemented left and right parastrophes (conjugate operations) $\backslash$ and $/$ of the quasigroup $(Q, *)$. The implementation for left and right Parastrophes of the quasigroup is similar, the only difference is the used quasigroup key component of private key $(*)$. The selected 81 bytes constants of quasigroup are used with odd iterations of j while the transpose of selected quasigroup constants are used in even iterations of j. The structure of the 81 bytes quasigroup $(*)$ component of private key is given below:

The quasigroup $(*)$ is divided into eleven elements, ten elements are vectors and these are used for odd iterations of j.

$(A1_{odd}, A2_{odd}, A3_{odd}, \ldots, A8_{odd}, B_{odd}, C_{odd})$.

Each one is byte vector of 8 dimensions $(10 * 8 = 80)$ and the remaining 11th element D_{odd} is scalar byte. All eleven elements together needs $(80 + 1 = 81)$ bytes. Structure of $A1_{odd}$ is given in figure 4.7.

Even iterations of j utilize manipulated form of quasigroup $(*)$ component of private key. The procedure of manipulation is mentioned below:

$A1_{even}$ = $A1_{odd}[0] + A2_{odd}[0] + A3_{odd}[0] + \cdots + A8_{odd}[0]$

$A2_{even}$ = $A1_{odd}[1] + A2_{odd}[1] + A3_{odd}[1] + \cdots + A8_{odd}[1]$

$A3_{even}$ = $A1_{odd}[2] + A2_{odd}[2] + A3_{odd}[2] + \cdots + A8_{odd}[2]$

$\vdots$

$A8_{even}$ = $A1_{odd}[7] + A2_{odd}[7] + A3_{odd}[7] + \cdots + A8_{odd}[7]$

$C_{even} = B_{odd}$,

$B_{even} = C_{odd}$,

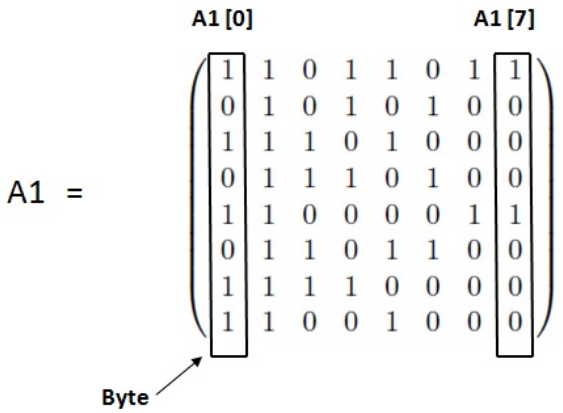


Figure 4.7: Structure of Quasigroup $(*)$ vector A1

$D_{even} = D_{odd}$

Here $(+)$ sign means concatenation. $A1_{even}$ is formed by combining first byte of $A1_{odd}, \ldots, A8_{odd}$ and similarly other $A2_{even}$ to $A8_{even}$ are formed by combining corresponding byte of $A1_{odd}$ to $A8_{odd}$.

There are total 19 iterations in left and right parastrophes and among them 10 are even and nine are odd. Single iteration is explained as follows:

It takes three inputs: previous result of parastrophe X_{j-1}, current iteration, j, and Y_j.

Value of j is examined for even and odd then corresponding quasigroup $(*)$ constants $(A1$ to $A8, B, C, D)$ are considered.

Multiplication and addition binary operations are performed among X_{j-1} and nine elements of quasigroup $(A1$ to $A8, C)$ in GF(2).

$R[0] = A1.X_{j-1} + C[0]$

$R[1] = A2.X_{j-1} + C[1]$

$R[2] = A3.X_{j-1} + C[2]$

$\vdots$

$R[7] = A8.X_{j-1} + C[7]$

Where $C[0]$ is first byte of vector C and $C[7]$ is 8th byte of vector C. In

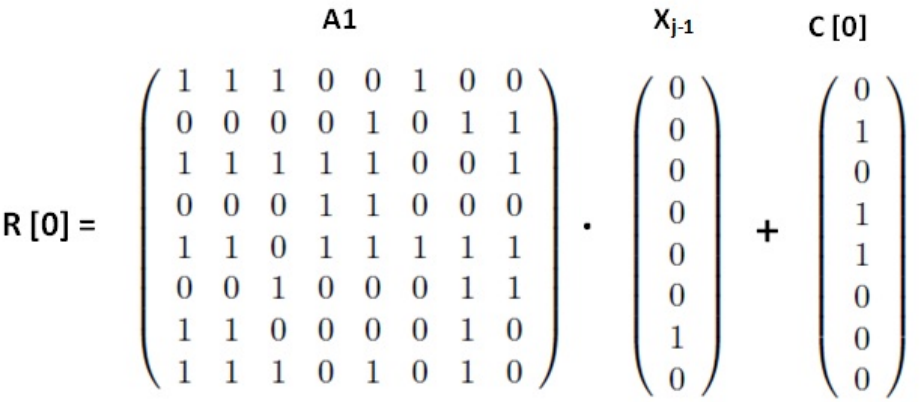


Figure 4.8: Multiplication and Addition among A1, X_{j-1} and C

multiplication mentioned above each byte is considered as column of 8 bits as shown in figure 4.8 where bytes of $A1$, X_{j-1}, and C[0] are interpreted in bits as colomn format. The multiplication is performed using XOR operations. This will result a Boolean matrix with 8 rows and 8 columns where 8 columns shows result of binary opertaions (multiplication and addition) performed among X_{j-1}, $A1$ to $A8$ and C. Therefore, the result, R, is a byte vector of 8 dimensions as shown in figure 4.9.

X_{j-1} is multiplied with other remaining elements of quasigroup B, D and second input Y_j as shown in figure 4.10.

$F = X_{j-1}.B + Y_j + D$

The result of above expression, F, is single column with 8 rows which means a byte as shown in figure 4.11. We have interpreted a byte in bits as a column of 8 rows in our implementation.

After having both results R and F, we have combined them together in a vector T which has 8 dimensions $(T[0] \ldots T[7])$ of type Short (16 bits).

$T[0]$ = 0th bit of $R[0]$ to $R[7]$ + seven zeros (0000000) + 0th bit of F

$T[1]$ = 1st bit of $R[0]$ to $R[7]$ + seven zeros (0000000) + 1st bit of F

$T[2]$ = 2nd bit of $R[0]$ to $R[7]$ + seven zeros (0000000) + 2nd bit of F

$\vdots$

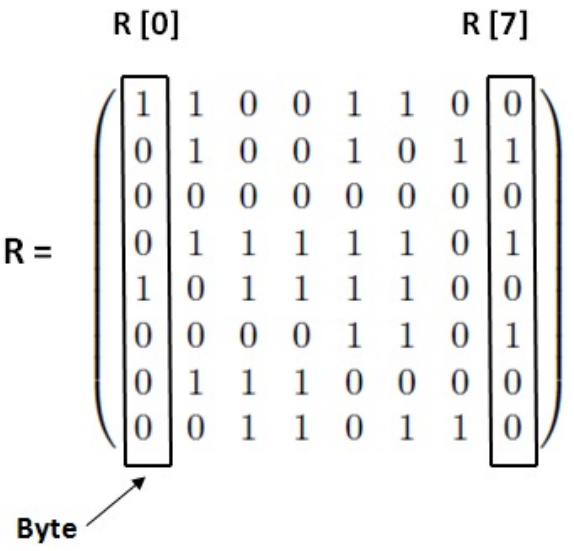


Figure 4.9: Structure of Result Byte vector in columns of bits

$T[7]$ = 7th bit of $R[0]$ to $R[7]$ + seven zeros (0000000) + 7th bit of F

The procedure of building vector T of type Short is also shown in figure 4.12. The idea behing combining both vector R and scalor F into one vector T of type Short is to perform less operations in Guassion elimination Method.

Now vector T represents 8 linear equations that will be solved using Guassion Elimination method.

The LSB of each row of matrix T contains the result after applying Guassion Elimination method. A byte G is constructed by extracting LSB of each row of T matrix as shown in figure 4.13. The MSB of byte G contains value from 1st row and LSB of byte G contains value from last row. This result, G, is the final result of a single iteration, j, and the value of G is then sent to next iteration j. This step is repeated 19 times, 10 times for odd values of iteration j and 9 times for even values of iteration j using corresponding values of Quasigroup $(*)$ part of private key.

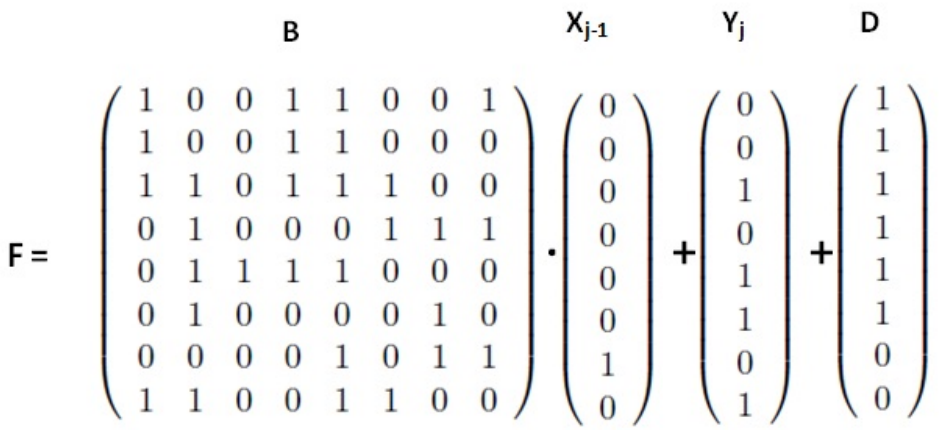


Figure 4.10: Multiplication and Addition among B, X_{j-1}, Y_j and D

Step 5

This step takes input from 20 byte output of previous step x'. The operation is identical as performed in step 2 only a vector v is added to input values. Vector v is formed by extracting 4 LSB from first 40 bytes of RP5 (σ_5). These bits are then combined to form 20 bytes which are then added with input values using XOR operation.

Step 6

The 160 bits (20 byte) output of step 5 is digital signature.

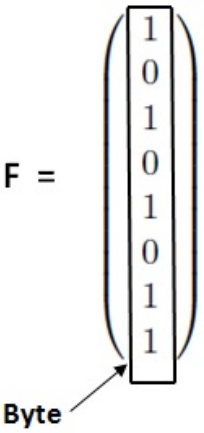


Figure 4.11: Structure of byte F in bits as a colomn

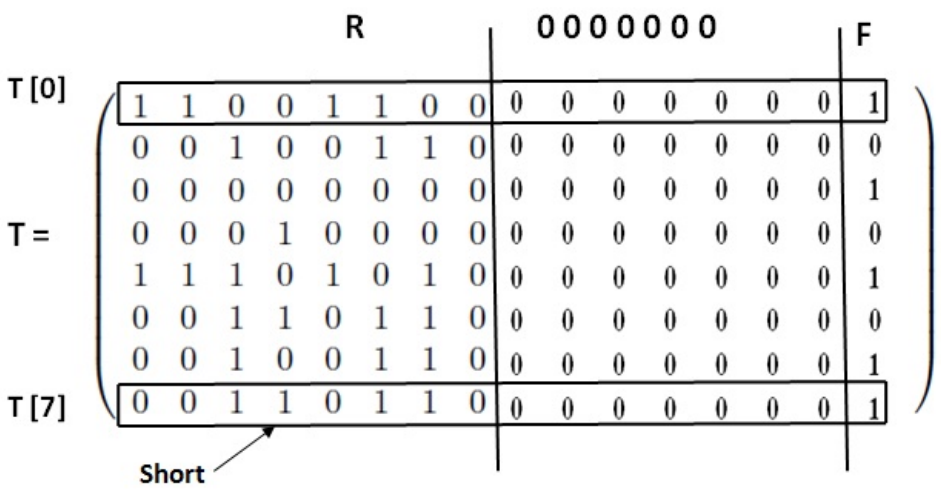


Figure 4.12: Structure of T matrix by combining R and F

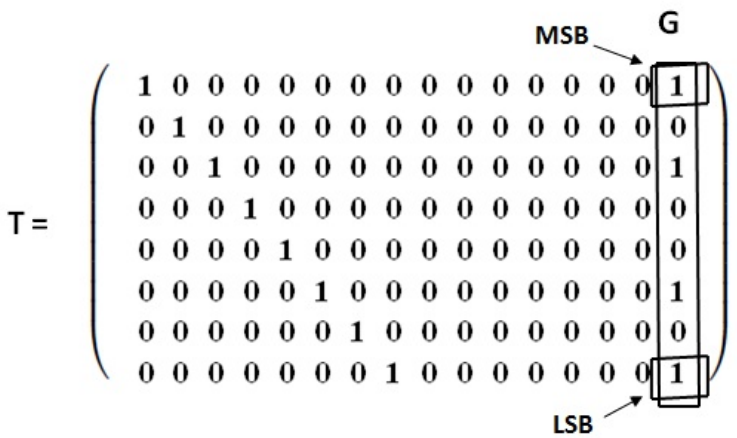


Figure 4.13: T matrix after applying Guassion Elimination

Chapter 5

Evaluation and Discussion of Results

We have implemented MQQ-SIG insdie java card and here we show our results showing required memory space inside java card and the exeution speed of the algorithm. we have used randomly generated messages of 256 bytes inside java card for signing purpose.

5.1 Results

As we have discussed earlier there are three types of memory used in java card architecture. ROM is used while card is manufactured so we do not use this memory in our implementation. We used EEPROM to store our implementation program code and keys used for signing a message. EEPROM is around 30 times slower than RAM in write operations [30]. RAM is used to store temporarily the results of cryptographic operations.

We can divide the MQQ-SIG signing algorithm into three parts:

1. Permutation of of hash $y = SHA1(M)$ by taking into account key components σ_1 and σ_5. This step is referenced as Part1 $(y' = S^{-1}(y))$.

2. Left and right parastrophes of the quasigroup $\backslash$ and $/$ (Even and Odd

operations of quasigroup component of key). This step is referenced as Part2 $(x' = \backslash and/operations)$.

3. Permutation of input along with affine transformation. This step is referenced as Part3 $(x = S^{-1}(x') + v)$.

In the following table 5.1, we have mention average number of operations performed in Java program for each part and also average execution time taken by each part of MQQ-SIG signing algorithm. Here number of operations means addition, multiplication, comparison, modulus and assignment.

Parameters	**Part1**	**Part2**	**Part3**
Avg. No. of Operations	5260	22610	5840
Avg. Time (ms)	158	900	170

Table 5.1: Number of operations and Time of Three Parts of MQQ-SIG

In the following table 5.2, we have mentioned the amount of RAM and EEPROM consumed in bytes for our implementation of MQQ-SIG signing algorithm. The amount of consumed RAM is excluding the 256 bytes of randomly generated message.

EEPROM in Bytes	**RAM in Bytes**
3060	2722

Table 5.2: Memory used in the Signing of MQQ-SIG

We have also tested two already existing signing algorithms RSA and ESDSA which are implemented inside special co-processors in java card using randomly generated messages of 256 bytes. RSA-CRT with 1024 bit private key and ECDSA with K-163 curve recommended by NIST have been used for comparison purpose against MQQ-SIG. The signing speed of these both algorithms are mentioned in milliseconds (ms) on average and digital signature size is mentioned in bits in table 5.3.

The total execution time of MQQ-SIG signing algorithm including hash function SHA1 (implemented inside javacard co-processor) , Input/Output (C-APDU/R-APDU) between javacard and reader, and randomly generated message is on average 1330 ms.

Execution time of applet inside java card and amount of EEPROM consumed by applet is provided by java card JCRE and is shown in section 4.3.

Algorithm	Avg. Signing Time (ms)	Signature Size (bits)
ECDSA K-163	96	384
RSA1024	185	1024
MQQ-SIG160	1228	160

Table 5.3: Comparisons of three algorithm in NXP JCOP 41 V2.2.

In order to find out consumed amount of RAM inside java card we have inserted following codes inside our applet as shown in figure 5.1.

This part of code in applet can be executed using C-APDU (80000000) and it returns R-APDU which contains the free space of RAM and EEPROM as shown below:

```
cm>  /send 80000000
 => 80 00 00 00
 (13247 usec)
 <= 7F FF 06 DE 06 DE 90 00
Status: No Error
```

R-APDU return six bytes in Hex along with (90 00) which shows correct execution response status. These 6 bytes can be divided into three parts and are discussed as follows:

1. 7F FF (32767 in Decimal): These two bytes shows the free space of EEPROM inside java card. Unfortunately these values return 7F FF if the free space in EEPROM is greater than 32767. In our case EEPROM is 72 KBytes and therefore it returns 7FFF in hex which is equivalent to 32767 in decimal. On the other hands we have used consumed amount of EEPROM returned by JCRE at the time of loading applet as shown in section 4.3.
2. 06 DE (1758 in Decimal): This shows the amount of free RAM memory available as clear on reset in bytes. In our used java card, B memory configuration is utilized which has 4480 bytes of available RAM. This information is mentioned in section 4.1. We can calculate consumed amount of RAM by subtracting the free RAM space from total RAM space such as (4480 - 1758 = 2722) bytes.
3. 06 DE (1758 in Decimal): This shows the amount of free RAM as

```
if (buf[ISO7816.OFFSET_INS]==(byte)0x00)
/*** for calculating allocated memory, type (//send b0000000) ***/
            {
                  final short varNull =0;
                  short varLocation = 0;
            /*** Get Persistant memory (EEPROM)  ***/
                  Util.setShort(buf,varLocation,varNull);
                  Util.setShort(buf,varLocation,JCSystem.
getAvailableMemory(JCSystem.MEMORY_TYPE_PERSISTENT));
                  varLocation += 2;
            /*** Transient memory (RAM) clear_on_reset  ***/
                  Util.setShort(buf,varLocation,varNull);
                  Util.setShort(buf,varLocation,JCSystem.
getAvailableMemory(JCSystem.MEMORY_TYPE_TRANSIENT_RESET));
                  varLocation += 2;
            /*** Transient memory (RAM) clear_on_deselect  ***/
                  Util.setShort(buf,varLocation,varNull);
                  Util.setShort(buf,varLocation,JCSystem.
getAvailableMemory(JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT));
                  varLocation += 2;
                  apdu.setOutgoingAndSend(varNull,varLocation);
            }
```

Figure 5.1: Program Code for Revealing Consumed RAM

clear on Deselect in bytes. both values are same that shows the remaining avaliable memory space (1758 bytes) that can be used either as clear on reset or as clear on deselect type.

Objects and Arrays that are declared as Clear on Reset means their used memory space will be clear when the java card session is complete. Java card session remains live until there is communication between card and reader.

Objects and Arrays that are declared as Clear on Deselect means their used memory will be clear when another applet is selected. In that case the previous selected applet will be deselected and its amount of used RAM

will be clear. Our used arrays are declared as Clear on Deselect because we donot want to accomodate RAM space when another applet is selected.

5.2 Evaluation

Java programming language for java card provides architecture independent platform at the cost of slow execution speed. It also lacks some features of typical Java programming due to constrained resources of java card. Lack of garbage collection makes programming difficult and the use of RAM should be done carefully. It is recommended that objects should be initialized once during applet installation phase such as inside install() method or inside constructor. As java card does not support multidimensional arrays, single dimensional arrays should be reused as much as possible in order to reduce memory usage. Comments does not take any memory space in java card and should be used as much as possible for understanding of source code. Java card does not support integer data type and most of the time byte and short data types are being used. This needs careful type casting because constants are by default integers in java. It should be noted that byte data type is by default signed so care must be given while manipulating them as unsigned Byte or using them as a loop driving variable. Another important consideration is usage of if conditions and for loops. These language blocks helps to reduce the code size but here in java card they makes execution of code slower and hence longer execution time. Therefore unnecessary usage of if conditions should be avoided.

We have tried to observe the possiblity of software implementation of MQQ-SIG signing algorithm inside 8-bit NXP JCOP 41 v2.2 contactless smart card with clock frequency upto 30MHZ. It seems that due to slow execution nature of java card the results achieved are not so attractive but if implemented as native code inside ROM of java card then hopefully this causes the increase of 5 to 10 times in execution speed and bring it within few 100 ms. Memory required to store program code and private key is only 3060 bytes for MQQ-SIG making it attractive choice for software implementation over java cards.

Chapter 6

Conclusion and Future Work

We have observered the possiblity of software implementation of a signing part of MQQ-SIG digital signature which is based on recently proposed ultrafast asymmetric algorithm MQQ. We have chosen NXP JCOP 42 java card. We have also made an effort to implement the left and right parastrophes of quasigroup in signing alogorithm which is more crucial and time consuming part of signing algorithm. We have tried to reduce the execution time of this part by efficiently implementing the usage of RAM and EEPROM. Program codes and private key for signing a message has been stored in EEPROM. It has been also observed that usage of if conditions and loop structures can reduce the code size on java card but increases the applet execution time. In our early implementation of left and right parastrophes, The signing algorithm takes around 2571 bytes of EEPROM in java card and its execution time was around 1748 milliseconds. After careful analysis of our code and reducing some possible if conditions and for loops we have achieved 1228 milliseconds execution speed of MQQ-SIG signing algorithm with the increase of EEPROM to 3060 bytes. It should be noted that execution time mentioned here, excludes hash funstion SHA1, message transfering C-APDU and R-APDU between java card and reader, and random message generation inside java card.

In our implementation we have used SHA1 hash function which has been implemented inside java card. We can say that the results shown in our implementation makes it suitable choice for cheap java cards without implementation of co-processors for performing complex cryptographic oper-

ations. We can also conclude that MQQ-SIG signing algorithm including our implementation of left and right parastrophes of quasigroup in signing algorithm if implemented as native code inside smart card can possibly significantly decreases the execution time to few 100 milliseconds.

In our future work we will try to implement MQQ-SIG signing algorithm inside other types of smart cards and we will also try to improve further the execution speed .

Glossary

AID	Application Identifier.
ANSI	American National Standards Institute.
APDU	Application Protocol Data Unit.
BAC	Basic Access control.
C-APDU	Command Application Protocol Data Unit.
CA	Certification Authority.
CAC-C	Common Access Card with Contactless.
CAD	Card Acceptance Device.
CAP	Converted Applet.
CRT	Chinese Remainder Theorem.
DER	Distinguished Encoding Rules.
DOD	Department of Defense.
DPA	Differential Power Analysis.
DSS	Digital Signature Standards.
EAC	Extended Access Control.
ECDLP	Elliptic Curve Discrete Logarithm Problem.
ECDSA	Elliptic Curve Digital Signature Algorithm.
EEPROM	Electrical Erasable Programmable ROM.
EMV	Europay MasterCard and Visa.
EU	European Union.
FIPS	Federal Information Processing Standards.
FRAC	First Responder Authentication Card.

HF	High Frequency.
HODFA	High Order Differential Power Analysis.
HSPD	Homeland Security Presidential Directive.
ICAO	International Civil Aviation Organization.
IEC	International Electronic Committee.
IEEE	Institute of Electrical and Electronic Engineering.
IFF	Identification of Friend or Foe.
ISO	International Standards Organization.
JCOP	Java Card OpenPlatform.
JCRE	Java Card Runtime Environment.
JCVM	Java Card Virtual Machine.
JRE	Java Runtime Environment.
JVM	Java Virtual Machine.
LSB	Least Significant Bit.
MQQ	Multivariate Quadratic Quasigroup.
MQQ-SIG	Multivariate Quadratic Quasigroup Signature.
MSB	Most Significant Bit.
MTBF	Meantime between Failure.
NIST	NationalInstitute for Standards and Technology.
OCF	OpenCard Framework.
OCR	Optical Character Recognition.
PC	Personal Computer.
PIV	Personal Identity Verification.
PIX	Proprietary Application Identifier Extension.
PKI	Public Key Infrastructure.

R-APDU	Response Application Protocol Data Unit.
RAM	Random Access Memory.
RFID	Radio Frequency Identification.
RID	Registered Application Provider Identifier.
RNG	Random Number Generator.
ROM	Read Only Memory.
RP1	Random Permutation 1.
RP5	Random Permutation 5.
RSA	Ron Rivest - Adi Shamir - Len Adleman.
SCP	Secure Channel Protocol.
SHA1	Secure Hash Algorithm 1.
SPA	Simple Power Analysis.
TWIC	TransportationWorker Identification Credential.
UHF	Ultra-High Frequency.
XOR	Exclusive-OR.

Bibliography

[1] Danilo Gligoroski, Svein J. Knapskog, Smile Markovski, *MQQ Digital Signature Scheme*. NTNU, Trondheim, Norway. https://www.ntnu.no/wiki/download/attachments/12815949/Public+Key.pdf, Cited 27 April 2010

[2] International Card Manufacturers Association, *Skimming Fraud*. http://www.icma.com/info/hypercom7801.htm, Cited 27 April 2010.

[3] Federal Information Processing Standards, *Publication 201-1: Personal Identity Verification (PIV) of Federal Employees and Contractors*, March, 2006.

[4] Common Access Card with Contactless. http://www.cac.mil/, Cited 27 April 2010.

[5] NFCTimes, *Contactless Shipment to Break 1 Billion by 2014*, January, 2010. http://www.nfctimes.com/news/contactless-shipments-break-1-billion-2014, Cited 27 April 2010.

[6] International Civil Aviation Organization (ICAO). *Document 9303 Machine Readable Travel Documents (MRTD). Part I: Machine Readable Passports*, 2005.

[7] Bundesamt fĺur Sicherheit in der Informationstechnik. *Advanced Security Mechanisms for Machine Readable Travel Documents Extended Access Control (EAC)*, Technical Guideline TR-03110, September, 2007.

[8] Smart Card Alliance. *Hong Kong Octopus Card*, `http://www.smartcardalliance.org/resources/lib/Hong_Kong_Octopus_Card.pdf`, Cited 27 April 2010.

[9] RFID Journal. *The History of RFID Technology*, `http://www.rfidjournal.com/article/articleview/1338/1/129/`, Cited 26 April 2010.

[10] Wolfgang Rankl, Wolfgang Effing, *Smart Card Handbook*. Wiley, 3rd edition, 2003.

[11] "Digital Signature Standard (DSS)". *Federal Information Standards Processing Publication 186-3*, National Institute of Standards and Technology, June 2009.

[12] Wang, X.; Yin, Y.; and Yu, H. *Finding Collisions in the Full SHA-1*. Proceedings, Crypto '05, 2005; published by Springer-Verlag

[13] Yu Ming-yan, Zhou Tong, Wang An-xiang, Ye Yi-zheng. *AN EFFICIENT ASIC IMPLEMENTATION OF SHA-1 ENGINE FOR TPM*. The 2004 IEEE Asia-Pacific Conference on Circuits and Systems, December 6-9.2004

[14] Rivest, R., Shamir, A., and Adleman, L. M., *Method for obtaining digital signatures and public-key cryptosystems*. Communications of the ACM, 21(2):120-126, 1978.

[15] RFC 3447. *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1*, February 2003.

[16] D. Gligoroski, S. Markovski, and S. J. Knapskog. *Multivariate Quadratic Trap-door Functions Based on Multivariate Quadratic Quasigroups*. In Proceedings of The American Conference on Applied Mathematics, (MATH08), Cambridge, Mas- sachusetts, USA, March 2008.

[17] D. Gligoroski, S. Markovski, and S. J. Knapskog. *Public Key Block Cipher Based on Multivariate Quadratic Quasigroups*. Report 320, Cryptology ePrint Archive, 2008.

[18] M. Kuhn, O. Kömmerling, *Design Principles for Tamper-Resistant Smartcard Processors*, Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard 99), Chicago, Illinois, 1999.

[19] Kocher, P. *Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems*. In Koblitz, N., editor, Advances in Cryptology CRYPTO 96, volume 1109 of Lecture Notes in Computer Science, pages 104-113. Springer-Verlag. 1996.

[20] Kocher, P., Jaffe, J., and Jun, B. *Differential power analysis*. In Wiener, M. J., editor, Advances in Cryptology CRYPTO 99, volume 1666 of Lecture Notes in Computer Science, pages 388-397. Springer-Verlag. 1999.

[21] Book: Keith E. Mayes, Konstantinos Markantonakis, *Smart Cards, Tokens, Security and Applications*. Boston, MA : Springer Science+Business Media, LLC, 2008.

[22] Article. *An Introduction to Java Card Technology*. `URL:http://java.sun.com/javacard/reference/techart/javacard1/`, Cited 20 May 2010.

[23] Book: Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. September, 2000

[24] DataSheet. Philips SmartMx P5CT072. *Secure Dual Interface PKI Smart Card Controller*. Rev. 1.3, October 2004.

[25] Datasheet. *OMNIKEY 5321 Reader*, 2007

[26] J. D. H. Smith, *An introduction to quasigroups and their representations*, Chapman & Hall/CRC, ISBN 1-58488-537-8, 2007.

[27] J. Dénes, A. D. Keedwell, *Latin Squares and their Applications*, English Univer. Press Ltd., 1974.

[28] Danilo Gligoroski, Rune Steinsmo, Ludovic Perret, Jean Charles, Rune Erland, *MQQ-SIG, A Digital Signature Scheme Based on MQQ*, 2010, Reprint.

[29] Yanling Chen, Svein Johan Knapskog, and Danilo Gligoroski, *A Study on Multivariate Quadratic Quasigroups (MQQs)*, Paper submitted in YACC 2010.

[30] Marcus Oestreicher, Ksheerabdhi Krishna, *Object Lifetimes in Java Card*, USENIX Workshop on Smartcard Technology, Chicago, Illinois, USA, May, 1999.

Appendix A

Program Code for MQQ-SIG Digital Signature

```
/**
 *
 */
package packMQQ;
//select cardM 636172644D
import javacard.framework.Applet;
import javacard.framework.ISO7816;
import javacard.framework.ISOException;
import javacard.framework.APDU;
import javacard.framework.JCSystem;
import javacard.framework.Util;
import javacard.security.RandomData;
import javacard.security.MessageDigest;


/**
 * @author kamran
 *
 */
public class AppMQQ extends Applet {
    private short dataSize=(short)256;
    private short hashSize=(short)20;
    private short qMatrixSize = (short)8;

    final static short REG_FAILURE=0x0001;


    //-->> TRANSIENT
    byte[] result_Q=JCSystem.makeTransientByteArray(hashSize,
            JCSystem.CLEAR_ON_DESELECT);
    byte[] data=JCSystem.makeTransientByteArray(dataSize,
```

```
32              JCSystem.CLEAR_ON_DESELECT);
33
34       byte[] HSMul=JCSystem.makeTransientByteArray(hashSize,
35              JCSystem.CLEAR_ON_DESELECT);
36      byte[] e=JCSystem.makeTransientByteArray(qMatrixSize,
37              JCSystem.CLEAR_ON_DESELECT);
38      // Array a in short
39      short[] a=JCSystem.makeTransientShortArray(qMatrixSize,
40              JCSystem.CLEAR_ON_DESELECT);
41      byte[] H1=JCSystem.makeTransientByteArray(hashSize,
42              JCSystem.CLEAR_ON_DESELECT);
43      byte[] hashValue=JCSystem.makeTransientByteArray(hashSize,
44              JCSystem.CLEAR_ON_DESELECT);
45      byte[] filter8=JCSystem.makeTransientByteArray(qMatrixSize,
46              JCSystem.CLEAR_ON_DESELECT);
47      short[] filter16=JCSystem.makeTransientShortArray(qMatrixSize,
48              JCSystem.CLEAR_ON_DESELECT);
49      byte[] fByte=JCSystem.makeTransientByteArray(qMatrixSize,
50              JCSystem.CLEAR_ON_DESELECT);
51
52
53
54      private MessageDigest sha1Hash;
55      private RandomData random;
56
57  //  Matrices for calculing Q function for Odd bytes
58      // Hex values ab ab a5 38 79 88 4f 6c
59      static final byte[] A11 = {
60          (byte)0xAB, (byte)0xAB, (byte)0xA5, (byte)0x38,
61          (byte)0x79, (byte)0x88, (byte)0x4F, (byte)0x6C
62      };
63      // ff ff a5 6c 2d dc 4f 6c
64      static final byte[] A12 = {
65          (byte)0xFF, (byte)0xFF, (byte)0xA5, (byte)0x6C,
66          (byte)0x2D, (byte)0xDC, (byte)0x4F, (byte)0x6C
67      };
68      //36 6c 67 15 4a 77 8d cc
69      static final byte[] A13 = {
70          (byte)0x36, (byte)0x6C, (byte)0x67, (byte)0x15,
71          (byte)0x4a, (byte)0x77, (byte)0x8D, (byte)0xCC
72      };
73      //d2 dc 1e c9 d7 c7 4a 5f
74      static final byte[] A14 = {
75          (byte)0xD2, (byte)0xDC, (byte)0x1E, (byte)0xC9,
76          (byte)0xD7, (byte)0xC7, (byte)0x4A, (byte)0x5F
77      };
78      //a5 f1 a0 be 26 86 44 67
79      static final byte[] A15 = {
80          (byte)0xA5, (byte)0xF1, (byte)0xA0, (byte)0xBE,
81          (byte)0x26, (byte)0x86, (byte)0x44, (byte)0x67
82      };
83      //54 54 44 9d 51 41 1e 0b
84      static final byte[] A16 = {
85          (byte)0x54, (byte)0x54, (byte)0x44, (byte)0x9D,
86          (byte)0x51, (byte)0x41, (byte)0x1E, (byte)0x0B
87      };
88      //88 88 4f 15 88 c7 4f 00
```

```
89      static final byte[] A17 = {
90          (byte)0x88, (byte)0x88, (byte)0x4F, (byte)0x15,
91          (byte)0x88, (byte)0xC7, (byte)0x4F, (byte)0x00
92      };
93      //88 d2 1b 1b dc 9d 41 54
94      static final byte[] A18 = {
95          (byte)0x88, (byte)0xD2, (byte)0x1B, (byte)0x1B,
96          (byte)0xDC, (byte)0x9D, (byte)0x41, (byte)0x54
97      };
98
99      // C11 to C18 for Odd Bytes
100     //02 58 DD AD 8C 30 AB 2D
101     static final byte[] C1 = {
102         (byte)0x02, (byte)0x58, (byte)0xDD, (byte)0xAD,
103         (byte)0x8C, (byte)0x30, (byte)0xAB, (byte)0x2D
104     };
105
106
107     // B1 for Odd Bytes It is equal to C2
108     //E1 3D 08 E8 EB 31 16 92
109     static final byte[] B1 = {
110         (byte)0xE1, (byte)0x3D, (byte)0x08, (byte)0xE8,
111         (byte)0xEB, (byte)0x31, (byte)0x16, (byte)0x92
112     };
113
114     // D1 for Odd Bytes
115     static final byte D1 = (byte)0xFC;
116
117 //  Matrices for calculing Q function for Even bytes
118     // Hex Value ab ff 36 d2 a5 54 88 88
119     static final byte[] A21 = {
120         (byte)0xAB, (byte)0xFF, (byte)0x36, (byte)0xD2,
121         (byte)0xA5, (byte)0x54, (byte)0x88, (byte)0x88
122     };
123     // Hex Value ab ff 6c dc f1 54 88 d2
124     static final byte[] A22 = {
125         (byte)0xAB, (byte)0xFF, (byte)0x6C, (byte)0xDC,
126         (byte)0xF1, (byte)0x54, (byte)0x88, (byte)0xD2
127     };
128     // Hex Value a5 a5 67 1e a0 44 4f 1b
129     static final byte[] A23 = {
130         (byte)0xA5, (byte)0xA5, (byte)0x67, (byte)0x1E,
131         (byte)0xA0, (byte)0x44, (byte)0x4F, (byte)0x1B
132     };
133     // Hex Value 38 6c 15 c9 be 9d 15 1b
134     static final byte[] A24 = {
135         (byte)0x38, (byte)0x6C, (byte)0x15, (byte)0xC9,
136         (byte)0xBE, (byte)0x9D, (byte)0x15, (byte)0x1B
137     };
138     // Hex Value 79 2d 4a d7 26 51 88 dc
139     static final byte[] A25 = {
140         (byte)0x79, (byte)0x2D, (byte)0x4A, (byte)0xD7,
141         (byte)0x26, (byte)0x51, (byte)0x88, (byte)0xDC
142     };
143     // Hex Value 88 dc 77 c7 86 41 c7 9d
144     static final byte[] A26 = {
145         (byte)0x88, (byte)0xDC, (byte)0x77, (byte)0xC7,
```

```
146         (byte)0x86, (byte)0x41, (byte)0xC7, (byte)0x9D
147     };
148     // Hex Value 4f 4f 8d 4a 44 1e 4f 41
149     static final byte[] A27 = {
150         (byte)0x4F, (byte)0x4F, (byte)0x8D, (byte)0x4A,
151         (byte)0x44, (byte)0x1E, (byte)0x4F, (byte)0x41
152     };
153     // Hex Value 6c 6c cc 5f 67 0b 00 54
154     static final byte[] A28 = {
155         (byte)0x6C, (byte)0x6C, (byte)0xCC, (byte)0x5F,
156         (byte)0x67, (byte)0x0B, (byte)0x00, (byte)0x54
157     };
158
159     // C21 to C28 for Even Bytes
160     //E1 3D 08 E8 EB 31 16 92
161     static final byte[] C2 = {
162         (byte)0xE1, (byte)0x3D, (byte)0x08, (byte)0xE8,
163         (byte)0xEB, (byte)0x31, (byte)0x16, (byte)0x92
164     };
165
166     // B2 for Odd Bytes It is equal to C1
167     static final byte[] B2 = {
168         (byte)0x02, (byte)0x58, (byte)0xDD, (byte)0xAD,
169         (byte)0x8C, (byte)0x30, (byte)0xAB, (byte)0x2D
170     };
171
172     // D2 = D1
173     static final byte D2 = (byte)0xFC;
174
175
176
177
178     /* The matrix SInv is stored in ROM as two onedimensional
179     arrays RP1[] and RP5[] of 160 bytes */
180     static final byte[] RP1 =
181     {
182     (byte)111, (byte)137, (byte)49,  (byte)134, (byte)9,   (byte)116,
183     (byte)11,  (byte)52,  (byte)43,  (byte)55,  (byte)74,  (byte)130,
184     (byte)119, (byte)144, (byte)31,  (byte)7,   (byte)72,  (byte)79,
185     (byte)105, (byte)59,  (byte)57,  (byte)120, (byte)50,  (byte)94,
186     (byte)141, (byte)135, (byte)149, (byte)44,  (byte)109, (byte)100,
187     (byte)113, (byte)1,   (byte)143, (byte)126, (byte)117, (byte)37,
188     (byte)65,  (byte)67,  (byte)152, (byte)107, (byte)10,  (byte)98,
189     (byte)15,  (byte)23,  (byte)138, (byte)19,  (byte)121, (byte)18,
190     (byte)28,  (byte)156, (byte)123, (byte)106, (byte)48,  (byte)29,
191     (byte)97,  (byte)34,  (byte)85,  (byte)157, (byte)64,  (byte)3,
192     (byte)60,  (byte)35,  (byte)24,  (byte)32,  (byte)108, (byte)147,
193     (byte)158, (byte)21,  (byte)129, (byte)84,  (byte)5,   (byte)70,
194     (byte)118, (byte)112, (byte)30,  (byte)68,  (byte)47,  (byte)40,
195     (byte)150, (byte)13,  (byte)61,  (byte)73,  (byte)132, (byte)22,
196     (byte)95,  (byte)153, (byte)4,   (byte)76,  (byte)87,  (byte)114,
197     (byte)127, (byte)62,  (byte)27,  (byte)36,  (byte)125, (byte)45,
198     (byte)142, (byte)39,  (byte)101, (byte)63,  (byte)88,  (byte)96,
199     (byte)12,  (byte)115, (byte)82,  (byte)91,  (byte)159, (byte)93,
200     (byte)155, (byte)154, (byte)148, (byte)110, (byte)25,  (byte)0,
201     (byte)41,  (byte)20,  (byte)54,  (byte)26,  (byte)14,  (byte)83,
202     (byte)81,  (byte)80,  (byte)131, (byte)33,  (byte)78,  (byte)77,
```

```
203     (byte)124, (byte)104, (byte)133, (byte)17,  (byte)145, (byte)139,
204     (byte)122, (byte)102, (byte)42,  (byte)56,  (byte)75,  (byte)66,
205     (byte)2,   (byte)16,  (byte)86,  (byte)140, (byte)71,  (byte)136,
206     (byte)69,  (byte)99,  (byte)58,  (byte)6,   (byte)92,  (byte)90,
207     (byte)8,   (byte)103, (byte)128, (byte)38,  (byte)46,  (byte)146,
208     (byte)89,  (byte)151, (byte)51,  (byte)53};
209
210     static final byte[] RP5 =
211     {
212     (byte)90,  (byte)113, (byte)130, (byte)115, (byte)132, (byte)27,
213     (byte)46,  (byte)72,  (byte)33,  (byte)50,  (byte)35,  (byte)136,
214     (byte)42,  (byte)148, (byte)146, (byte)143, (byte)116, (byte)158,
215     (byte)98,  (byte)41,  (byte)39,  (byte)5,   (byte)54,  (byte)86,
216     (byte)106, (byte)56,  (byte)30,  (byte)138, (byte)80,  (byte)44,
217     (byte)91,  (byte)49,  (byte)1,   (byte)149, (byte)159, (byte)101,
218     (byte)74,  (byte)9,   (byte)110, (byte)131, (byte)25,  (byte)51,
219     (byte)123, (byte)76,  (byte)104, (byte)28,  (byte)82,  (byte)140,
220     (byte)2,   (byte)108, (byte)120, (byte)144, (byte)10,  (byte)145,
221     (byte)124, (byte)119, (byte)62,  (byte)57,  (byte)117, (byte)121,
222     (byte)17,  (byte)73,  (byte)105, (byte)69,  (byte)155, (byte)7,
223     (byte)154, (byte)75,  (byte)100, (byte)141, (byte)157, (byte)38,
224     (byte)14,  (byte)60,  (byte)47,  (byte)112, (byte)95,  (byte)85,
225     (byte)43,  (byte)93,  (byte)24,  (byte)12,  (byte)4,   (byte)71,
226     (byte)81,  (byte)13,  (byte)94,  (byte)68,  (byte)107, (byte)67,
227     (byte)142, (byte)150, (byte)61,  (byte)6,   (byte)122, (byte)26,
228     (byte)139, (byte)59,  (byte)102, (byte)153, (byte)109, (byte)48,
229     (byte)103, (byte)65,  (byte)23,  (byte)92,  (byte)87,  (byte)40,
230     (byte)135, (byte)133, (byte)129, (byte)134, (byte)8,   (byte)55,
231     (byte)83,  (byte)125, (byte)31,  (byte)96,  (byte)147, (byte)36,
232     (byte)0,   (byte)126, (byte)70,  (byte)64,  (byte)20,  (byte)11,
233     (byte)137, (byte)78,  (byte)89,  (byte)58,  (byte)21,  (byte)114,
234     (byte)127, (byte)111, (byte)99,  (byte)34,  (byte)152, (byte)79,
235     (byte)66,  (byte)97,  (byte)22,  (byte)15,  (byte)151, (byte)32,
236     (byte)84,  (byte)37,  (byte)77,  (byte)88,  (byte)16,  (byte)29,
237     (byte)3,   (byte)128, (byte)118, (byte)18,  (byte)156, (byte)19,
238     (byte)52,  (byte)45,  (byte)53,  (byte)63};
239
240
241 //        ***********START  CONSTRUCTOR↩
        ******************************///
242         private AppMQQ(byte[] bArray,short bOffset,byte bLength){
243
244
245             //Register applet
246                     try {
247                         register();
248                     } catch (Exception e) {
249                         ISOException.throwIt(REG_FAILURE);
250                     }
251
252
253         sha1Hash = MessageDigest.getInstance(MessageDigest.ALG_SHA,false↩
                );
254         random =  RandomData.getInstance(RandomData.ALG_PSEUDO_RANDOM);
255
256
257         }
```

```
258 //         ***********END  CONSTRUCTOR*******************************///
259
260     public static void install(byte[] bArray, short bOffset, byte ↩
            bLength) {
261         // GP-compliant JavaCard applet registration
262         new AppMQQ(bArray, bOffset, bLength);
263     }
264
265     public void process(APDU apdu) {
266         // Good practice: Return 9000 on SELECT
267         if (selectingApplet()) {
268             return;
269         }
270
271         byte[] buf = apdu.getBuffer();
272         // Check CLA field against perticular value 80 in Hex
273         if (buf[ISO7816.OFFSET_CLA]!=(byte)0x80)
274             ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
275
276         //Check INS field against perticular value 10 in Hex for ↩
                following instruction
277         if (buf[ISO7816.OFFSET_INS]==(byte)0x10)
278         {
279               filter8[0] = (byte)0x80;  filter8[1]=(byte)0x40;
280               filter8[2] = (byte)0x20;  filter8[3]=(byte)0x10;
281               filter8[4] = (byte)0x08;  filter8[5]=(byte)0x04;
282               filter8[6] = (byte)0x02;  filter8[7]=(byte)0x01;
283
284             filter16[0] = (short)0x8000; filter16[1] = (short)0x4000;
285             filter16[2] = (short)0x2000; filter16[3] = (short)0x1000;
286             filter16[4] = (short)0x0800; filter16[5] = (short)0x0400;
287             filter16[6] = (short)0x0200; filter16[7] = (short)0x0100;
288
289
290             /**** Generate 256 byte random data *****/
291             random.generateData(data,(short)0,dataSize);
292             sha1Hash.doFinal(data,(short)0,dataSize, HSMul, (short)0);
293
294         // calculate sha-1 hash of input data and store result in ↩
                hashValue
295             sha1Hash.doFinal(data,(short)0,dataSize, hashValue, (short)↩
                    0);
296
297             /******* Affine transformation is just for
298              ******* the second call. The constant is extracted
299              ******* from the 4 LSBs of the first 40 bytes of
300              ******* RP5[] and xor-ed to input_bytes[]. ********/
301
302             inverseAffineTransformation(hashValue,(byte)0);
303
304             byte i;
305             //Following function perform Q operation 19 times
306             result_Q[0] = HSMul[0];
307             for (i=1; i<20; i++){
308                 result_Q[i] = perform_Q_Operation(i,result_Q[i-1],HSMul[↩
                        i]);
309             }
```

```
310
311
312
313             /*** Now Result of Q operation will be multiplied with S matrix
314             **** in addition to affine transformation
315             **** in order to generate digital signature in HSMul[] of 20 index ***/
316             inverseAffineTransformation(result_Q,(byte)1);
317
318             Util.arrayCopy(HSMul,(short)0,buf,(short)0,hashSize);
319             apdu.setOutgoingAndSend((short)0,hashSize);
320
321
322         /************ This will find out used memories in javacard ********/
323         } else if (buf[ISO7816.OFFSET_INS]==(byte)0x00)
324             // for observing allocated memory type //send 80000000
325         {
326             final short varNull =0;
327             short varLocation = 0;
328             //Get Persistant memory (EEPROM)
329             Util.setShort(buf,varLocation,varNull);
330             Util.setShort(buf,varLocation,JCSystem.getAvailableMemory(JCSystem.MEMORY_TYPE_PERSISTENT));
331             varLocation += 2;
332
333             //Transient memory (RAM) clear on reset
334             Util.setShort(buf,varLocation,varNull);
335             Util.setShort(buf,varLocation,JCSystem.getAvailableMemory(JCSystem.MEMORY_TYPE_TRANSIENT_RESET));
336             varLocation += 2;
337
338             //Transient memory (RAM) clear on deselect
339             Util.setShort(buf,varLocation,varNull);
340             Util.setShort(buf,varLocation,JCSystem.getAvailableMemory(JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT));
341             varLocation += 2;
342             apdu.setOutgoingAndSend(varNull,varLocation);
343
344         } else
345             ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
346
347     }
348     /*********** end of main Process method *************************/
349
350
351     /****** Following method perform MQQ Quasi operation 20 times based on
352      ****** Odd and Even call  It uses different matrices **************
353
354     private byte perform_Q_Operation(byte num, byte b1, byte b2){
355         //Perform Q operation
356         byte i,j,f,totRows,totCols;
357         totRows=8;
358         totCols=8;
```

```
359         f=(byte)0;
360
361
362         if ((num%2) != 0) {
363         /*** This perform matrix multiplication for Odd call values ***/
364
365             for (i=0;i<totCols;i++)
366             {
367                 e[i] = C1[i];
368                 a[i] = 0;
369             }
370
371             for (i=0;i<totCols;i++){
372             if ((b1 & filter8[i]) != 0){
373                 f = (byte)(f ^ B1[i]);
374                 e[0] = (byte)(e[0] ^ A11[i]) ;
375                 e[1] = (byte)(e[1] ^ A12[i]) ;
376                 e[2] = (byte)(e[2] ^ A13[i]) ;
377                 e[3] = (byte)(e[3] ^ A14[i]) ;
378                 e[4] = (byte)(e[4] ^ A15[i]) ;
379                 e[5] = (byte)(e[5] ^ A16[i]) ;
380                 e[6] = (byte)(e[6] ^ A17[i]) ;
381                 e[7] = (byte)(e[7] ^ A18[i]) ;
382             }
383         }
384
385         f= (byte)(f ^ D1);
386         f= (byte)(f ^ b2);
387
388
389         }
390         /*** end of if for checking odd value and Odd matrices ↩
                Operations ***/
391
392         else
393         {
394         /*** This perform matrix multiplication for Even call values **↩
                */
395
396             for (i=0;i<totCols;i++)
397             {
398                 e[i] = C2[i];
399                 a[i] = 0;
400             }
401
402             for (i=0;i<totCols;i++){
403                 if ((b1 & filter8[i]) != 0){
404                     f = (byte)(f ^ B2[i]);
405                     e[0] = (byte)(e[0] ^ A21[i]) ;
406                     e[1] = (byte)(e[1] ^ A22[i]) ;
407                     e[2] = (byte)(e[2] ^ A23[i]) ;
408                     e[3] = (byte)(e[3] ^ A24[i]) ;
409                     e[4] = (byte)(e[4] ^ A25[i]) ;
410                     e[5] = (byte)(e[5] ^ A26[i]) ;
411                     e[6] = (byte)(e[6] ^ A27[i]) ;
412                     e[7] = (byte)(e[7] ^ A28[i]) ;
413                 }
```

```
414             }
415
416
417             f= (byte)(f ^ b2);
418             f= (byte)(f ^ D2);
419
420
421         }// end of else for even value
422
423
424     /********** 16 bit implementation of Guassion Elimination method ↩
            *******/
425     // Converting columns array (e) into row array (a)
426         short makeshort;
427
428
429         makeshort = (short)((e[0]&0x80) | (((e[1]&0x80)>>1)&0x7F)
430                 | (((e[2]&0x80)>>2)&0x3F) | (((e[3]&0x80)>>3)&0x1F)
431                 | (((e[4]&0x80)>>4)&0x0F) | (((e[5]&0x80)>>5)&0x07)
432                 | (((e[6]&0x80)>>6)&0x03) | (((e[7]&0x80)>>7)&0x01)) ;
433         a[0] = (short)((makeshort<<8) | (short)(((f & 0x80)>>7)&0x01));
434
435         makeshort = (short)(((e[0]&0x40)<<1) | (e[1]&0x40)
436                 | (((e[2]&0x40)>>1)&0x7F) | (((e[3]&0x40)>>2)&0x3F)
437                 | (((e[4]&0x40)>>3)&0x1F) | (((e[5]&0x40)>>4)&0x0F)
438                 | (((e[6]&0x40)>>5)&0x07) | (((e[7]&0x40)>>6)&0x03)) ;
439         a[1] = (short)((makeshort<<8) | (short)((f & 0x40)>>6));
440
441         makeshort = (short)(((e[0]&0x20)<<2) | ((e[1]&0x20)<<1)
442                 | (e[2]&0x20) | (((e[3]&0x20)>>1)&0x7F)
443                 | (((e[4]&0x20)>>2)&0x3F) | (((e[5]&0x20)>>3)&0x1F)
444                 | (((e[6]&0x20)>>4)&0x0F) | (((e[7]&0x20)>>5)&0x07)) ;
445         a[2] = (short)((makeshort<<8) | (short)((f & 0x20)>>5));
446
447         makeshort = (short)(((e[0]&0x10)<<3) | ((e[1]&0x10)<<2)
448                 | ((e[2]&0x10)<<1) | (e[3]&0x10)
449                 | (((e[4]&0x10)>>1)&0x7F) | (((e[5]&0x10)>>2)&0x3F)
450                 | (((e[6]&0x10)>>3)&0x1F) | (((e[7]&0x10)>>4)&0x0F)) ;
451         a[3] = (short)((makeshort<<8) | (short)((f & 0x10)>>4));
452
453         makeshort = (short)(((e[0]&0x08)<<4) | ((e[1]&0x08)<<3)
454                 | ((e[2]&0x08)<<2) | ((e[3]&0x08)<<1)
455                 | (e[4]&0x08) | (((e[5]&0x08)>>1)&0x7F)
456                 | (((e[6]&0x08)>>2)&0x3F) | (((e[7]&0x08)>>3)&0x01F)) ;
457         a[4] = (short)((makeshort<<8) | (short)((f & 0x08)>>3));
458
459         makeshort = (short)(((e[0]&0x04)<<5) | ((e[1]&0x04)<<4)
460                 | ((e[2]&0x04)<<3) | ((e[3]&0x04)<<2)
461                 | ((e[4]&0x04)<<1) | (e[5]&0x04)
462                 | (((e[6]&0x04)>>1)&0x7F) | (((e[7]&0x04)>>2)&0x3F)) ;
463         a[5] = (short)((makeshort<<8) | (short)((f & 0x04)>>2));
464
465         makeshort = (short)(((e[0]&0x02)<<6) | ((e[1]&0x02)<<5)
466                 | ((e[2]&0x02)<<4) | ((e[3]&0x02)<<3)
467                 | ((e[4]&0x02)<<2) | ((e[5]&0x02)<<1)
468                 | (e[6]&0x02) | (((e[7]&0x02)>>1)&0x7F)) ;
469         a[6] = (short)((makeshort<<8) | (short)((f & 0x02)>>1));
```

```
470
471         makeshort = (short)(((e[0]&0x01)<<7) | ((e[1]&0x01)<<6)
472                 | ((e[2]&0x01)<<5) | ((e[3]&0x01)<<4)
473                 | ((e[4]&0x01)<<3) | ((e[5]&0x01)<<2)
474                 | ((e[6]&0x01)<<1) | (e[7]&0x01)) ;
475         a[7] = (short)((makeshort<<8) | (short)(f & 0x01));
476
477
478     // Applying Guassion Elimination Method for upper Triangle
479     short fBit = 0x0001;
480     byte row,col,check;
481     short temp;
482     for (col=0;col<totCols-1;col++){
483         check=0;
484         for (row=col;row<totRows;row++){
485             if ((a[row] & filter16[col]) != 0) // && check == 0){
486             {    if (check == 0){
487                 check=1;
488                 temp = a[row];
489                 a[row] = a[col];
490                 a[col] = temp;
491                 continue;
492             } else
493                 a[row] = (short)(a[row] ^ a[col]);
494             }
495         }
496     }
497
498     //Convert upper triangle to all zeros
499     for (col=7; col > 0; col--)
500         for(row=(byte)(col-1);row>=0;row--)
501             if ((a[row] & filter16[col]) != 0)
502                 a[row] = (short)(a[row] ^ a[col]);
503
504
505     // Converting least significant bit of a[] into byte result
506     byte result;
507     result=0;
508     for (i=0;i<totRows;i++){
509         if ((a[i] & fBit) != 0)
510             result = (byte)(result | filter8[i]);
511     }
512
513     return result;
514 }
515
516     /***** End of 16 bit implementation of Guassion elimination method ↩
            *****/
517
518     /********** End of perform_Q_Operation function ***************/
519
520     /******** Following function accept hash of data and
521      ******** perform multiply with SInv (two permutations of 160 ↩
              elements)
522      ******** matrix and stores result
523      ******** in HSMul[] of index 20 *********/
524
```

```
525     private void inverseAffineTransformation(byte[] InputBytes,byte ↩
            second_call) {
526         /* The matrix SInv is given as two permutations of 160 elements.↩
                */
527         byte j, byteindex,byteIndexD, bitindex,bitIndexD;
528         short index;
529 
530         /* Initialize H1 and HSMUL = 0 */
531         for (j=0; j<20; j++) {
532             H1[j]=0;
533             HSMul[j]=0;
534         }
535 
536         byteindex=0;
537         /********** Affine transformation is just for the second call.
538          ***** The constant is extracted from the 4 LSBs of the first
539          ***** 40 bytes of RP5[] and xor-ed to input_bytes[] **********/
540         if (second_call == 1)
541             for (j=0; j<20; j++)
542             {
543                 InputBytes[j] ^= ((RP5[byteindex]<<4)|(RP5[byteindex↩
                        +1]&0x0F));
544                 byteindex += 2;
545             }
546 
547 
548         /*
549            Fill H1[] with bits of InputBytes (hash of message)
550            accordingly to RP1 permutation values and fill again
551           H1 with bits of InputBytes accordingly to RP5 permutation
552         */
553         for (index=0; index<160; index++)
554         {
555                 byteindex = (byte)((RP1[index]>>3)&0x1F);
556                 bitindex = (byte)(0x80 >> (RP1[index]&0x07));
557                 if ((InputBytes[byteindex] & bitindex) != 0) {
558                     byteIndexD = (byte)(index>>3);
559                     bitIndexD = (byte)(0x80 >> (byte)(index&0x0007));
560                     H1[byteIndexD] = (byte)(H1[byteIndexD] | bitIndexD);
561                 }
562 
563 
564             byteindex = (byte)((RP5[index]>>3)&0x1F);
565             bitindex = (byte)(0x80 >> (RP5[index]&0x07));
566             if ((InputBytes[byteindex] & bitindex) !=0) {
567                 byteIndexD = (byte)((index>>3)&0x1FFF);
568                 bitIndexD = (byte)(0x80 >> (byte)(index&0x0007));
569                 HSMul[byteIndexD] ^= bitIndexD;
570             }
571 
572         }
573 
574         for (j=0; j<20; j++)
575             HSMul[j] ^= (byte)(H1[j] ^ H1[(j+4)%20] ^ H1[(j+12)%20] ^ H1↩
                    [(j+16)%20]);
576 
577     }
```

```
578     /******** end of inverseAffineTransformation method *************/
579
580 }
581 /************ end of Applet AppMQQ **********************/
```

Appendix B

(Program Code for RSA Digital Signature

```
1  /**
2   *
3   */
4  package packRSA;
5  // select cardR 6361726452
6  import javacard.framework.Applet;
7  import javacard.framework.ISO7816;
8  import javacard.framework.ISOException;
9  import javacard.framework.APDU;
10 import javacard.framework.JCSystem;
11 import javacard.framework.Util;
12 import javacard.security.RSAPrivateCrtKey;
13 import javacard.security.RSAPublicKey;
14 import javacard.security.RSAPrivateKey;
15 import javacard.security.KeyBuilder;
16 import javacard.security.KeyPair;
17 import javacard.security.RandomData;
18 import javacard.security.Signature;
19
20
21 /**
22  * @author kamran
23  *
24  */
25 public class AppRSA extends Applet {
26
27     private short dataSize=(short)256;
28     private short signSizeRSA=(short)128;
29     private short signParSize=(short)256;
30
31     byte[] signRSA=JCSystem.makeTransientByteArray((short)signSizeRSA,
```

```
32              JCSystem.CLEAR_ON_DESELECT);
33      byte[] signParRSA=JCSystem.makeTransientByteArray((short)signParSize↩
            ,
34              JCSystem.CLEAR_ON_DESELECT);
35      byte[] signParRSA2=JCSystem.makeTransientByteArray((short)↩
            signParSize,
36              JCSystem.CLEAR_ON_DESELECT);
37      byte[] data=JCSystem.makeTransientByteArray(dataSize,
38              JCSystem.CLEAR_ON_DESELECT);
39
40      final static short REG_FAILURE=0x0001;
41
42      private KeyPair keyPairRSA;
43      private RSAPrivateCrtKey RSAPrivateCrt;
44      private RSAPrivateKey RSAPrivate;
45      private RSAPublicKey RSAPublic;
46      private RandomData random;
47
48
49      private Signature signatureRSA;
50
51      /********** Parameters of RSA_CRT Private key ************/
52
53      //P and Q prime co-factors of N
54        static final byte[] P = {
55            (byte)0xF9, (byte)0x7C, (byte)0xEB, (byte)0x9D, (byte)0xA1, (↩
                  byte)0x86,
56            (byte)0x92, (byte)0x6C, (byte)0x70, (byte)0x9F, (byte)0x18, (↩
                  byte)0xA3,
57            (byte)0xA8, (byte)0xD8, (byte)0x25, (byte)0xB6, (byte)0x4B, (↩
                  byte)0xBC,
58            (byte)0xBF, (byte)0x9F, (byte)0x87, (byte)0xF1, (byte)0x5C, (↩
                  byte)0xA0,
59            (byte)0x20, (byte)0xEF, (byte)0x3E, (byte)0x5A, (byte)0x6E, (↩
                  byte)0xF6,
60            (byte)0x64, (byte)0x14, (byte)0xEC, (byte)0xB7, (byte)0xF8, (↩
                  byte)0x08,
61            (byte)0x42, (byte)0xAE, (byte)0x05, (byte)0xE4, (byte)0x3C, (↩
                  byte)0x50,
62            (byte)0x7A, (byte)0x2B, (byte)0x8F, (byte)0x2A, (byte)0xC6, (↩
                  byte)0xD8,
63            (byte)0x88, (byte)0xF9, (byte)0x3A, (byte)0x76, (byte)0x77, (↩
                  byte)0x27,
64            (byte)0x86, (byte)0x10, (byte)0x6C, (byte)0x05, (byte)0x92, (↩
                  byte)0xA3,
65            (byte)0xE2, (byte)0x42, (byte)0xA2, (byte)0x33,
66        };
67
68        static final byte[] Q = {
69            (byte)0xB9, (byte)0x72, (byte)0xE5, (byte)0x09, (byte)0xA5, (↩
                  byte)0x72,
70            (byte)0xB5, (byte)0x9B, (byte)0x79, (byte)0x09, (byte)0x02, (↩
                  byte)0xBE,
71            (byte)0xAB, (byte)0xE3, (byte)0xA6, (byte)0xA6, (byte)0x9D, (↩
                  byte)0x6F,
72            (byte)0x73, (byte)0x05, (byte)0x25, (byte)0x54, (byte)0xC5, (↩
                  byte)0x17,
```

```
73              (byte)0x50, (byte)0x53, (byte)0x71, (byte)0x03, (byte)0x1F, (↩
                    byte)0xBE,
74              (byte)0x27, (byte)0xFF, (byte)0x20, (byte)0xAF, (byte)0x2D, (↩
                    byte)0x27,
75              (byte)0x6C, (byte)0x71, (byte)0x09, (byte)0xCF, (byte)0x0F, (↩
                    byte)0x3E,
76              (byte)0x00, (byte)0xFD, (byte)0xBD, (byte)0xA5, (byte)0xA0, (↩
                    byte)0xC0,
77              (byte)0x65, (byte)0x01, (byte)0xC2, (byte)0x9C, (byte)0xC0, (↩
                    byte)0x77,
78              (byte)0xDF, (byte)0x25, (byte)0x27, (byte)0xDE, (byte)0xD9, (↩
                    byte)0x1B,
79              (byte)0x8F, (byte)0xC6, (byte)0x82, (byte)0x9F,
80
81          };
82
83          // RSA_CRT co-efficient PQ (PQ=1/q mod p)
84          static final byte[] PQ = {
85              (byte)0xB6, (byte)0xB1, (byte)0x23, (byte)0x99, (byte)0xD5, (↩
                    byte)0x12,
86              (byte)0xDA, (byte)0x50, (byte)0x38, (byte)0x2E, (byte)0x44, (↩
                    byte)0xA4,
87              (byte)0x21, (byte)0x94, (byte)0x3B, (byte)0x50, (byte)0x49, (↩
                    byte)0x59,
88              (byte)0x61, (byte)0xF7, (byte)0xF9, (byte)0x29, (byte)0xAA, (↩
                    byte)0xD2,
89              (byte)0x5E, (byte)0x6D, (byte)0x02, (byte)0x55, (byte)0xB4, (↩
                    byte)0x1E,
90              (byte)0x4C, (byte)0xDE, (byte)0x20, (byte)0x2F, (byte)0x59, (↩
                    byte)0xC4,
91              (byte)0x95, (byte)0xD9, (byte)0x42, (byte)0x6B, (byte)0x40, (↩
                    byte)0x21,
92              (byte)0x97, (byte)0x0B, (byte)0xA6, (byte)0xF4, (byte)0x32, (↩
                    byte)0x96,
93              (byte)0x8B, (byte)0x6B, (byte)0xC3, (byte)0xEB, (byte)0x2E, (↩
                    byte)0x26,
94              (byte)0x32, (byte)0x86, (byte)0xBB, (byte)0x91, (byte)0x0F, (↩
                    byte)0x30,
95              (byte)0xA3, (byte)0x69, (byte)0xC6, (byte)0xB1,
96
97          };
98
99          // RSA_CRT exponents DP1 and DQ1 (DP1=d mod (p-1)) (DQ1=d mod (q↩
                -1))
100         static final byte[] DP1 = {
101             (byte)0x1E, (byte)0x7C, (byte)0x2D, (byte)0x2E, (byte)0x2D, (↩
                    byte)0xB6,
102             (byte)0x8B, (byte)0xDD, (byte)0xC4, (byte)0x45, (byte)0x2C, (↩
                    byte)0x75,
103             (byte)0x93, (byte)0x04, (byte)0x16, (byte)0x57, (byte)0x98, (↩
                    byte)0x19,
104             (byte)0x90, (byte)0x30, (byte)0xA6, (byte)0x23, (byte)0xCF, (↩
                    byte)0xF5,
105             (byte)0xA1, (byte)0x10, (byte)0x9A, (byte)0xC5, (byte)0xE2, (↩
                    byte)0x19,
106             (byte)0x29, (byte)0x51, (byte)0x85, (byte)0x3B, (byte)0x55, (↩
                    byte)0x8B,
```

```
107             (byte)0x6C, (byte)0xDA, (byte)0x66, (byte)0xCD, (byte)0xE4, (↩
                    byte)0xB0,
108             (byte)0xD0, (byte)0xBC, (byte)0xD1, (byte)0xD9, (byte)0xA0, (↩
                    byte)0x42,
109             (byte)0x85, (byte)0x3A, (byte)0x2E, (byte)0xF2, (byte)0x9A, (↩
                    byte)0xCC,
110             (byte)0xB1, (byte)0x8D, (byte)0x00, (byte)0x26, (byte)0x0E, (↩
                    byte)0x2D,
111             (byte)0x08, (byte)0x50, (byte)0xAC, (byte)0x11,
112
113         };
114
115         static final byte[] DQ1 = {
116             (byte)0x52, (byte)0x30, (byte)0xDF, (byte)0xDD, (byte)0xF4, (↩
                    byte)0x9B,
117             (byte)0xF0, (byte)0x6D, (byte)0x65, (byte)0xA9, (byte)0x5E, (↩
                    byte)0xB4,
118             (byte)0x0F, (byte)0x0E, (byte)0xA8, (byte)0x6F, (byte)0xB3, (↩
                    byte)0xDB,
119             (byte)0x0F, (byte)0x49, (byte)0x3A, (byte)0x90, (byte)0x65, (↩
                    byte)0x81,
120             (byte)0xBD, (byte)0xB2, (byte)0x1D, (byte)0xA6, (byte)0x5A, (↩
                    byte)0xCD,
121             (byte)0x36, (byte)0x80, (byte)0xD6, (byte)0x85, (byte)0x8D, (↩
                    byte)0x27,
122             (byte)0xA9, (byte)0xE2, (byte)0x37, (byte)0x8C, (byte)0xB3, (↩
                    byte)0x9E,
123             (byte)0xB1, (byte)0x65, (byte)0xC4, (byte)0x45, (byte)0xC2, (↩
                    byte)0x07,
124             (byte)0x43, (byte)0x3D, (byte)0x12, (byte)0x79, (byte)0xD2, (↩
                    byte)0xBB,
125             (byte)0xCE, (byte)0x04, (byte)0x73, (byte)0xB5, (byte)0x4A, (↩
                    byte)0xD7,
126             (byte)0xF2, (byte)0x52, (byte)0xF2, (byte)0xD5,
127
128         };
129
130
131
132         // Exponentiation of Public Key
133         static final byte[] expE = {
134             (byte)0x10, (byte)0x00, (byte)0x01
135         };
136
137         static final byte[] modN = {
138             (byte)0xB4, (byte)0xBB, (byte)0x3F, (byte)0x1B, (byte)0xFB, (↩
                    byte)0x51,
139             (byte)0xC6, (byte)0x3F, (byte)0xFC, (byte)0xDE, (byte)0xDF, (↩
                    byte)0x96,
140             (byte)0x25, (byte)0x5F, (byte)0x5D, (byte)0x10, (byte)0xE1, (↩
                    byte)0xA5,
141             (byte)0xAC, (byte)0x29, (byte)0xC0, (byte)0x5A, (byte)0x37, (↩
                    byte)0x1B,
142             (byte)0x82, (byte)0x38, (byte)0xB5, (byte)0xFC, (byte)0xB9, (↩
                    byte)0x66,
143             (byte)0x79, (byte)0xAF, (byte)0xF5, (byte)0x93, (byte)0x43, (↩
                    byte)0xDC,
```

```
144             (byte)0x32, (byte)0xE7, (byte)0x1C, (byte)0xBA, (byte)0xC1, (↩
                    byte)0xF0,
145             (byte)0xC5, (byte)0xD6, (byte)0x84, (byte)0x2D, (byte)0xA8, (↩
                    byte)0x2C,
146             (byte)0x22, (byte)0x34, (byte)0xD8, (byte)0x44, (byte)0xFD, (↩
                    byte)0x95,
147             (byte)0x72, (byte)0xE2, (byte)0x4E, (byte)0xEF, (byte)0xC0, (↩
                    byte)0xC4,
148             (byte)0xC1, (byte)0xEF, (byte)0x48, (byte)0x5A, (byte)0x51, (↩
                    byte)0x48,
149             (byte)0x60, (byte)0x80, (byte)0x84, (byte)0xD7, (byte)0x00, (↩
                    byte)0xEF,
150             (byte)0xA0, (byte)0x75, (byte)0x50, (byte)0x00, (byte)0xD5, (↩
                    byte)0x80,
151             (byte)0x3A, (byte)0x76, (byte)0x8F, (byte)0x9C, (byte)0xD5, (↩
                    byte)0x14,
152             (byte)0x85, (byte)0xAD, (byte)0x53, (byte)0x9E, (byte)0x59, (↩
                    byte)0x6D,
153             (byte)0xCC, (byte)0x64, (byte)0xA2, (byte)0x86, (byte)0x49, (↩
                    byte)0x8A,
154             (byte)0x23, (byte)0xC2, (byte)0xE1, (byte)0x50, (byte)0xF6, (↩
                    byte)0xAC,
155             (byte)0x2C, (byte)0x95, (byte)0x4E, (byte)0xD8, (byte)0x0F, (↩
                    byte)0xE5,
156             (byte)0xF2, (byte)0x0D, (byte)0xE0, (byte)0x88, (byte)0x87, (↩
                    byte)0x4F,
157             (byte)0xE1, (byte)0x7E, (byte)0x94, (byte)0xC5, (byte)0xE1, (↩
                    byte)0x37,
158             (byte)0xA7, (byte)0xCF, (byte)0xDB, (byte)0x34, (byte)0x4E, (↩
                    byte)0x32,
159             (byte)0xA3, (byte)0xAD,
160             };
161
162
163
164 //        ***********START   CONSTRUCTOR↩
    *******************************///
165         private AppRSA(byte[] bArray,short bOffset,byte bLength){
166
167
168             //Register applet
169                     try {
170                         register();
171                     } catch (Exception e) {
172                         ISOException.throwIt(REG_FAILURE);
173                     }
174
175
176
177                     random = RandomData.getInstance(RandomData.↩
                            ALG_PSEUDO_RANDOM);
178                     signatureRSA = Signature.getInstance(Signature.↩
                            ALG_RSA_SHA_PKCS1,false);
179         }
180 //        ***********END   CONSTRUCTOR*******************************///
181
182
```

```
183     public static void install(byte[] bArray, short bOffset, byte ↩
            bLength) {
184         // GP-compliant JavaCard applet registration
185         new AppRSA(bArray, bOffset, bLength);
186 
187     }
188 
189     public void process(APDU apdu) {
190         // Good practice: Return 9000 on SELECT
191         if (selectingApplet()) {
192             return;
193         }
194 
195         byte[] buf = apdu.getBuffer();
196         // Check CLA field against perticular value 80 in Hex
197         if (buf[ISO7816.OFFSET_CLA]!=(byte)0x80)
198             ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
199 
200         //Check INS field against perticular value 10 in Hex for ↩
                following instruction
201         if (buf[ISO7816.OFFSET_INS]==(byte)0x10)
202         {
203             /*  Calculating Digital signature
204              * using RSA_CRT with SHA1
205              */
206             random.generateData(data,(short)0,dataSize);
207             generate_KeyPair();
208 
209         }else if (buf[ISO7816.OFFSET_INS]==(byte)0x20)
210             {
211 
212             generate_signature();
213 
214             Util.arrayCopy(signRSA,(short)0,buf,(short)0,(short)↩
                    signSizeRSA);
215             apdu.setOutgoingAndSend((short)0,signSizeRSA);
216 
217 
218 
219     } else
220         ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
221     }
222 
223     private void generate_KeyPair(){
224         keyPairRSA = new KeyPair(KeyPair.ALG_RSA_CRT, KeyBuilder.↩
                LENGTH_RSA_1024);
225 
226         RSAPrivateCrt = (RSAPrivateCrtKey) keyPairRSA.getPrivate();
227         RSAPublic = (RSAPublicKey) keyPairRSA.getPublic();
228 
229         RSAPrivateCrt.setP(P, (short)0, (short)P.length);
230         RSAPrivateCrt.setQ(Q, (short)0, (short)Q.length);
231         RSAPrivateCrt.setPQ(PQ, (short)0, (short)PQ.length);
232         RSAPrivateCrt.setDP1(DP1, (short)0, (short)DP1.length);
233         RSAPrivateCrt.setDQ1(DQ1, (short)0, (short)DQ1.length);
234 
235 
```

```
236             keyPairRSA.genKeyPair();
237
238         }
239
240         private void generate_signature(){
241
242             signatureRSA.init(RSAPrivateCrt,Signature.MODE_SIGN);
243             signatureRSA.sign(data,(short)0,dataSize,signRSA,(short)0);
244
245         }
246
247     }
```

Appendix C

Program Code for ECDSA Digital Signature

```
1  /**
2   *
3   */
4  package packEC;
5  // select cardE 6361726445
6  import javacard.framework.Applet;
7  import javacard.framework.ISO7816;
8  import javacard.framework.ISOException;
9  import javacard.framework.APDU;
10 import javacard.framework.JCSystem;
11 import javacard.framework.OwnerPIN;
12 import javacard.framework.PINException;
13 import javacard.framework.Util;
14 import javacard.security.DESKey;
15 import javacard.security.ECPrivateKey;
16 import javacard.security.ECPublicKey;
17 import javacard.security.KeyBuilder;
18 import javacard.security.KeyPair;
19 import javacard.security.RandomData;
20 import javacard.security.Signature;
21 import javacardx.crypto.Cipher;
22
23
24 /**
25  * @author kamran
26  *
27  */
28 public class AppEC extends Applet {
29
30     private short dataSize=(short)256;
31     private short signSizeEC=(short)48;
```

```
32
33      byte[] data=JCSystem.makeTransientByteArray(dataSize,
34              JCSystem.CLEAR_ON_DESELECT);
35      byte[] signEC=JCSystem.makeTransientByteArray((short)signSizeEC,
36              JCSystem.CLEAR_ON_DESELECT);
37
38      final static short REG_FAILURE=0x0001;
39
40
41
42
43      private KeyPair keyPairEC;
44      private ECPrivateKey ECPrivate;
45      private ECPublicKey ECPublic;
46      private Signature signatureEC;
47      private RandomData random;
48
49      /********** Parameters to sign ECDSA Signature **********/
50      // NSA K-163 with Polynomial basis f(x) = x^163 + x^7 + x^6 + x^3 + ↩
            1
51      static final  short e1 = 0x0007;
52      static final  short e2 = 0x0006;
53      static final  short e3 = 0x0003;
54
55      //cofactor
56      static final  short k = 0x0002;
57
58
59
60
61      // Two field elements a and b = 1 define EC equation y^2+xy = x^3+ax↩
            ^2+b
62    static final byte[] a = {
63    (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
64    (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
65    (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
66    (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
67    (byte)0x01};
68
69    static final byte[] b = {
70    (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
71    (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
72    (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
73    (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
74    (byte)0x01};
75
76    //base point G = (Xg,Yg) in Elliptic Curve
77    static final byte[] pointG = {
78    (byte)0x04, (byte)0x02, (byte)0xFE, (byte)0x13, (byte)0xC0,
79    (byte)0x53, (byte)0x7B, (byte)0xBC, (byte)0x11, (byte)0xAC,
80    (byte)0xAA, (byte)0x07, (byte)0xD7, (byte)0x93, (byte)0xDE,
81    (byte)0x4E, (byte)0x6D, (byte)0x5E, (byte)0x5C, (byte)0x94,
82    (byte)0xEE, (byte)0xE8, (byte)0x02, (byte)0x89, (byte)0x07,
83    (byte)0x0F, (byte)0xB0, (byte)0x5D, (byte)0x38, (byte)0xFF,
84    (byte)0x58, (byte)0x32, (byte)0x1F, (byte)0x2E, (byte)0x80,
85    (byte)0x05, (byte)0x36, (byte)0xD5, (byte)0x38, (byte)0xCC,
86    (byte)0xDA, (byte)0xA3, (byte)0xD9};
```

```
87
88    // Prime order of base point G
89    static final byte[] r = {
90    (byte)0x04, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
91    (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
92    (byte)0x02, (byte)0x01, (byte)0x08, (byte)0xA2, (byte)0xE0,
93    (byte)0xCC, (byte)0x0D, (byte)0x99, (byte)0xF8, (byte)0xA5,
94    (byte)0xEF};
95
96      private final static byte[] s = { (byte) 0x04, (byte) 0x0C,
97          (byte) 0x66, (byte) 0xA6, (byte) 0x44, (byte) 0x59, (byte) 0x48,
98          (byte) 0x4B, (byte) 0x85, (byte) 0x65, (byte) 0x7C, (byte) 0x5D,
99          (byte) 0x64, (byte) 0x0A, (byte) 0x24, (byte) 0xED, (byte) 0x3C,
100         (byte) 0x75, (byte) 0x49, (byte) 0x83, (byte) 0x5F };
101
102     private final static byte[] w = { (byte) 0x04, (byte) 0x0C,
103         (byte) 0x66, (byte) 0xA6, (byte) 0x44, (byte) 0x59, (byte) 0x48,
104         (byte) 0x4B, (byte) 0x85, (byte) 0x65, (byte) 0x7C, (byte) 0x5D,
105         (byte) 0x64, (byte) 0x0A, (byte) 0x24, (byte) 0xED, (byte) 0x3C,
106         (byte) 0x75, (byte) 0x49, (byte) 0x83, (byte) 0x5F, (byte) 0x6E,
107         (byte) 0xFB, (byte) 0x5B, (byte) 0x71, (byte) 0x9E, (byte) 0xF7,
108         (byte) 0x6A, (byte) 0x42, (byte) 0xF7, (byte) 0x88, (byte) 0x68,
109         (byte) 0xFF, (byte) 0xA9, (byte) 0x91, (byte) 0x6D, (byte) 0x50,
110         (byte) 0x85, (byte) 0x29, (byte) 0xB9, (byte) 0xE6, (byte) 0xA6,
111         (byte) 0x55 };
112
113
114 //        ***********START  CONSTRUCTOR↩
        ********************************///
115         private AppEC(byte[] bArray,short bOffset,byte bLength){
116
117
118             //Register applet
119                     try {
120                         register();
121                     } catch (Exception e) {
122                         ISOException.throwIt(REG_FAILURE);
123                     }
124
125
126
127             signatureEC = Signature.getInstance(Signature.ALG_ECDSA_SHA,↩
                      false);
128             random = RandomData.getInstance(RandomData.ALG_PSEUDO_RANDOM↩
                  );
129
130         }
131 //        ***********END  CONSTRUCTOR*********************************///
132
133
134
135
136     public static void install(byte[] bArray, short bOffset, byte ↩
        bLength) {
137         // GP-compliant JavaCard applet registration
138         new AppEC(bArray, bOffset, bLength);
139     }
```

```
140
141     public void process(APDU apdu) {
142         // Good practice: Return 9000 on SELECT
143         if (selectingApplet()) {
144             return;
145         }
146
147         byte[] buf = apdu.getBuffer();
148
149         // Check CLA field against perticular value 80 in Hex
150         if (buf[ISO7816.OFFSET_CLA]!=(byte)0x80)
151             ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
152
153         //Check INS field against perticular value 10 in Hex for ↩
                following instruction
154         if (buf[ISO7816.OFFSET_INS]==(byte)0x10)
155         {
156             /******  Digital Signature Implementation using
157              * Elliptic Curve ECDSA and Hash function SHA1
158              ********/
159
160             random.generateData(data,(short)0,dataSize);
161
162             generate_KeyPair();
163
164
165
166         }else if (buf[ISO7816.OFFSET_INS]==(byte)0x20)
167             {
168
169             generate_signature();
170
171             Util.arrayCopy(signEC,(short)0,buf,(short)0,(short)↩
                    signSizeEC);
172             apdu.setOutgoingAndSend((short)0,signSizeEC);
173
174
175
176         } else
177             ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
178
179     }
180
181     private void generate_KeyPair(){
182         keyPairEC = new KeyPair(KeyPair.ALG_EC_F2M, KeyBuilder.↩
                LENGTH_EC_F2M_163);
183
184         ECPublic = (ECPublicKey)keyPairEC.getPublic();
185         ECPrivate = (ECPrivateKey)keyPairEC.getPrivate();
186
187         ECPublic.setFieldF2M(e1, e2, e3);
188         ECPublic.setA(a, (short)0, (short)21);
189         ECPublic.setB(b,(short)0,(short)21);
190         ECPublic.setG(pointG,(short)0,(short)43);
191         ECPublic.setK(k);
192         ECPublic.setR(r,(short)0,(short)21);
193         ECPublic.setW(w,(short)0,(short)w.length);
```

```
194
195         ECPrivate.setFieldF2M(e1, e2, e3);
196         ECPrivate.setA(a, (short)0, (short)21);
197         ECPrivate.setB(b,(short)0,(short)21);
198         ECPrivate.setG(pointG,(short)0,(short)43);
199         ECPrivate.setK(k);
200         ECPrivate.setR(r,(short)0,(short)21);
201         ECPrivate.setS(s,(short)0,(short)s.length);
202
203         keyPairEC.genKeyPair();
204
205     }
206
207     private void generate_signature(){
208
209         signatureEC.init(ECPrivate,Signature.MODE_SIGN);
210         signatureEC.sign(data,(short)0,dataSize,signEC,(short)0);
211     }
212
213
214 }
```