UNIVERSIDADE TÉCNICA DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Departamento de Engenharia Electrotécnica e de Computadores

Comparative analysis of authentication schemes on a Java Card smart card

José Rafael Trigueiro de Carvalho

Dissertation submitted for obtaining the degree of Master in
Electrical and Computer Engineering

Jury

President:	Prof Nuno Cavaco Gomes Horta
Supervisor:	Prof Rui Gustavo Nunes Pereira Crespo
Members:	Prof Henrique João Domingos

October 2011

Acknowledgements

I especially would like to thank Professor Rui Gustavo Crespo, who made it all possible. Although I cannot really put it into words, I know that without his guidance, availability, and constant support, these past few months would not have turned out to be such a wonderful experience.

To my family, who I can never thank enough for their continuous support, patience and for allowing me certain indulgences. You were always there for me.

To my friends, who sometimes understand me better than myself, thanks for the ”escapes” that I needed to clear my mind.

To Julie and all my colleagues at the Cambridge institute, for all their friendship and constant flow of motivation. Even though you were completely clueless about my work, you were the best!

To grampa...

Pelo sonho é que vamos

Pelo sonho é que vamos,
comovidos e mudos.

Chegamos? Não chegamos?
Haja ou não haja frutos,
pelo sonho é que vamos.

Basta a fé no que temos.
Basta a esperança naquilo
que talvez não teremos.
Basta que a alma demos,
com a mesma alegria,
ao que desconhecemos
e ao que é do dia a dia.

Chegamos? Não chegamos?
- Partimos. Vamos. Somos.

Sebastião da Gama

Resumo

Esta tese tem como objectivo comparar o desempenho de métodos de autenticação, implementados num cartão inteligente Java Card. Os métodos de autenticação considerados dividem-se em dois grupos: desafio/resposta e protocolos de conhecimento nulo.

A autenticação por desafio/resposta baseada na cifra AES é mais rápida e segura do que a baseada na cifra RSA. Esta última apenas é possível com auxílio do coprocessador. Devido à complexidade das operações de multiplicação e exponenciação modular, o mesmo se aplica à autenticação baseada em protocolos de conhecimento nulo, nomeadamente o Feige-Fiat-Shamir e Guillou-Quisquater. A máquina virtual não permite executar tarefas computationalmente exigentes em tempo útil, e além do mais, o acesso ao coprocessador criptográfico encontra-se limitado pelo API do Java Card.

Também propomos um método de autenticação por desafio/resposta, baseado numa cifra de fluxo caótico de chaves chamada eLoBa. Para uma implementação sem auxílio do coprocessador, o tempo de resposta varia entre 1.4 segundos para 16 rondas e 2.5 segundos para 31 rondas. A autenticação baseada na cifra eLoBa demora cerca de 10 vezes mais do que a baseada na cifra AES, que é o criptosistema mais rapido analisado. O limite máximo de 2.5 segundos é aceitável para um utilizador comum, e como tal, a autenticação baseada na cifra eLoBa é adequada para cartões inteligentes contemporâneos.

Palavras chave: Autenticação · Java Card · Criptografia Caótica · Desafio/Resposta · Provas de conhecimento nulo · Desempenho

Abstract

In this thesis we report on the performance of authentication schemes in a Java Card smart card. Authentication schemes have been divided into two sorts, challenge-response and zero-knowledge protocols.

Challenge-response authentication based on AES is faster and more secure than RSA, which is only feasible with coprocessor support. Due to the complexity of modular multiplication and modular exponentiation, the same applies to authentication based on zero-knowledge protocols, namely Feige-Fiat-Shamir and Guillou-Quisquater. Demanding computations on the virtual machine are too slow for practical use, but on the other hand, access to the cryptographic coprocessor is limited by the restrictive Java Card API.

We also present a challenge-response authentication scheme based on a stream-based chaotic cipher named eLoBa. The response time, for Java-based implementation, varies between 1.4sec for 16 rounds, and 2.5sec for 31 rounds. The eLoBa-based authentication scheme is about 10 times slower than the AES-based scheme, which is the fastest cryptosystem analysed. The maximum authentication time of 2.5 seconds is acceptable for most end-users, therefore, eLoBa-based authentication is suitable for contemporary smart cards.

Keywords: Authentication · Java Card · Chaos cryptography · Challenge-response · Zero-knowledge · Performance

Contents

Resumo v

Abstract vii

List of Figures xiii

List of Tables xv

List of Abbreviations xvii

1 Introduction 1

1.1 Motivation 1

1.2 Thesis goals 2

1.3 Dissertation structure 3

2 Authentication Methods 5

2.1 Cryptography - Basic definitions 5

2.2 Authentication - Basic definitions 8

2.3 Notations 10

2.4 Challenge-Response techniques 10

2.4.1 Time-variant parameters 11

2.4.2 Challenge-response by symmetric-key cryptosystems 11

2.4.3 Challenge-response by public-key cryptosystems 12

2.4.4 Challenge-response by stream-based chaotic system 13

2.5 Zero-Knowledge techniques 14

2.5.1 Zero Knowledge Introduction 14

2.5.2 Feige-Fiat-Shamir authentication protocol 16

2.5.3 Guillou-Quisquater authentication protocol 17

3 Smart Card Architecture and Programming 19

3.1 History 20

3.2 Benefits and applications 20

3.3 Smart card basics 22

3.3.1 Smart card types 22

3.3.2 Smart card hardware 22

3.3.3 Communication models 24

3.3.4 Standards and Specifications 26

3.4 Chip Operating Systems 27

3.4.1 Java Card 28

3.4.2 MultOS 30

3.4.3 BasicCard 31

3.4.4 Smartcard.Net 32

3.5 Choice of smart card platform 32

4 Implementing Big Number Operations 35

4.1 Large Number addition and subtraction 36

4.1.1 Addition and subtraction 36

4.1.2 Modular Addition and Subtraction 36

4.2 Large Number Multiplication 38

4.2.1 Large Integer Multiplication 39

4.2.2 Russian Multiplication 39

4.2.3 Modular multiplication 40

4.3 Large Number Exponentiation 45

4.3.1 Montgomery exponentiation 46

4.3.2 Chinese Remainder Theorem 47

4.3.3 Java Card's RSA and exponentiation 48

5 Implementing Authentication Protocols 51

5.1 Block cipher AES 52

5.1.1 Architecture 52

5.1.2 Challenge-Response with AES 55

5.2 Stream cipher eLoBa 55

5.2.1 Architecture 56

5.2.2 Challenge-response with eLoBa 61

5.3 RSA 61

5.3.1 Arquitecture 61

5.3.2 Challenge-response with RSA 63

5.4 Zero-knowledge protocols 64

5.4.1 Feige-Fiat-Shamir 64

5.4.2 Guillou-Quisquater 65

6 Result Analysis, Conclusions and Further Work 69

6.1 The development environment 70

6.2 Performance evaluation 71

6.2.1 Execution time isolation 72

6.2.2 Test module configuration 74

6.3 Performance results . . . 75
6.3.1 Communication overhead . . . 75
6.3.2 Modular Multiplication . . . 76
6.3.3 Performance of Authentication Protocols . . . 76
6.3.3.1 Authentication based on AES . . . 76
6.3.3.2 Authentication based on RSA . . . 77
6.3.3.3 Authentication based on eLoBa . . . 77
6.3.3.4 Authentication based on ZKP . . . 77
6.3.3.5 Overall comparison . . . 78
6.4 Conclusions . . . 79
6.5 Further Work . . . 80

Bibliography **81**

A RSA exponentiation **91**

B AES tables **93**
B.1 AES S-Boxes . . . 93
B.2 AES multiplication tables . . . 94

C Modular multiplication in eLoBa **99**
C.1 1 Byte by 16 Bytes modular multiplication . . . 99
C.2 16 Bytes by 16 Bytes modular multiplication . . . 100

List of Figures

2.1 Encryption and decryption . . . 6
2.2 Authentication based on symmetric-key cryptosystem . . . 12
2.3 Authentication through private key decryption . . . 12
2.4 eLoBa Authentication . . . 14
2.5 Ali Baba's Cave . . . 15
2.6 Feige-Fiat-Shamir protocol . . . 17
2.7 Guillou-Quisquater authentication protocol . . . 18

3.1 smart cards . . . 21
3.2 Card Types [65] . . . 22
3.3 Smart card contact points and architecture . . . 23
3.4 Smart card communication model . . . 24
3.5 Smart card state machine . . . 25
3.6 Java Card system architecture . . . 30
3.7 MultOS architecture . . . 31

5.1 AES round methods . . . 53
5.2 eLoBa architecture . . . 56
5.3 SDL description of eLoBa functionality . . . 62
5.4 RSA-based challenge-response authentication . . . 63

5.5 Feige-Fiat-Shamir authentication scheme 65

5.6 Guillou-Quisquater authentication protocol 67

6.1 Eclipse integrated Java Card development Environment 71

6.2 Communication path between the application on the computer and the card . . 73

6.3 Comparison of authentication protocols 78

C.1 1 Byte by 16 Bytes modular multiplication 99

C.2 16 Bytes by 16 Bytes modular multiplication 100

List of Tables

3.1 APDU structure . . . 25

3.2 sample ATRs . . . 26

3.3 Supported and unsupported Java features . . . 29

3.4 Overview of smart card platforms . . . 33

4.1 Multiplication by squaring . . . 44

4.2 comparison of the number of operations needed to perform modular multiplication 44

4.3 Complexity of reduction algorithms in reducing a $2k$-digit number . . . 45

4.4 RSA public key limitations . . . 50

5.1 Rcon array . . . 55

5.2 Operations required by the FFS protocol . . . 65

5.3 Operations required by the GQ protocol . . . 66

6.1 Time spent for a data APDU . . . 75

6.2 Execution times for modular multiplication, with overheads excluded . . . 76

6.3 Execution times for AES unilateral authentication . . . 77

6.4 Execution times for RSA . . . 77

6.5 ZKP execution times . . . 78

B.1 AES forward S-box . . . 93

B.2 AES inverted S-box 94

B.3 AES m2 table 94

B.4 AES m3 table 95

B.5 AES m9 table 95

B.6 AES mB table 96

B.7 AES mD table 96

B.8 AES mE table 97

List of Abbreviations

AES: advanced encryption standard

APDU : application protocol data unit

API : application programming interface

ASCII : American Standard Code for Information Interchange

ATR : answer to reset

CAD : card acceptance device

CAP : converted applet

C-APDU : command application protocol data unit

CLK : clock

COS : chip operating system

CPU : central processing unit

CRT : Chinese remainder theorem

DH : Diffie-Helman

DSA : digital signature algorithm

DSS : digital signature standard

ECC : elliptic curve cryptography

ECDSA : elliptic curve digital signature algorithm

EEPROM: electrical erasable program read-only memory

eID : electronic identification

EMV : Europay MasterCard Visa

GND : ground (electrical)

GPL : GNU General Public License

I/O : input/output

ICC : integrated chip card

ISO : International Organization for Standardization

JC : Java Card

JCRE : Java Card runtime environment

JCVM : Java Card virtual machine

LSB : least significant bit

MD5 : Message Digest Algorithm 5

MEL : Multos Executable Language

MSB : most significant bit

MUSCLE : movement for the use of smart card in a Linux environment

OS : operative system

OTP: one-time pad

PC/SC : Personal Computer/Smart Card

PIN: personal identification number

RAM : random access memory

R-APDU : response application protocol data unit

RFU : reserved for future use

ROM : read-only memory

RSA : Rivest, Shamir and Adleman cryptographic algorithm

RST : Reset

SHA-1 : Secure Hash Algorithm 1

SIM : subscriber identity module

Vcc : supply voltage

VM : virtual machine

XOR : logical exclusive OR operation

WfSC : Windows for Smart Cards

Chapter 1

Introduction

> Everyone suspects himself of at least one of the cardinal virtues, and this is mine: I am one of the few honest people that I have ever known.
>
> The Great Gatsby
> F. SCOTT FITZGERALD

1.1 Motivation

With the decreasing cost of smart cards, as well as their growing computational power and storage capacity, a new door has been opened to a whole new variety of security mechanisms, making them ideal candidates for authentication devices. A smart card owner who wants to access a given resource can, therefore, show a verifier he is who he claims to be, in such a way that nobody else can pose as himself.

Cards have gone a long way to meet the users' demands. At the very beginning they were mere PVC embossed cards, used for exclusive membership. However, the growing need for more sophisticated identification mechanisms led to the creation of magnetic-stripe credit cards and, ultimately, to the development of smart cards. The latter resemble tiny computers as they not only provide portability, ease of use and data storage but also processing ability. Currently, smart cards are replacing magnetic-stripe cards as a more secure alternative; they not only provide far greater storage capacity than that of a magnetic-stripe card, but their stored data can also be protected against unauthorised access and manipulation.

The current generation of smart cards is both convenient and efficient. Their security properties, the multi-application smart card operating systems and the standardization of smart

card are expanding the range of potential applications. For instance, currently deployed smart cards are widely used in public transportation, e.g. Lisboa Viva — see http://www.carris.pt/en/Lisboa-viva and Andante — see http://en.wikipedia.org/wiki/Andante_ticket. Payment cards also provide authentication and digital signature functionalities to citizens, e.g. portuguese identity card named Cartão Cidadão and developed by Gemalto — see: http://www.cartaodecidadao.pt, allowing them to access a variety of services.

Advanced authentication schemes are possible thanks to the smart cards' ability to execute applications and perform cryptographic calculations like hashing, encrypting and decrypting. However, in order to develop such applications we must not only adhere to international standards to ensure interoperability, but also understand the capabilities and limitations of current technologies. Despite the advances in smart card technology, they still remain resource-constrained devices.

1.2 Thesis goals

In this thesis we set out to evaluate the performance of different authentication mechanisms on a Java Card [14, 6] smart card. The objective is to achieve authentication within a 5 second period and to conclude which protocols and key lengths are useful in practice. For that purpose, a Java Card smart card is used as a trusted platform module and challenge-response as well as zero-knowledge authentication schemes are considered.

Java Card devices run a subset of the Java language, tailored to suit resource-constrained devices. However, aside from the advantages of code portability and multi-application support, the use of an interpreted language does not come without a performance penalty. Therefore, we focus on it's suitability to implement already established authentication schemes, as well as developing new ones.

Since smart cards often incorporate a coprocessor to speed up mathematical operations, cryptographic primitives (*e.g.* symmetric cryptosystem AES and public-key cryptosystem RSA) are implemented from scratch, to enable a comparison between the performance of pure-java implementations and the optimized and coprocessor-enabled cryptographic library available on Java Card smart cards.

The thesis contribution is the implementation on a Java smart card of a authentication

scheme [15], based on recently proposed chaotic keystream cipher eLoba [76, 77]. Its performance is also compared against other known authentication schemes.

1.3 Dissertation structure

In this chapter we provide a brief work motivation and the thesis main goals. The remainder of this dissertation is organized as follows.

In Chapter 2 we give a brief introduction to cryptography and present two methods for entity authentication: challenge-response and zero-knowledge protocols.

Chapter 3 covers the main aspects of smart cards and the Java Card technology, namely communication mechanisms between smart cards, smart card operating systems and the programming environment.

In Chapter 4 we describe the modular arithmetic operations necessary to implement the protocols described in chapter 2. Algorithms for multi-precision addition, subtraction, multiplication and exponentiation are presented.

Subsequently, in chapter 5, the focus is put into the challenges and their respective solutions, regarding an implementation on a Java Card platform. We describe the implementation and optimization of the authentication schemes, and how the Java Card cryptographic library can be used to speed up modular multiplication.

The configuration of the development environment is discussed in chapter 6. Further, performance measurement tests and experimental results are presented and discussed. To conclude, we present our conclusions as well as suggestions for further work.

Chapter 2

Authentication Methods

On the Internet, nobody knows you're a dog.

PETER STEINER

In the following sections we will look at different authentication schemes which are built upon symmetric and public-key cryptosystems. The reader is not required to fully grasp the internal behaviour of the cryptographic functions used, nevertheless, it is important to get familiarized with the computational costs and implementation challenges involved. We will cover the essential background; readers who want a deeper understanding of cryptographic theories, have at their disposal several good books devoted to cryptography, such as *Cryptography and Network Security*by William Stallings [79].

2.1 Cryptography - Basic definitions

Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication [5]. Cryptography is the building block of many security services [88]. However, by itself

it does not guarantee security [89], other issues, such as protocol analysis — see Needham-Schroeder, Lowe attack [48] are required. In addition, an attacker may perform side channel attacks, where information is gained from the (physical) implementation of a cryptosystem, rather than by cryptanalysis. Nevertheless, we will assume the existence of a physically secure channel between the communication participants, as we are only concerned with the security and performance aspects of the authentication protocols.

Cryptography manipulates many kinds of text, such as letters, computer data and pictures. We may convert all of them to integers: for example, latin letters may be replaced by their ASCII codes.

Cipher is a cryptographic algorithm that works in combination with a key and a message. The original message, which can be understood without any special measures, is called plaintext, while the disguised message is called ciphertext. The process of transforming a plaintext into a ciphertext, using a cipher, is called encryption (*encipherment*). Thereafter, the plaintext can be recovered from the ciphertext by the inverse process, decryption (*decipherment*). The encryption and decryption process is illustrated in Figure 2.1.

One of the major goals of cryptographic systems is the data privacy of plaintext , *i.e.* to prevent unauthorised parties (intruders) from listening in on private communications. Moreover, ciphertext may be accessed by anyone - communication parties and intruders, because ciphertext circulate on open access media such as Internet.

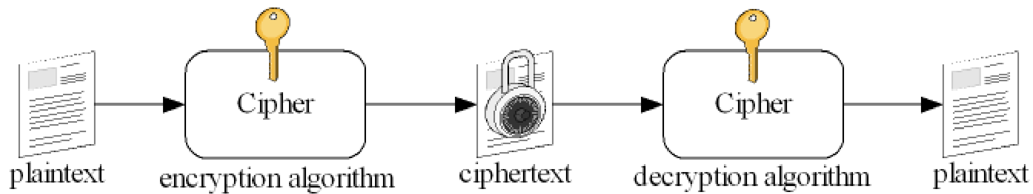


Figure 2.1: Encryption and decryption

The exact transformations performed by a cipher depend both on the type of key used and the type of input data. Symmetric key algorithms (*Private-key cryptography*) use the same key (*secret key*) for encryption and decryption, shared by the two communicating parties, whereas asymmetric key algorithms (*Public-key cryptography*) use two different keys, one which is only know to each individual (*private key*) and another which is publicly available (*public key*).

Symmetric key algorithms are further divided into two categories: block ciphers and stream

ciphers. ***Block ciphers*** split the message into blocks of fixed length; if plaintext size is not a multiple of a block size, then it must be extended - this operation is known as padding. Each block is iterated a given number times, where permutation and substitution operations take place. In addition, the original key is expanded so that a different key is used for each round. One of the most well-known block ciphers is the Advanced Encryption Standard (AES) [25], which was adopted as a standard on 2001 in order to replace the Data Encryption Standard (DES) [16], now considered to be insecure.

Unlike block ciphers, ***stream ciphers*** operate on streams of bits. Instead of a set of fixed transformations, a keystream generator is used to produce a stream of bits which is to be XORed with the stream of input bits, thereby producing the stream of output bits. The output of the keystream generator is a function of the internal state of the keystream generator and hence encryption(decryption) depends not only on the key and plaintext(ciphertext), but also on the internal state of the keystream generator. Therefore, the closer the keystream generator's output is to randomness (when in fact it is actually deterministic since it depends on the key), the harder it will be to break the system's security. Cryptosystems of this type include the Linear Feedback Shift Register [47], RC4 [73] and the chaotic stream cipher, eLoBa [77]. The reality of a stream cipher security lies somewhere between the simple XOR and the one-time pad (OTP) [70], which uses random keys that are as long as the message. The OTP is completely unbreakable since it guarantees that after an opponent receives the ciphertext he has no more information than before receiving the ciphertext. However, the fact that keys cannot be reused creates severe key management problems and prevents keystream generators from being widely used [83].

Key distribution is the biggest disadvantage of symmetric algorithms and it is why asymmetric ciphers are often used to circumvent this issue. In a ***public key cryptosystem***, each user has a pair of keys - the public and the private key. Public keys are not secret and can be broadcast freely, which allows Alice to send Bob a message by ciphering it with his public key. Bob can later decipher the message with his private key. In the same manner, anyone can verify a signature because only the owner of the secret can sign it, and all other parties can check the signature with the owner public key.
The security of these cryptosystems lie in the size of the keys and in the difficulty of factoring large numbers (RSA), or difficulty on discrete logarithm (ElGamal , Diffie-Hellman , DSA, ECC). Furthermore, the private key cannot be calculated from the public key (at least not

computable in polynomial time). However, as the keys increase in length, so do the computational costs involved for ciphering/deciphering, specially for resource-constrained devices such as 8-bit smart card processors. That is why for asymmetric ciphers, such as RSA [69], special math coprocessors are needed. As the minimum key size for RSA keeps increasing, alternatives have already been proposed such as the Elliptic Curve Cryptography (ECC) [54, 46], which when compared to the RSA cryptosystem, requires smaller key sizes for the same level of security.

2.2 Authentication - Basic definitions

This chapter considers different techniques that allow a *claimant* to show a *verifier* that he is who he claims to be; in other words, an authentication scheme allows someone's identity to be confirmed and prevent impersonation.

Very often, the terms Identification and Authentication are used interchangeably, however, while the former refers to the process of establishing an identity, the latter refers to a process of linking this identity to someone. Identification involves a claim or statement of identity : ”I am José Rafael”, while Authentication is a verification of that claim.

Definition 1 *Identification is the means by which an user provides a claimed identity to the system. Authentication is the means of establishing the validity of this claim [58].*

Entity authentication is not to be mistaken with message authentication (MAC), which allows to detect if any changes were made to the message content.
The objectives of identification protocols have been listed as [5]:

1. In the case of honest parties A and B, A is able to successfully authenticate itself to B, i.e., B will complete the protocol having accepted A's identity.

2. (*transferability*) B cannot reuse an identity exchanged with A so as to successfully impersonate A to a third party C.

3. (*impersonation*) The probability is negligible that any party C distinct from A, carrying out the protocol and playing the role of A, can cause B to complete and accept A's identity. Here negligible means ”is so small that it is not of practical significance”.

4. Points 1 to 3 remain true even if a (polynomially) large number of previous authentications between A and B have been observed; the adversary C has participated in previous protocol executions with either or both A and B; and multiple instances of the protocol, possible initiated by C, may be run simultaneously. The idea of zero-knowledge-based protocols, described in section 2.5, is that protocol executions do not even reveal any partial information which makes C's tasks any easier whatsoever.

To provide a proof of identity, authentication can be based on several different factors, which can be used alone or combined. Some commonly used factors are:

1. Something *known* : something which is **known** to the individual. This secret information can be, for example, a password, a personal identification number (PIN), or a cryptographic key.

2. Something *possessed* : something that the individual **owns**. Tokens such as magnetic-strip cards or smart cards are commonly used and will be discussed in the next chapter of this report.

3. Something *inherited* : something that the individual **is**, which usually refers to biometric data (e.g., handwritten signatures, fingerprints, retinal patterns, voice, ...).

There are several characteristics of authentication protocols that must be addressed, such as:

1. *Reciprocity* : either unilateral or mutual authentication is possible, provided that only one, or both entities provide a proof of identity, respectively.

2. *Computational efficiency* : the number of operations required to execute a protocol.

3. *Communication efficiency* : this includes the number messages exchanged between entities, as well as the bandwidth required(total number of bits transmitted).

While passwords provide authentication schemes, such as fixed password schemes and one-time password schemes, they are still vulnerable to a variety of threats such as replay attacks and dictionary attacks. Therefore, we focus on the cryptographic mechanisms available in smart cards which allow us to design stronger authentication schemes.

Several authentication schemes have been proposed and discussed, however our focus is on finding those which are suitable for smart cards. Being resource-constrained devices, the

amount of computation and the memory requirements should be kept as small as possible. In the following sections we will discuss several authentication schemes, based on challenge-response cryptosystems (section 2.4) and Zero-Knowledge protocols (section 2.5).

2.3 Notations

The following notation will be used throughout the dissertation:

- A and B identify, respectively, the verifier and the claimant, and are used to prevent reflection attacks;
- r_A denotes a random number generated by A;
- E_K a symmetric encryption algorithm (*e.g.* AES), with a secret key K shared by entities A and B;
- Optional message fields are denoted by an asterisk (*), while a comma (,) within the scope of E_K denotes concatenation;
- *h(x)* denotes a one-way hash function, where x is the input to the function.
- P_A denotes the public-key of A, required to an asymmetric cryptographic algorithm (*e.g.*, RSA).

2.4 Challenge-Response techniques

In password authentication, the claimant proves her identity by demonstrating that she knows a secret, the password. However, revealing the secret makes it susceptible to interception by the adversary. Replaced by a time-varying challenge, the secret no longer needs to be sent to the verifier; instead, in challenge-response authentication, the claimant demonstrates knowledge of a secret by correctly responding to the challenge, where the response is a function of the entity's secret and the challenge. Since every challenge is different, even if an adversary is monitoring the communications, the response from one execution of the authentication protocol should not provide an adversary with useful information for a subsequent authentication, as subsequent challenges will differ, thereby precluding replay attacks.

2.4.1 Time-variant parameters

Challenges must incorporate time-variant parameters. Such parameters are usually called nonces and are essential to provide uniqueness and distinguish one protocol instance from another. Without time-variant parameters, protocols are vulnerable to counteract replay, interleaving attacks as well as chosen-text attacks.

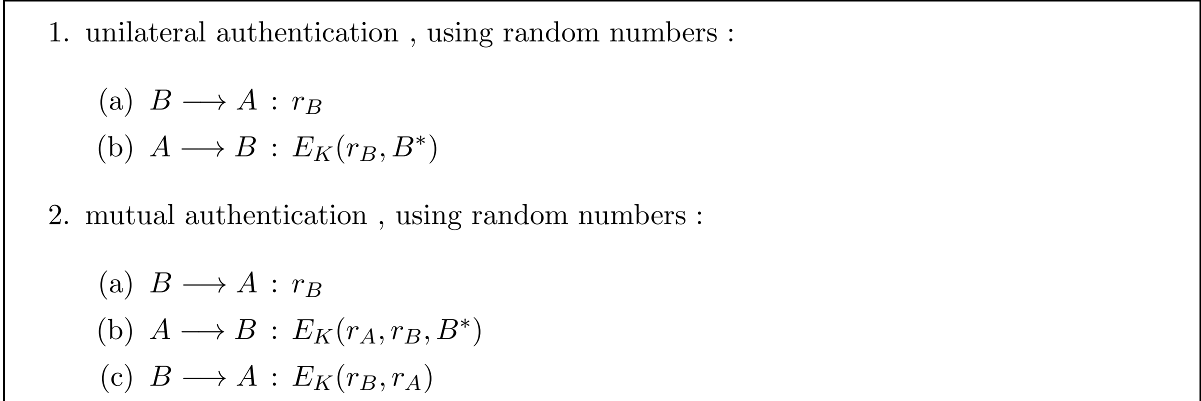
Definition 2 *A nonce is a time variant parameter, to be used no more than once, with the purpose of distinguishing one protocol instance from another. It typically serves to prevent (undetectable) replay.*

There are three main classes of time-variant parameters that can be used: random numbers, sequence numbers, or timestamps. Both have their advantages and disadvantages [5]; however, since the majority of smart cards lack internal time source [65, 66] (*e.g.,* real-time clock) they are, therefore, not adequate for timestamp-based protocols (it is not always possible to extend the use of timestamps to any schema, especially when dealing with smart card authentication processes) or anything which involves time synchronization. They miss standardized access to a timer and what is more, they lack as well appropriate data types to process time [80, 87]. Since timestamps in protocols may typically be replaced by a random number challenge plus a return message, in the further discussed protocols we will solely be using the latter.

2.4.2 Challenge-response by symmetric-key cryptosystems

Challenge-response mechanisms, based on symmetric-key cryptosystems, require the claimant and the verifier to share a symmetric key, which can be derived from the card data (*e.g.*, Chip serial number).

Two simple techniques based on ISO/IEC 9798-2 [40] (Mechanisms using symmetric encipherment algorithms) are described on Figure 2.2, which assume the prior existence of a shared secret key. Two parties may either carry out unilateral entity authentication or mutual authentication. The claimant corroborates its identity by demonstrating knowledge of the shared secret by encrypting a challenge using the shared secret key E_K. The challenge-response procedure is as follows : B generates r_B which he sends to A (step a). Upon reception of the random number, A will either proceed with unilateral authentication or mutual authentication. In the former, he encrypts the received number using its secret key, K, while in the latter



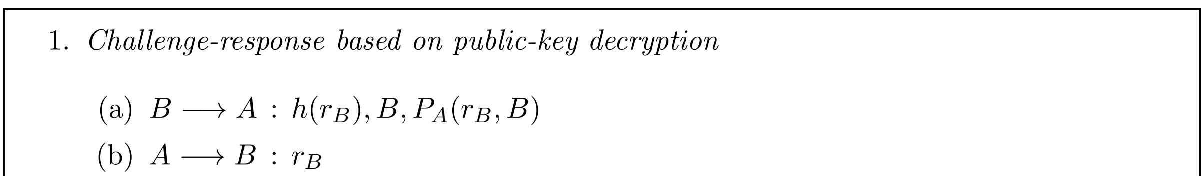
1. unilateral authentication , using random numbers :
   (a) $B \longrightarrow A : r_B$
   (b) $A \longrightarrow B : E_K(r_B, B^*)$
2. mutual authentication , using random numbers :
   (a) $B \longrightarrow A : r_B$
   (b) $A \longrightarrow B : E_K(r_A, r_B, B^*)$
   (c) $B \longrightarrow A : E_K(r_B, r_A)$

Figure 2.2: Authentication based on symmetric-key cryptosystem

he generates r_A to which he appends r_B and encrypts the resulting number using K. After receiving the ciphertext from A (step b), B deciphers it and compares the results against the previously generated r_B. If they match, A will be authenticated by B. Before authenticating A, B may also check the identifier to prevent a reflection attack. In the case of a mutual authentication, upon reception of the ciphertext (b), B recovers r_A, which he swaps with r_B thus concatenating them; thereafter he encrypts the resulting number using K; this step allows the challenge and response to be distinguished from each other. Finally, he sends the resulting ciphertext to A, who will authenticate B if the results match.

2.4.3 Challenge-response by public-key cryptosystems

Public-key cryptosystems may be used for challenge-response based authentication, with a claimant demonstrating knowledge of its private key in one of two ways: the claimant either decrypts a challenge encrypted under its public key or digitally signs a challenge. The former is described in Figure 2.3 [5].



1. *Challenge-response based on public-key decryption*
   (a) $B \longrightarrow A : h(r_B), B, P_A(r_B, B)$
   (b) $A \longrightarrow B : r_B$

Figure 2.3: Authentication through private key decryption

B chooses r_B, computes the witness $x = h(r_B)$ (x demonstrates knowledge of r_B without disclosing it), and computes the challenge $e = P_A(r, B)$. B sends (a) to A. After A decrypts the ciphertext and recovers r'_B and B' , he computes $x' = h(r'_B)$, and quits if $x \neq x'$ or if B is

not equal to its own identifier. Otherwise, A sends $r_B = r'_B$ to B. B succeeds with (unilateral) entity authentication of A upon verifying the received r_B agrees with that sent earlier. The use of the witness precludes chosen-text attacks.

2.4.4 Challenge-response by stream-based chaotic system

Keystream generators are useful for cryptography and may as well be used in authentication schemes. Ciphers, such as eLoBa - "enhanced Lorenz based" [76, 77], which uses a chaotic module as a keystream generator is a good candidate for an authentication scheme [15].

Going over the architecture of the eLoBa cipher falls out of scope of this report, as for the purpose of this section it suffices to explain how it works as an authentication device; three main modules constitute the eLoBa architecture : the chaotic subsystem implements the Lorenz System of equations; the chaotic disturbance subsystem is responsible for the introduction of changes in the chaotic subsystem, therefore avoiding convergence to short-cycle length orbits; finally, the key-mix subsystem translates the internal state of the chaotic system into two 128-bit keys.

The authentication scheme based on the eLoBa cipher is depicted on Figure 2.4, and can be informally explained as follows : A will prove its identity to B via knowledge of a secret 128 bit number, the *seed*. In order to do that, B will produce a 128 bit number that will serve as a challenge. Upon reception of the challenge, the 128 bits are split as shown next :

- 123 bits are XORed with the chaotic system seed, producing the *Key*.
- 4 bit are XORed with 0x10 and will define the number of rounds (*i.e.*, iterations to the chaotic system). This ensures a minimum of 16 rounds and a maximum of 31.
- 1 bit is used to choose the output flux, since each iteration to the chaotic system produces two keys.

The challenge r requires that A is able to answer to the challenge, which demonstrates her knowledge of the secret *seed*. An adversary impersonating A might try to cheat by carefully selecting r, such as $r = 0$, and then collect A's answer to the challenge. That would, however, fail to provide the value of the *seed*, as the scheme ensures that the system is iterated at least 16 times. Moreover, the cipher mechanism encompasses two mechanisms that mask the

internal state of the system : the chaotic system undergoes an initialization process that aims at protecting the seed from algebraic deduction attacks; also, the keys produced by the system do not expose the entire state of the Lorenz system equations.

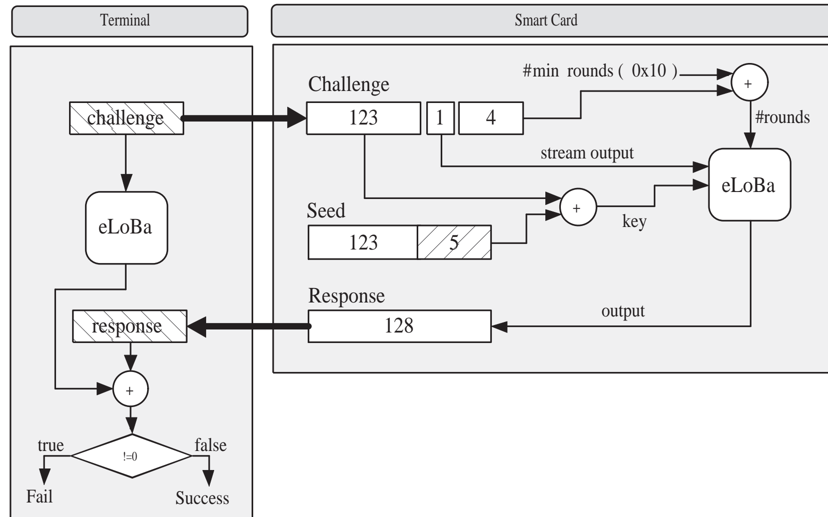


Figure 2.4: eLoBa Authentication

2.5 Zero-Knowledge techniques

In zero-knowledge interactive proofs the claimant only needs to demonstrate the knowledge of the secret, and not anything else that might reveal or endanger the confidentiality of the secret. *S. Goldwasser, S. Micali and C. Rackoff* introduced the concept [31] and the first practical solution was proposed by *A. Fiat and A. Shamir* [27]. There are several well-known zero-knowledge proof authentication schemes. The Feige-Fiat-Shamir(FFS) [26] is based on the difficulty of factoring. The Guillou-Quisquater(GQ) [34] improves FFS protocol in terms of memory requirements and number of rounds required. In the following years, the Schnorr [72, 71] and Okamoto [59] schemes were proposed, whose security is based on the intractability of certain discrete logarithm problems.

2.5.1 Zero Knowledge Introduction

An interactive proof is said to be a proof of knowledge if it has both the properties of completeness and soundness [5].

Completeness If the statement is true, the honest verifier will be convinced of this fact by an honest prover.

Soundness If the statement is false, no cheating prover can convince the honest verifier that it is true, except with some small probability.

Zero-knowledge If the statement is true, no cheating verifier learns anything other than this fact.

The general structure of zero-knowledge protocols is the following:

$$A \longrightarrow B : witness$$
$$B \longrightarrow A : challenge$$
$$A \longrightarrow B : response$$

The entity claiming to be A selects a random number from a predefined set, as its secret *commitment*, from which he computes the *witness*. This mechanism provide randomness which allows to distinguish different protocol runs. Upon reception of the *witness*, B issues a *challenge* to which only the legitimate party A can provide a correct *response*. To decrease the probability of successful cheating, the protocol is iterated if necessary.

The zero-knowledge concept is often presented with the Ali Baba's cave example [64], where *Peggy* wants to prove to *Victor* that she knows the secret password that allows her to open the cave's door, depicted in Figure 2.5.

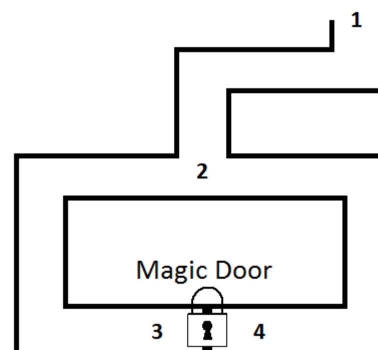


Figure 2.5: Ali Baba's Cave

Example 2.5.1 Peggy *wants to convince* Victor *that she knows the secret key to unlock the door between points 3 and 4, without having to reveal it. While* Victor *stands at point 1,* Peggy *enters the cave and stands either at point 3 or 4* ***(commitment)****. When ready,* Peggy *cries to Victor to come to point 2. At this point,* Victor *has no way to know whether* Peggy *is at point 3 or 4.* Victor *then calls to* Peggy*, asking her to come out either from the "left" or the "right" side of the passage* ***(challenge)****; at this point,* Peggy *might need to use the secret password to comply with the command* ***(response)****.*

Peggy *and* Victor *will then repeat this procedure until* Victor *is confident enough that* Peggy *knows the secret, but no matter how many times that the proof repeats,* Victor *does not learn the secret. If* Peggy *knows the secret, then she always passes the test* **(completeness)**. *If she does not,* Peggy's *chances of anticipating* Victor's *requests would become vanishingly small as the number of rounds increase, and she could only comply with* Victor's *request with probability* 2^{-n} **(proof)**. *For an impostor to fool* Victor *there must be an alternative way from 3 to 4 and anybody can use it* **(soundness).**

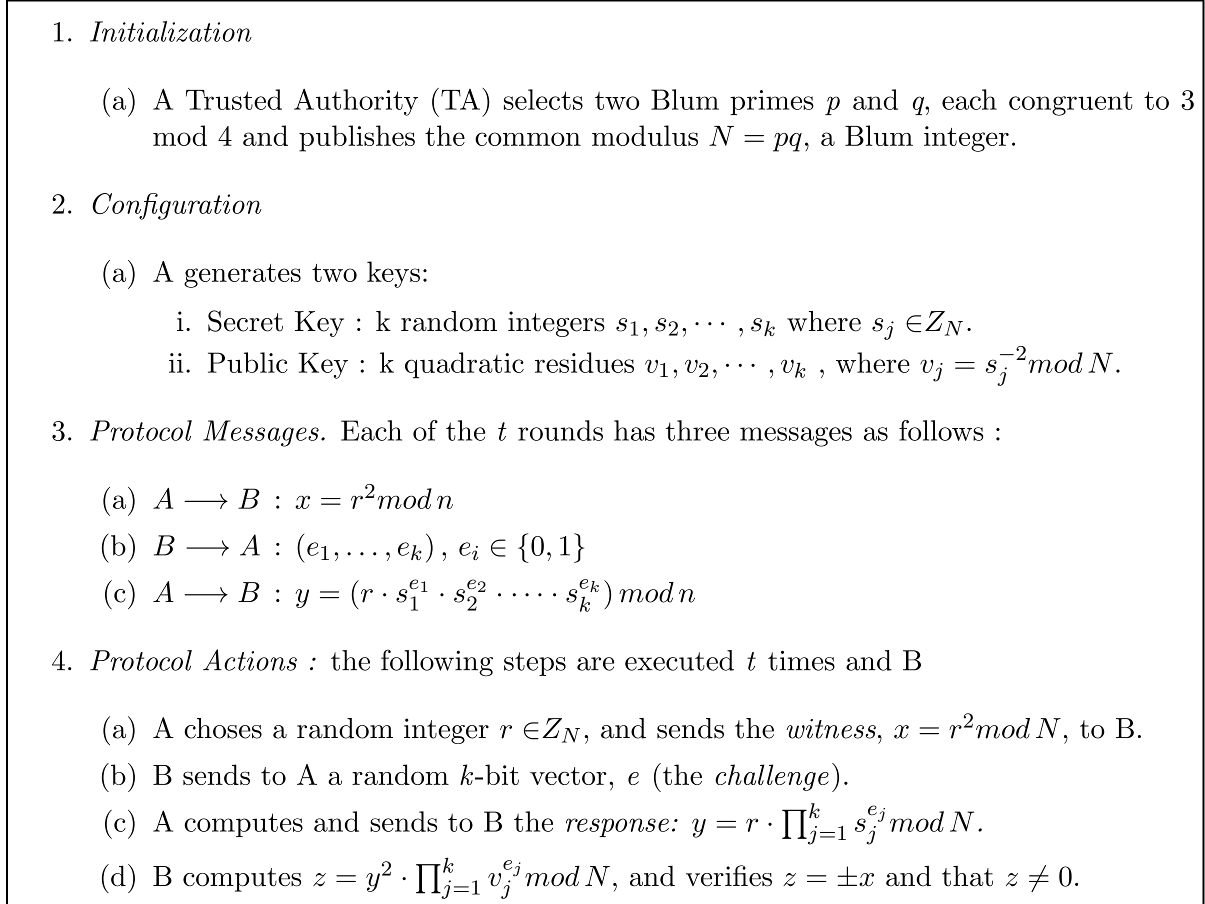
2.5.2 Feige-Fiat-Shamir authentication protocol

The Fiat-Shamir authentication protocol [27] requires a large number of iterations, consequently the authentication process is slow and computationally expensive for both the prover and the verifier. Nevertheless, a more efficient protocol exists: the Feige-Fiat-Shamir protocol [26] (Figure 2.6). This protocol is a variation of the Fiat-Shamir protocol and is based on the difficulty of computing square roots modulo composite numbers. The protocol is repeated t times, where a large enough t reduces the chances of an impostor successfully carrying out an impersonation.

The security of the FFS scheme is 2^{-kt}: provided that the factorization of N is difficult, the best attack has a probability 2^{-kt} of successful impersonation [5].

Example 2.5.2 (Feige-Fiat-Shamir protocol with artificially small paramenters)

1. *The* TA *selects* $p = 683$, $q = 811$, *yielding* $N = 553913$.
2. Peggy *selects her private keys:* $s_1 = 157$, $s_2 = 43215$, $s_3 = 4646$ *and computes her public keys:* $v_1 = 441845$, $v_2 = 338402$, *and* $v_3 = 124423$.
3. Peggy *selects* $r = 1279$, *and computes the witness:* $x = r^2 \, mod \, N = 528015$.
4. Victor *sends* Peggy *the 3-bit challenge: vector (0,0,1).*
5. Peggy *computes* $y = r \cdot s_1^0 \cdot s_2^0 \cdot s_3^1 \, mod \, N = r \cdot s_3 mod \, N = 403104$.
6. Victor *computes* $z = y^2 \cdot v_1^0 \cdot v_2^0 \cdot v_3^1 \, mod \, N = y^2 \cdot v_3 \, mod \, N = 25898$ *and verifies that* $z \neq 0$ *and that* $z = -x \, mod \, N$.



1. *Initialization*

   (a) A Trusted Authority (TA) selects two Blum primes p and q, each congruent to 3 mod 4 and publishes the common modulus $N = pq$, a Blum integer.

2. *Configuration*

   (a) A generates two keys:

      i. Secret Key : k random integers $s_1, s_2, \cdots, s_k$ where $s_j \in Z_N$.

      ii. Public Key : k quadratic residues $v_1, v_2, \cdots, v_k$, where $v_j = s_j^{-2} mod\, N$.

3. *Protocol Messages.* Each of the t rounds has three messages as follows :

   (a) $A \longrightarrow B \,:\, x = r^2 mod\, n$

   (b) $B \longrightarrow A \,:\, (e_1, \ldots, e_k)\,,\, e_i \in \{0, 1\}$

   (c) $A \longrightarrow B \,:\, y = (r \cdot s_1^{e_1} \cdot s_2^{e_2} \cdot \cdots \cdot s_k^{e_k})\, mod\, n$

4. *Protocol Actions :* the following steps are executed t times and B

   (a) A choses a random integer $r \in Z_N$, and sends the *witness*, $x = r^2 mod\, N$, to B.

   (b) B sends to A a random k-bit vector, e (the *challenge*).

   (c) A computes and sends to B the *response:* $y = r \cdot \prod_{j=1}^{k} s_j^{e_j} mod\, N$.

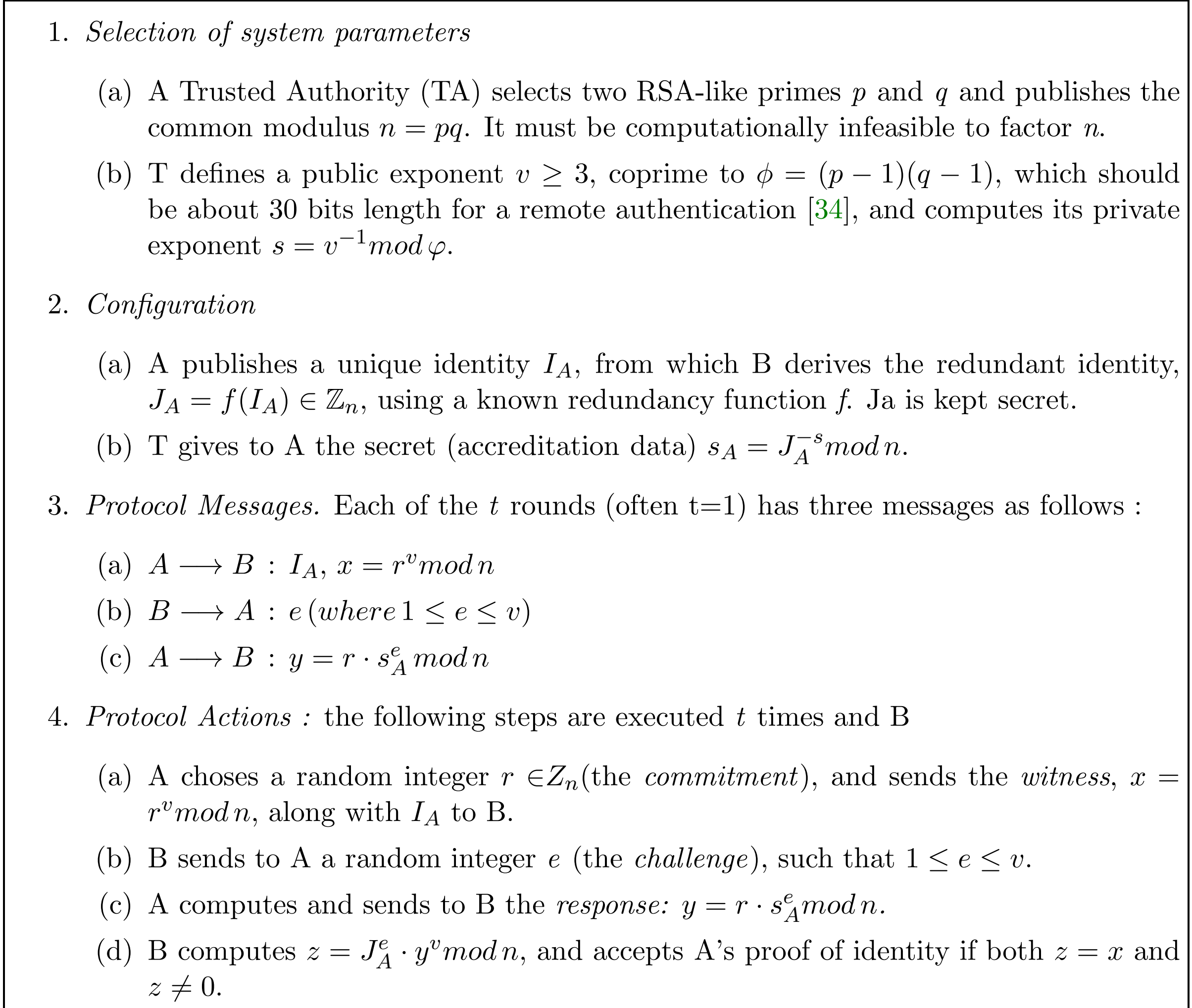
   (d) B computes $z = y^2 \cdot \prod_{j=1}^{k} v_j^{e_j} mod\, N$, and verifies $z = \pm x$ and that $z \neq 0$.

Figure 2.6: Feige-Fiat-Shamir protocol

2.5.3 Guillou-Quisquater authentication protocol

The GQ protocol [34], depicted in Figure 2.7, is an extension of the Fiat Shamir protocol that limits the number of rounds required; this enhancement is particularly suitable for resource-constrained devices, such as smart cards, and is achieved by reducing both the number of messages exchanged and the memory requirements for user secrets.

In the GQ protocol, v is the security parameter and determines the security level. The probability of false acceptance is equal to v^{-t}, where t is the number of iterations. Therefore, the recommended bitlength of v depends on the environment under which attacks could be mounted [5, 34]. For instance, for signature schemes it is recommended to use a public exponent v of at least 160 bits, nevertheless, in the corresponding authentication scheme, shorter exponents are allowed [85]. A small v allows more efficient computations, however it would require an a significant increase in the number of iterations. Therefore, in practice GQ protocols require only one iteration, t=1.



1. *Selection of system parameters*
   (a) A Trusted Authority (TA) selects two RSA-like primes p and q and publishes the common modulus $n = pq$. It must be computationally infeasible to factor n.
   (b) T defines a public exponent $v \geq 3$, coprime to $\phi = (p-1)(q-1)$, which should be about 30 bits length for a remote authentication [34], and computes its private exponent $s = v^{-1} mod\, \varphi$.
2. *Configuration*
   (a) A publishes a unique identity I_A, from which B derives the redundant identity, $J_A = f(I_A) \in \mathbb{Z}_n$, using a known redundancy function f. Ja is kept secret.
   (b) T gives to A the secret (accreditation data) $s_A = J_A^{-s} mod\, n$.
3. *Protocol Messages.* Each of the t rounds (often t=1) has three messages as follows :
   (a) $A \longrightarrow B : I_A,\, x = r^v mod\, n$
   (b) $B \longrightarrow A : e\, (where\, 1 \leq e \leq v)$
   (c) $A \longrightarrow B : y = r \cdot s_A^e\, mod\, n$
4. *Protocol Actions :* the following steps are executed t times and B
   (a) A choses a random integer $r \in Z_n$(the *commitment*), and sends the *witness*, $x = r^v mod\, n$, along with I_A to B.
   (b) B sends to A a random integer e (the *challenge*), such that $1 \leq e \leq v$.
   (c) A computes and sends to B the *response:* $y = r \cdot s_A^e mod\, n$.
   (d) B computes $z = J_A^e \cdot y^v mod\, n$, and accepts A's proof of identity if both $z = x$ and $z \neq 0$.

Figure 2.7: Guillou-Quisquater authentication protocol

Example 2.5.3 (GQ protocol with artificially small parameters and t=1)

1. *The* TA *selects* $p = 569$ *and* $q = 739$. *Therefore,* $N = p \cdot q = 420491$.

   TA *computes* $\phi = (p-1)(q-1) = 419184$, *selects* $v = 54955$ *and computes the secret value* $s = v^{-1}\, mod\, \phi = 233875$.

   The pair $(54955, 420491)$ *is made available to all users.*
2. *Suppose that* Peggy's *identity is* $J_A = 34579$.

   Peggy's *accreditation data is* $s_A = (J_A)^{-s}\, mod\, N = 403154$.
3. Peggy *selects* $r = 65446$ *and computes* $x = r^v\, mod\, N = 89525$. *Afterwards, she sends the pair* $(I_A, 89525)$ *to* Victor.
4. Victor *sends the challenge* $e = 38980$ *to* Peggy.
5. Peggy *computes* $y = r \cdot s_A^e\, mod\, N\ = 83551$ *and sends it to* Victor.
6. Victor *computes* $z = J_A^e \cdot y^v\, mod\, N = 89525$ *and authenticates* Peggy *since* $z = x$.

Chapter 3

Smart Card Architecture and Programming

It may well be doubted whether human ingenuity can construct an enigma... which human ingenuity may not, by proper application, resolve.

The Gold Bug
EDGAR ALLAN POE

Smart cards are devices much similar to credit cards, being capable of storing and processing information through the electronic circuits embedded in silicon in the plastic substrate of its body [14]. Despite its major hardware constraints, its portability, tamper resistance and capability to execute security protocols and algorithms, remain one of the mobile computing devices of choice [17, 75]. In fact, because smart cards can be used do provide authentication, identification and transaction processing, the smart card security market keeps expanding [24].

One of the most widely used multi-application smart card platforms [52], Java Card, is at least partly responsible for the success of smart cards as the "*write-once-run-everywhere*" concept brought smart card application developers more development flexibility and platform independence.

This chapter provides basic introduction to smart card technology. Extra information is available elsewhere [86, 66, 65, 52, 14].

3.1 History

Before the advent of smart cards, other types of cards have been used to provide some sort of identification. The story traces back to a little incident with the businessman Frank McNamara; unable to pay for dinner at a restaurant because he had left his wallet on another suit, he faced a terrible embarrassment and decided to create the Diners Club — see `http://www.dinersclubinternational.com`. The first cards date back to 1950, when the Diners Club issued the first "multi-purpose charge card".

Since synthetic material PVC was cheap and plastic cards began to proliferate, it wasn't long until Visa and Mastercard began issuing their own cards. A demand for machine-readable cards led IBM to develop the magnetic stripe cards in 1960. This step allowed to replace paper-based transactions by electronic data processing. However, this technology's security was brittle, as the data stored on the stripe could be read, deleted or rewritten by any one with the right equipment.

The enormous progress in microelectronics in the 1970s created the path for smart card creation, making it possible to integrate data storage and processing logic on a single silicon chip. The first smart card patents (1970s) and field trials with prepaid telephone cards (1984) soon followed. With the advances in chip technology and modern cryptography, smart cards' areas of application widened to the telecommunications (GSM networks: 1991), credit cards (EMV specifications: 1994) and electronic signatures (European directive [23]: 1999).

From a software developer's perspective, smart card software was initially rigid and monolithic [17], with closed proprietary systems that made the process of application development lengthy and difficult. Nevertheless, the success of open smart cards like MultOS [49] and Java Card [14, 6] became an important milestone in the history of smart cards. These brought flexible and interoperable mechanisms, by which multiple applications could be installed after the card had been issued.

3.2 Benefits and applications

Nowadays, smart cards are becoming ubiquitous, widely used in the telecommunication industry (SIM cards for mobile phones), payment and banking industries (fig. 3.1b), transportation(fig. 3.1a), health care, and citizen's electronic authentication (eID) (fig. 3.1c). For

example, the currently in progress Stork3 project — see http://www.eid-stork.eu/, aims at providing a reliable electronic authentication (eID) student mobility platform, with the objective of facilitating student's mobility across Europe.

(a) Lisboa Viva (b) university eID (c) Citizen card (PT)

Figure 3.1: smart cards

Some advantages of the smart card technology are listed bellow:

- security : smart cards are tamper-resistant devices with embedded chip which allow data storage, processing and personal key management in a secure way. Unlike magnetic stripe cards, smart card exploitation requires not only the physical possession of the card, but also intimate knowledge of the smart card hardware, software and specialized equipment.
- multi-application support : multiple applications can reside on a single card. Moreover, these can be installed and removed after the card has been issued, without compromising the security of the various applications.
- standardized features : standards such as the EMV [20], ISO7816 [39] , ISO 14443 [38] and GSM [22] ensure interoperability between different card manufacturers and different card readers (see section 3.3.4).
- cryptographic support : faster microprocessors and bigger storage capacity allow the exploitation of complex cryptographic algorithms. Current smart cards provide both symmetric and public-key cryptography through AES [25] and RSA [69], respectively, hashing (SHA-1 [63] , MD5 [68]) and digital signature schemes such as DSA and ECDSA [28]. Furthermore, these can be used to develop more advanced cryptographic protocols and provide security schemes (*e.g.*, authentication).

3.3 Smart card basics

In this section we present some basic concepts related to smart cards. In section 3.3.1 we introduce the two main types of smart cards, and in section 3.3.2 we describe their hardware. Communication between a smart card and a computer is described in section 3.3.3 and finally, in section 3.3.4, we outline a certain number of standards and specifications, which have been defined to achieve interoperability between smart card systems.

3.3.1 Smart card types

Smart cards are classified as memory card or microprocessor cards. The former are only capable of storing data because they do not contain a microprocessor. Because of their non-programmable logic they cannot be reprogrammed and reused, nevertheless, their low cost is adequate for prepaid services, such as public phones cards. Microprocessor cards, on the other hand, are more expensive than memory cards but possess of a central processing unit (CPU) and provides the card with multifunctional capabilities.

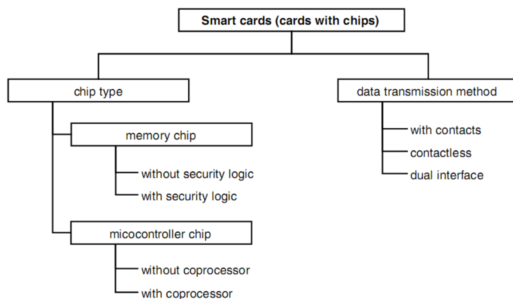


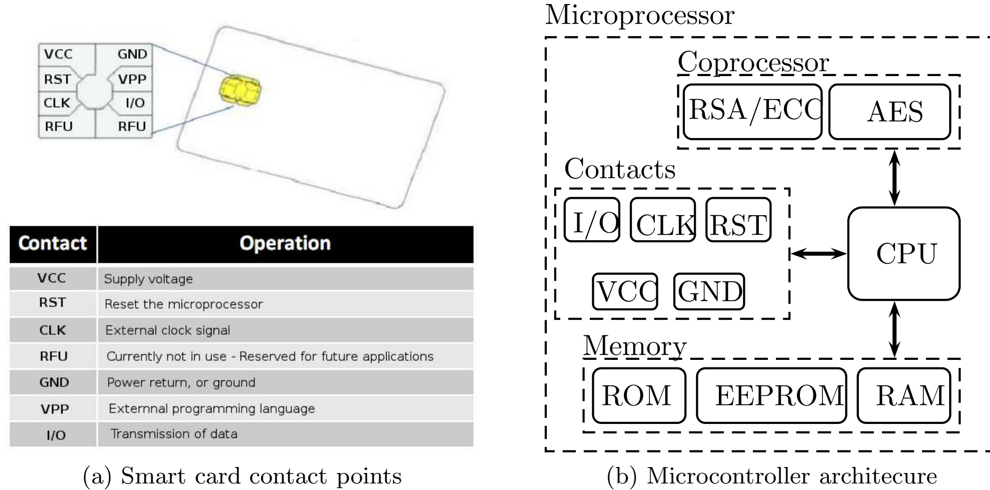
Figure 3.2: Card Types [65]

In terms of the access mechanism, smart cards can be further categorized as contact cards, contactless cards, or hybrid when they offer both interfaces. While Contact cards must be inserted in a card acceptance device (CAD), contactless cards communicate through an antenna, where energy and data are transferred without any electrical contact between the card and the terminal. These cards need not to be placed in a CAD, there is also no mandatory direction or orientation, in fact, these can even remain in the user's purse or wallet, making them particularly suited for public transport systems and payment — see Master Card's PayPass : http://www.paypass.com/.

3.3.2 Smart card hardware

Smart card contact points and architecture are depicted in figure 3.3. The Smart card architecture, depicted in figure 3.3b, is made of one embedded CPU; three types of memory:

electrical erasable program read-only memory (EEPROM), random access memory (RAM) and read-only memory (ROM); and may also have a coprocessor for mathematical computations (fig. 3.3b).



Contact	Operation
VCC	Supply voltage
RST	Reset the microprocessor
CLK	External clock signal
RFU	Currently not in use - Reserved for future applications
GND	Power return, or ground
VPP	Externnal programming language
I/O	Transmission of data

(a) Smart card contact points

(b) Microcontroller architecure

Figure 3.3: Smart card contact points and architecture

- Smart Card Contact Points: As depicted in figure 3.3a, a smart card has eight contact points which allow it to communicate with a CAD : Vcc, RST, CLK, GND, I/O and RFU.

- Processor : Most of CPUs on smart cards are 8-bit size. However those with a 16-bit or 32-bit microcontroller exist and are likely to become more common in the future. The clock signal is supplied externally as smart card processors usually do not have internal clock generators. Even though the standards restrict the clock signal to a range of 1-5 MHz, an internal clock multiplier allows cards to operate at higher frequencies.

- Coprocessor : Smart cards have very limited resources, and without a specialized mathematical coprocessor, some cryptographic operations would otherwise be infeasible. Security applications which involve modular arithmetic and large-integer calculations commonly resort to the coprocessor (*e.g.*, RSA [69], ECC [28]).

- Memory System :

  - read-only-memory (ROM) : This persistent memory can only be programmed once, by the manufacturer, and usually includes the operating system routines, cryptographic algorithms and transmission protocols.

- EEPROM : Data like the smart card applications and operating system parameters is stored in this type of memory. It can hold data after the power supply is switched off and also be erased electronically and rewritten. However, writing this memory is considerably slower than RAM, which should be carefully taken into consideration when designing and implementing applications. For instance, Chen [14] says that writing to EEPROM is 1000 times slower than writing to RAM, while Karima [67] from 10 to 50 times slower.
- RAM : RAM is the fastest and most scarce type of memory in a smart card, meant to store and modify temporary data. The data is stored temporarily in RAM, being immediately lost when the power supply is switched off.
- Depending on the application area, memory capacity may range from 16 to 400KB of ROM, 1 to 500KB of EEPROM and 256 bytes to 16KB of RAM [66].

3.3.3 Communication models

In order to communicate with a computer, a card interacts with a Card Acceptance Device (CAD). This device can either be a reader, which in turn communicates with the computer it is connected to, or a terminal if it comprises both the tasks of a reader and a computer (*e.g.,* ATM machine). In this communication model (fig. 3.4), the applications that communicate with the smart card are called host applications. Smart cards adopt client-server paradigm, being smart card the server and the host application the client. Due to the client-server paradigm, communication between a host and a CAD is half duplex, therefore data can only be sent in one direction at a time. This is achieved through a request-response protocol (fig. 3.4), in which application protocol data units (APDU's) are exchanged. It is the host who initiates the communication, which he does by sending to the card a *command* APDU (C-APDU), thereafter the smart card replies with a *response* APDU (R-APDU). The smart card state machine is depicted in Figure 3.5.

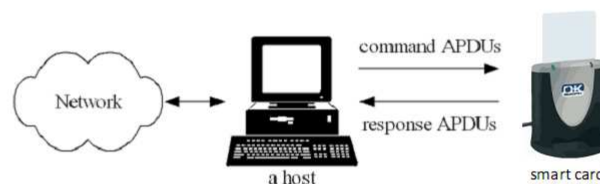


Figure 3.4: Smart card communication model

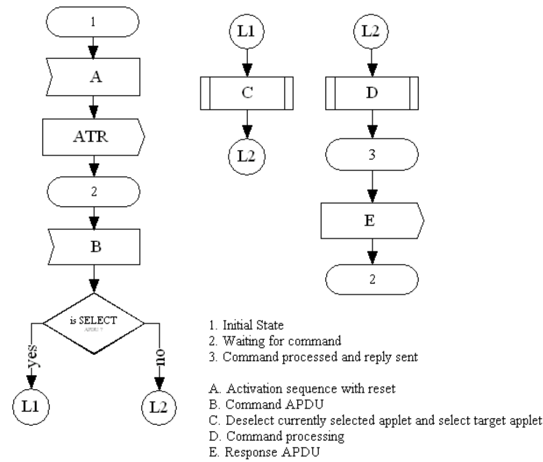


Figure 3.5: Smart card state machine

The structure of APDUs are illustrated in table 3.1. In the C-APDU the *CLA* byte identifies the class of instructions and the *INS* byte further specifies the specific instruction, while bytes *P1* and *P2* provide extra parameters. The *SW1* and *SW2* bytes form the status word, which is used to provide feedback about the execution of the C-APDU. Several status words are predefined in the ISO 7816 standard [39]; examples of status words are "0x9000", which means that the command was successfully executed and "0x6D00" for an invalid INS value. The remaining fields are optional: the *data field* may contain up to 255 bytes, where *Lc* defines the number of data bytes in the C-APDU and *Le* the maximum number of bytes expected in the R-APDU *data field*.

Table 3.1: APDU structure

(a) C-APDU

Mandatory header				Optional body		
CLA	INS	P1	P2	Lc	Data field	Le
1 byte	*1 byte*	*1 byte*	*1 byte*	*1 byte*	*up to 255 bytes*	*1 byte*

(b) R-APDU

Optional body	Mandatory trailer	
Data field	SW1	SW2
up to 255 bytes	*1 byte*	*1 byte*

When powered up, or after receiving a RST command, a smart card sends out an answer to reset (ATR) message to the host. ATR message contains various parameters related to the transmission protocol; card hardware parameters but also allows the host to identify the card as cards from the same family share the same ATR (table 3.2). In this thesis, a Gemalto TOP GX4 [4] smart card is used.

Transmission protocols are designated as "T=" (for 'transmission protocol') plus a sequential

Table 3.2: sample ATRs

Manufacter/family	ATR
Portuguese Citizen's card	3B 7D 95 00 00 80 31 80 65 B0 83 11 C0 A9 83 00 90 00
IST eID card	3B 29 00 80 72 A4 45 64 00 FF 00 10
Lisboa Viva subway pass from Portugal	3B 6F 00 00 80 5A 08 06 08 20 02 00 92 37 89 73 82 90 00
Lisboa Viva subway pass Sub23	3B 6F 00 00 80 5A 08 06 08 20 02 00 92 55 3C 39 82 90 00
Gemalto TOP GX4	3B 7D 96 00 00 80 31 80 65 B0 83 11 D0 A9 83 00 90 00

number. The two most used protocols are called T=0 and T=1 [65], where the former is asynchronous, half-duplex, byte oriented, and the latter is asynchronous, half-duplex, block oriented.

3.3.4 Standards and Specifications

Many standards and specifications have been developed over the years to ensure the interoperability between smart card systems. Along the way, many projects have born, evolved, fused together or even dropped:

- ISO/IEC 7816 [39]: The most important standard regarding smart cards, it defines multiple aspects of smart cards, such as physical characteristics, transmission protocols and their security architecture are defined by this international standard.
- ISO/IEC 14443 [38]: Describes the properties and operation modes of contactless smart cards with a range of approximately 10cm.
- GSM [22, 56]: Set of standards that cover the use of smart cards in public and cellular telephone systems. GSM devices use Subscriber Identity Module (SIM) smart cards, which are security modules capable of holding personal identity information and performing security operations, such as entity authentication.
- EMV: Europay, MasterCard and VISA defined this specification, based on the ISO 7816, with the aim of promoting a global standard for interoperability between smart-card-based payment systems — see `http://www.emvco.com/`.
- Open/Global platform [51]: GlobalPlatform specifications ensure secure and interoperable deployment and management of smart card applications, regardless of technology vendor or service provider. These specifications encompass the communication between smart cards, CADs and the host's system infrastructure.

- OpenCard Framework (OCF): Java API to access both to smart card readers and the applications deployed on the smart cards. The project was discontinued, however, other projects are still using OCF, namely the Open Smart Card Development Platform (OpenSCDP) — see http://www.openscdp.org/.

- PC/SC specification [2] (Interoperability Specifications for ICCs and Personal Computer Systems) defines a general purpose architecture for using smart cards on personal computer systems.

- M.U.S.C.L.E (Movement for the Use of Smart Card in a Linux Environment) — see http://www.linuxnet.com/, defines a Linux API, a set of compliant drivers and a resource manager through a GNU environment. MUSCLE is built on PC/SC, but unlike PC/SC, the source code is openly available under a GPL license.

3.4 Chip Operating Systems

The smart card's chip operating system (COS) is a sequence of instructions, permanently embedded in the ROM, that allow user applications to be stored from an outside development system and provide resources for their execution.

Today's smart card's COSs are nothing like the monolithic first-generations of smart cards, which did not allow the management and execution of third-party applications. As a result, early smart cards were inflexible and failed to provide portability, since applications needed to be developed with a single microprocessor in mind.

It was the introduction of open smart card platforms, namely Multos [52] and Java Card [14, 6] that allowed both hardware abstraction and multi-application deployment. Later on, other technologies emerged, such as Windows for Smart Cards (WfSC), BasicCard and smart card .NET [86, 65, 66, 52].

While Multos and Java Card remain the most widely used smart card platforms, the latter appears to be the one enjoying the widest acceptance amongst security researchers. WfSC was intended to be an alternative to Java Card, however, due to lack of acceptance by the smart card industry, the project was abandoned by Microsoft. On the other hand, BasicCard and SmartCard.NET platforms appear to be growing in popularity, specially the latter one, taking into consideration the growing numbers of published articles involving these smart cards.

Note that not all of the above technologies are smart card operating systems : Multos and WfSC are underlying smart card operating systems, whereas Java Card, Basic Card and smart card .Net are located on top of a smart card operating system. For example, both Gemalto's TOP and NXP's JCOP smart cards are Java Card and Global Platform compliant; nevertheless, each manufacturer provides a different underlying operating system.

The basic functions of an operating system that are common across all smart card products are [66]:

1. Management of interchange between the card and the outside world, primarily in terms of interchange protocol.
2. Management of local files and data held in memory.
3. Access control to information and functions.
4. Management of card security and cryptographic algorithm procedures.

A brief overview of the above mentioned technologies is provided in sections 3.4.1 to 3.4.4. More information on the above mentioned technologies can be found in [65, 52, 19], since a detailed comparison of these platforms is out of the scope of this thesis.

3.4.1 Java Card

Java Card technology [14, 6] enables programs written in a subset of Java programming language to run on smart cards and other resource-constrained devices, where applications written for the Java Card platform are referred to as applets. Due to the smart card hardware limitations, only a subset of the features of the Java can be supported. Furthermore, Java virtual machine (JVM) must be distributed between the smart card and the workstation. The supported Java subject of Gemalto's TOP GX Java Card [4] is depicted in Table 3.3. However, note that garbage collection and integer data type support are features optional to smart card builders and, therefore, are not supported by all cards. An overview of the architecture of a Java Card system is shown in Figure 3.6.

The Java Card runtime environment (JCRE) manages card resources, network communications, applet execution and security. It also makes sure that different applets do not interfere through a security mechanism referred to as applet firewall.

Table 3.3: Supported and unsupported Java features

Supported	Unsupported
boolean, byte, short	int, long, double, float, char
one-dimensional arrays	string, multi-dimensional arrays
packages, classes, interfaces , exceptions	dynamical class loading
objects	object cloning, object serialization
garbage collection	threads
	security manager

The bottom layer of the JCRE contains the Java Card virtual machine (JCVM) and native methods. The JCVM executes bytecodes, controls memory allocation, manages objects, and enforces runtime security. Unlike the Java virtual machine (JVM), the Java Card Virtual Machine is split between the card and the workstation. The former is referred to as the on-card VM, or *interpreter*, and the latter as off-card VM, or *converter.* While the *converter* loads and processes the class files and outputs a CAP (converted applet), the *interpreter* executes the CAP, by which we mean that it executes bytecode instructions and ultimately executes applets. Another component of the JCVM is the Java Card application framework classes (APIs) which provides functions coded in the native instructions of the target processor. Since this can yield a considerable increase in processing speed, APIs should be used as much as possible (*e.g.*, cryptographic operations).

The steps for creating and downloading a Java Card application are summarized as follows:

1. At the workstation, the application programmer writes the Java source code and compiles it, with a standard Java compiler, creating a class file and an export file. At this point, the process is identical to Java programming for PCs.

2. The class file is then transferred to the Java Card Converter (the off-card portion of the VM), which performs static tests and, if all these tests are passed successfully, delivers a second export file and a card application file (CAP file).

3. The applet is loaded into the smart card in the form of a CAP file, which is often carried out using GlobalPlatform.

4. Oncard VM (interpreter) tests and interprets the bytecode line by line and generates machine instructions for smart card processor from bytecode.

Java Card offers several advantages which could explain why it has enjoyed such a wide acceptance from programmers : extensive documentation, development tools, cards and code portability. On the other hand, Java is an interpreted language, which comes at a performance price. Nevertheless, it is relatively difficult to make fair comparisons between assembler or C programs and Java. We can only assume that with proper use of the Java API, the execution time will be approximately 50% longer than for a comparable implementation in C or assembler [66].

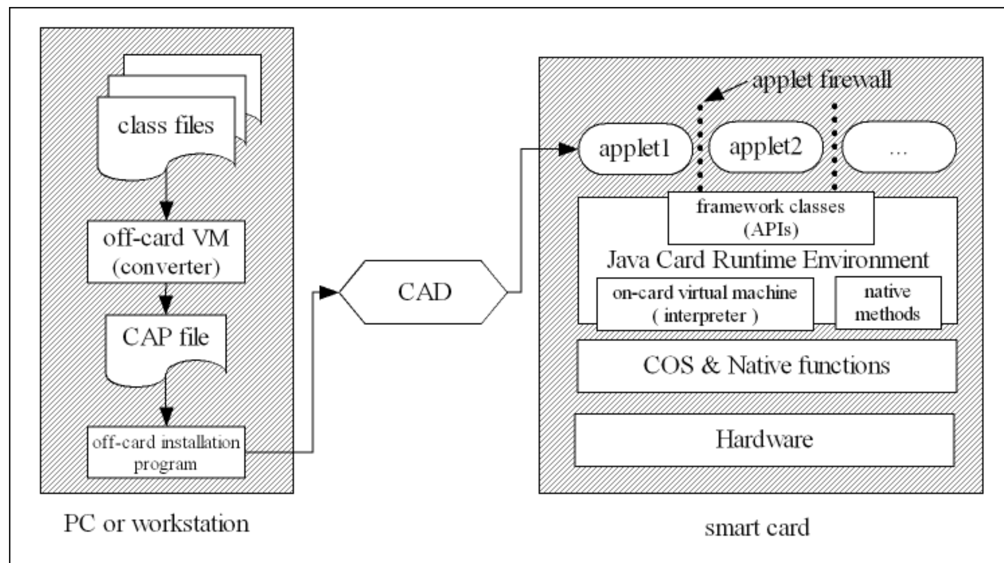


Figure 3.6: Java Card system architecture

3.4.2 MultOS

The internal architecture of a MultOS card is depicted in Fig. 3.7. The MultOS operating system is executed natively by the microprocessor and provides communication with the underlying hardware and the virtual machine, where the applications are executed. It provides the basic required functionality of I/O, file management, cryptographic services, application management and command dispatchment.

The MultOS operating system can only execute Multos Executable Language (MEL) byte code, the language for MultOS applications. Both security and hardware abstraction are achieved since the MEL language is interpreted not by the underlying smart card hardware but by the MultOS interpreter. Nonetheless, MultOS software development is not confined to MEL bytecodes; applications can be developed in high-level languages like C and Java

and then converted to MEL through compilers. Language independence is probably the most important feature of MultOS [75]; this flexibility, however, comes with a price : a developer has less control over the produced byte-code, and for example, that may result in larger and of less performance code [41]. Nevertheless, the significantly smaller code size and an overall superior performance over the competitors, remain attractive features [19].

MultOS smart card operating system takes security and performance into serious consideration. For example, certificates are required to manage applications on the smart card (with exception of the development cards).

However, MultOS' closed design and the difficulty in obtaining development tools and cards put constraints on its deployment [41]. Progressively, the entities promoting MultOS have realised their mistakes and attempted to overcome some of the forementioned problems, but the impact on the market remains to be seen [52]. Currently, extensive documentation as well as the SmartDeck development environment are available free-of-charge and can be found in the MultOS' website — see `http://www.multos.com/developer/`.

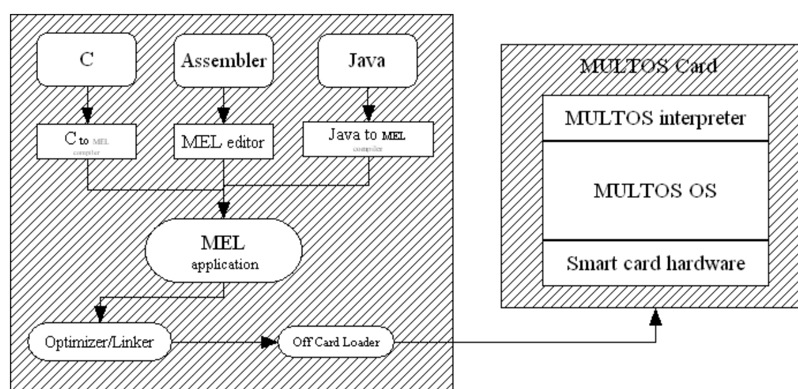


Figure 3.7: MultOS architecture

3.4.3 BasicCard

BasicCard [52] is a smart card platform owned by ZeitControl Cardsystems. Applications are written in the Basic language, being particularly suited for constrained devices.

According to [65], the program code is very compact and the execution speed is relatively high when compared with other smart card operating systems with interpreters.

Nevertheless, the most advertised feature is the low selling price of the hardware compared to other smart cards such as Multos or Java Card — see `http://www.basiccard.com/`.

The BasicCard offers a number of additional programming libraries (mainly cryptographic) that aim to support the development of advanced and dedicated applications [52]. BasicCards support different Cryptographic functionalities, ranging from Public-Key algorithms (RSA, EC) to symmetric-key algorithms (DES, AES) and Hashing (SHA-1, SHA-256).

3.4.4 Smartcard.Net

The .NET Card technology is a multi-application, multi-language, smart card platform that allows the integration of smart cards with other .NET based technologies.

The platform is based on the HiveMinded Smartcard.NET reference implementation of the Common Language Infrastructure (CLI), and developed as close as possible to the international standardisation organisation ECMA335 specifications [18], and the .NET framework [52].

An appropriate subset of the .NET class libraries allows applications to run on smart cards while supporting a programming model consistent with that of the full .NET framework. In addition, applications can be written in any .NET-compliant programming language, such as C#, C++, Visual Basic (VB), J# or JavaScript.

However, at the time of writing this dissertation it was not easy to obtain detailed information about the platform. Moreover, we were only aware of Gemalto's and Feitian's .NET smart cards — see http://www.gemalto.com/products/dotnet_card/ and http://www.ftsafe.com/products/dotNet-Card.html.

3.5 Choice of smart card platform

Considering all the smart card platform available and the fact that each platform has unique selling points [19], the choice is far from obvious. Moreover, there are constant improvements to the actual specifications, platforms, and smart card hardware; therefore indicative comparison factors might not be valid [52].

In terms of performance, BasicCard and Multos appear to be best choices, with the former focusing on the low cost of the cards and the latter taking security into very serious consideration. However, we could not find any published research on BasicCard and very little on

Multos [41]. SmartCard.NET seems promising and improving in popularity, but there is still not much information available though [29, 80].

We have chosen to implement the authentication protocols on a Java Card smart card, even though it should have been possible to implement it on any of the other platforms.

There were no strong reasons motivating this choice - the subtle reason is that Java Card has enjoyed wide support from a number of programmers, and as a result a number of development tools and extensive documentation became widely available — see Java Card Forum: http://www.javacardforum.org/, Java Card Documentation: http://www.oracle.com/technetwork/java/javacard/documentation/index.html and Chen [14].

Contrary to the other platforms, Java Card smart cards feature in a variety of published articles, with research topics ranging from identity & privacy protection [11, 9, 82, 84] and attacks on smart cards [81, 10] to performance measurement on smart cards [67, 60, 61].

The main advantages and disadvantages of each smart card platform are summarized in Table 3.4.

Table 3.4: Overview of smart card platforms

Platform		Description
Java Card	+	Extensive documentation, development tools and cards
	-	Performance
MultOS	+	Multi-language support, performance and security (ITSEC6)
	-	Closed design, card acquisition process
BasicCard	+	Performance, low-priced cards and free development software
	-	Single manufacturer, only supports BASIC language
.NET	+	Multi-language support, integration with Microsoft environments
	-	Detailed information unavailable
WfSC	+	—
	-	Project was cancelled

Chapter 4

Implementing Big Number Operations

> Teaism is a cult founded on the adoration of the beautiful among the sordid facts of everyday existence. It is essentially a worship of the Imperfect, as it is a tender attempt to accomplish something possible in this impossible thing we call life.
>
> The Book of Tea
> KAKUZO OKAKURA

This chapter describes the multi-precision integer arithmetic routines required to implement the authentication schemes presented in Chapter 2. These cryptosystems are based on modular arithmetic, such as modular multiplication and exponentiation, performed on very large numbers, thus requiring efficient methods to perform complex computations in $\mathbb{Z}_m$.
However, the Java-Card platform provides no adequate support to perform such operations. The only exception is the BigNumber class, from the optional package *javacardx.framework.math*, which is only be available in version 2.2.2 of the Java Card API. Not only there are few cards which implement version 2.2.2, but also the access to the cryptographic coprocessor is very limited, since only addition and multiplication are available and modular arithmetic is not implemented.
In addition, the cryptographic coprocessor is not directly accessible, which is further aggravated by the overhead caused by the JVM and the impossibility of optimization through low-level programming. Therefore, in order to be able to carry out the authentication fast enough for practical use, we must exploit the cryptographic API to gain access the

cryptographic processor.

The remainder of the chapter is organized as follows. In section 4.1 we introduce the addition and subtraction functions. Subsequently, in section 4.2 the focus is put on the multiplication, followed by exponentiation in section 4.3. In these last two sections we present two different approaches; one software implementation and another which relies on the cryptographic coprocessor to perform the calculations.

4.1 Large Number addition and subtraction

The two most elementary multiple-precision operations are the addition and subtraction, which are usually based on the classic pencil-and-paper, or Fibonacci, method [5]. Since the operations are carried out word by word, a suitable base, b, should be chosen so that $(x_i+y_i+c)\,mod\,b$ can be computed as efficiently as possible by the hardware on the computing device. Since most smart cards do not support integers, the base is stored in a byte array and shorts are used to store the intermediary results. Note that the word size has an impact on the performance of the algorithm; as the size of the word increases, the number of computations necessary to carry out the operation decreases as well.

The algorithms presented in this section are for computing addition, subtraction as well their respective modular operations. We also discuss the algorithm efficiency as big-Oh function [74] over the number of bits.

4.1.1 Addition and subtraction

Algorithms 1 and 2 describe how two large numbers can be added and subtracted, respectively.

Addition and subtraction complexity is $\mathcal{O}(n)$.

4.1.2 Modular Addition and Subtraction

Let $x = (x_{n-1} \cdots x_0)_b$ and $y = (y_{n-1} \cdots y_0)_b$ be two integers verifying $x, y \in \mathbb{Z}_m$, where m is the modulus. Extending Algorithms 1 and 2 in order to obtain the respective modular

Algorithm 1 Multiple-precision addition

```
Input: x = (x_{n-1}, ··· , x_0)_b and y = (y_{n-1}, ··· , y_0)_b
Output: w = x + y = (w_n w_{n-1} ··· w_1 w_0)_b
 1: carry ← 0;
 2: for i=0 to n-1 do
 3:     w_i ← (x_i + y_i + carry) mod b;
 4:     if (x_i + y_i + carry) > b then
 5:         carry ← 1;
 6:     else
 7:         carry ← 0;
 8:     end if
 9: end for;
10: w_n ← carry;
11: return(w);
```

Algorithm 2 Multiple-precision subtraction

```
Input: x = (x_{n-1}, ··· , x_0)_b and y = (y_{n-1}, ··· , y_0)_b
Output: w = x - y = (w_n w_{n-1} ··· w_1 w_0)_b
 1: carry ← 0;
 2: for i=0 to n-1 do
 3:     w_i ← (x_i - y_i + carry) mod b;
 4:     if (x_i - y_i + carry) ≥ 0 then
 5:         carry ← 0;
 6:     else
 7:         carry ← -1;
 8:     end if
 9: end for;
10: return(w);
```

operations is straightforward if we observe that

$$(x + y) \, mod \, m = \begin{cases} x + y, & if \, x + y < m \\ x + y - m & if \, x + y \geq m \end{cases} \tag{4.1}$$

Therefore, modular addition (Algorithm 3) and subtraction (Algorithm 4) can be performed without the need of long division, requiring only that we compare the result against the modulo and eventually perform a subtraction or an addition, respectively.

Therefore, modular addition and subtraction precision is $\mathcal{O}(n)$.

Algorithm 3 Large number modular addition

Input: $m = (m_{n-1}, \cdots, m_0)_b$, $x = (x_{n-1}, \cdots, x_0)_b$ and
$y = (y_{n-1}, \cdots, y_0)_b$, (x,y)<m
Output: $z = x + y \, mod \, m = (z_{n-1}, \cdots, z_0)_b$
1: $z \leftarrow x + y$; % Algorithm 1
2: if $z \geq m$ then
3: $z \leftarrow z - m$; % Algorithm 2
4: end if
5: return(z);

Algorithm 4 Large number modular subtraction

Input: $m = (m_{n-1}, \cdots, m_0)_b$, $x = (x_{n-1}, \cdots, x_0)_b$ and
$y = (y_{n-1}, \cdots, y_0)_b$, (x,y)<m
Output: $z = x + y \, mod \, m = (z_{n-1}, \cdots, z_0)_b$
1: $z \leftarrow x - y$; % Algorithm 2
2: if $z < 0$ then
3: $z \leftarrow z + m$; % Algorithm 1
4: end if
5: return(z);

4.2 Large Number Multiplication

Modular multiplication and modular exponentiation are the most common operations in RSA public-key cryptosystems. Moreover, modular exponentiation is composed of a sequence of modular multiplication operations, which means that an efficient implementation of modular reduction is the key to high performance [13].
Many software and hardware efficient implementations have been proposed to reduce the execution time modular multiplication [57, 21]. Two of the methods which have received more attention in the literature are due to Barrett [8] and Montgomery [55]. These techniques aim to reduce the computational requirements of the operation by avoiding to explicitly carry out the classical reduction step, i.e. to compute the remainder on a division by m. Several other fast reduction algorithms have been proposed though; for more information, the reader may consult [5].

In this section we describe and analyse the different approaches used to compute modular multiplication, discussing which are more adequate for a Java Card implementation. We start

by introducing our implementation of large integer multiplication and thereafter modular multiplication. Since these operations are not supported by the Java Card API, later in this section we describe an improved modular multiplication method, which accesses the cryptographic hardware of the smart card in order to speed up the computations.

4.2.1 Large Integer Multiplication

Algorithm 5, based on the algorithm in [5], describes a standard large number multiplication based on the classical pencil-and-paper method. Multiplication is done word by word, creating a carry word each time, and the intermediate result, $(uv)_b$, is stored in a double word, where u and v are base b digits, and u may be zero.

Algorithm 5 Standard large number multiplication

Input: $x = (x_{len1-1}, \cdots, x_0)_b$ and $y = (y_{len2-1}, \cdots, y_0)_b$, (x,y)<m
Output: $z = x \cdot y = (z_{len1+len2-1}, \cdots, z_0)_b$
1: $z \leftarrow 0$;
2: for $i = 0$ to $len_1 - 1$ do
3: $carry \leftarrow 0$;
4: for $j = 0$ to $len_2 - 1$ do
5: $(uv)_b \leftarrow z_{i+j} + x_j \cdot y_i + carry$;
6: $z_{i+j} \leftarrow v$;
7: $carry \leftarrow u$;
8: end for
9: $z_{i+len_2} \leftarrow u$;
10: end for
11: return(z);

Taking $n = len_1 = len_2$, standard multiplication complexity is $\mathcal{O}(n^2)$

4.2.2 Russian Multiplication

In base 2, multiplication can be implemented without requiring a multiplication table. Such an example is the multiplication algorithm discussed below, commonly known as the Russian Peasant Multiplication. This algorithm is very fast in hardware as it processes multiplication as a series of binary shifts and additions. Addition was already discussed in section 4.1 and division or multiplication by 2 can be implemented, respectively, by a right or left shift of bits. Russian peasant multiplication can be computed by Algorithm 6. It decomposes one of the multiplicands (generally the larger) into a sum of powers of two and creates a table of doublings

Algorithm 6 Russian Peasant Algorithm

```
Input: Three positive integers a,b and c
Output: z = a · b
1: x ← a; y ← b; z ← 0;
2: while (y > 1) do
3:   if LSB(y)=='1' then
4:     z ← z + y;
5:   end if
6:   x ≪ 1; y ≫ 1;
7: end while
8: z ← z + x;
9: return(z);
```

of the second multiplicand. The algorithm works by shifting the multiplicand one bit to the right (halving) and the multiplier one bit to the left (doubling). However, when the multiplicand's least significant bit (LSB) is '1', we must add the multiplier to the result prior to shifting. We repeat this procedure while the multiplicand is greater than one and terminate by adding the multiplier to the result.

Example 4.2.1 *To multiply 19 by 11, double the 19 and halve the 11, and add the doubles that correspond to an odd number in the result column.*

19	*× 11*	*result*	
19	*11*	*0*	*double 19, halve 11, add 19 to result*
38	*5*	*19*	*double 38, halve 5, add 38 to result*
76	*2*	*57*	*double 76, halve 2*
152	*1*	*57*	*add 152 to result, terminate*
		209	

The complexity of Russian peasent multiplication depends on the operand values. The shift right operation increases the bit number to $n+1, n+2, \ldots, 2n$ but the adding complexity remains $\mathcal{O}(n)$. The main issue is the LSB value of the each of n multiplicants: in the worst-case, all are 1 and complexity is $\mathcal{O}(n^2)$, in the best-case only one multiplicant is odd and complexity is $\mathcal{O}(n)$.

4.2.3 Modular multiplication

The simplest algorithm for modular multiplication, $xy \mod m$, consists in the computation of $x \cdot y$, followed by a reduction of the result modulo m. This method, often referred to as the

classical algorithm for modular multiplication, requires significant computational effort, since modular reduction is closely related to division and large dimension intermediate products are computed. On the other hand, methods which do not explicitly carry out a division step do require extra calculations, i.e., pre-calculations, argument transformations, and post-calculations.

Barrett reduction

Barrett reduction (Algorithm 7) computes $r = x \mod m$ given x and m, through an estimation of the quotient, in a way that the required operations are less expensive than division. The Barrett reduction method requires the precomputation of one parameter, $\mu = \lfloor b^{2k}/m \rfloor$, which does not change as long as the modulus remains constant.

Algorithm 7 Barrett modular reduction

Input: $x = (x_{2k-1} \cdots x_1 x_0)_b$, $m = (m_{k-1} \cdots m_1 m_0)_b$ with $m_{k-1} \neq 0$, and $\mu = \lfloor b^{2k}/m \rfloor$.
Output: $r = x \mod m$.
1: $q_1 \leftarrow \lfloor x/b^{k-1} \rfloor$; $q_2 \leftarrow q_1 \cdot \mu$; $q_3 \leftarrow \lfloor q_2/b^{k+1} \rfloor$;
2: $r_1 \leftarrow x \mod b^{k+1}$; $r_2 \leftarrow q_3 \cdot m \mod b^{k+1}$; $r \leftarrow r_1 - r_2$;
3: If $r < 0$ then
4: $r \leftarrow r + b^{k+1}$;
5: end if
6: While $r \geq m$ do $r \leftarrow r - m$;
7: Return(r);

Note that if b is a power of 2, divisions and modular reductions can be replaced, respectively, by right-shifts and AND operations (*i.e.*, truncation of the least significant bits of the operand). This results that the remaining operations are addition and multiplication, both of which are less expensive than division.

Montgomery's multiplication

In order to compute modular multiplication, Montgomery's multiplication (Algorithm 8) requires a *m-residue* transformation as well as a pre- and post-computation step. Despite not being suitable for single modular multiplications, it is very effective for performing modular exponentiation (see section 4.3.1).

The basic idea of Montgomery's theorem is to replace division by m with division by 2^k. First

we choose a positive integer coprime greater than m, R, *i.e.* $R > m$ and $gcd(m, R) = 1$. If m is represented as a base b integer of length n, where b is the length of the machine word, then a typical choice for R is b^n. Furthermore, if m is odd (such as RSA moduli), then b can be a power of 2 and $R = b^n$ will meet the condition $gcd(m, R) = 1$. Letting R be a power of two allows multiplication, division and modulo by R to be done by shifting or logical operations.

Let x, y and m be three integers of length n and $0 \le x, y < m$, with $x = (x_{n-1} \cdots x_0)_b$ and $y = (y_{n-1} \cdots y_0)_b$. For m odd, the Montgomery multiplication of x and y modulo m can be computed by Algorithm 8, taken from [5].

Algorithm 8 Montgomery multiplication

Input: $x = (x_{n-1} \cdots x_1 x_0)_b$, $, y = (y_{n-1}, \cdots, y_0)_b$ and $m = (m_{n-1} \cdots m_1 m_0)_b$, with $0 \le x, y < m$, $R = b^n$ with gcd(m,b)=1, and $m' = -m^{-1} \mod b$.
Output: $A = (a_n \cdots a_0)_b = xyR^{-1} \mod m$.

1: $A \leftarrow 0$;
2: for i=0 to (n-1) do
3: $u_i \leftarrow (a_0 + x_i \cdot y_0)m' \mod b$;
4: $A \leftarrow (A + x_i \cdot y + u_i \cdot m)/b$;
5: end for
6: if $A \ge m$ then
7: $A \leftarrow A - m$;
8: end if
9: return A

Computational efficiency of Algorithm 8: Suppose x, y and m are n-digit base b integer, with $0 \le x, y < m$. Neglecting the cost of the precomputation in the input, Algorithm 8 computes $xyR^{-1} \mod m$ with $2n(n + 1)$ single-precision multiplications [5].

Example 4.2.2 *(Montgomery multiplication)*

In Algorithm 8, let $m = 0x7d$, $x = 0x2b$ *and* $y = 0x5c$. *Here,* $n = 1$, *therefore* $R = 256^1$ *and* $m' = -m^{-1} \mod 256 = 0x2b$. *The steps in Algorithm 8 are the following:*

$$\text{(line 3)} \begin{cases} u_i \leftarrow & (a_0 + x_i y_0)m' \mod b \\ u_0 \leftarrow & (0 + 0x2B \cdot 0x5C) \cdot 0x2B \mod 0x100 = 0x7C \end{cases}$$

$$\text{(line 4)} \begin{cases} A \leftarrow & (A + x_i y + u_i m)/b \\ u_0 \leftarrow & (0 + 0x2B \cdot 0x5C + 0x7C \cdot 0x7D)/0x100 = 0x4C \end{cases}$$

Hence, the output of the algorithm is $xyR^{-1} \mod m = 0x4C$*. In order to obtain* $xy \mod m$*, we must convert the previous result from the Montgomery domain back to the integer domain, which can be achieved by applying Algorithm 8 to* $xyR^{-1} \mod m$ *and* $R^2 \mod m$*.*

Multiplication by squaring

Software implementations are always less efficient than dedicated hardware solutions, which is why most smart cards use dedicated hardware in order to speed up cryptographic operations. As a matter of fact, it appears that any software implementation of an efficient modular multiplication algorithms (such as Montgomery, Barrett reduction or row-multiplication and reduction) will perform poorly in the Java Card VM [7, 82, 32, 84, 9].

Nevertheless, by converting modular multiplications to modular exponentiations we can benefit from the fast calculations performed on the cryptographic coprocessor. This method which employs modular squaring to perform modular multiplication, is often referred to as *multiplication by squaring or RSA squaring* [82, 9, 84] and can be implemented either by Equation 4.2 or Equation 4.3.

$$a \cdot b \mod n = \frac{(a+b)^2 - a^2 - b^2}{2} \mod n \tag{4.2}$$

$$a \cdot b \mod n = \frac{(a+b)^2 - (a-b)^2}{4} \mod n \tag{4.3}$$

Both equations compute the result of a multiplication without actually computing this multiplication, differing only in the number and type of operations required. However, as can be seen in Table 4.1, both require other modular operations. Modular exponentiation (see section 4.3.3) is the only operation which can be efficiently computed on the coprocessor. Modular addition and subtraction, on the other hand, must be implemented in software as was already discussed in section 4.1. The remaining operation is division by 2 or by 4, which can be implemented by a right shift by one or two positions, respectively; Algorithm 9 depicts a right shifting algorithm.

Table 4.2 compares the number of operations required for Equations 4.2 and 4.3. Note that the

Algorithm 9 1-bit logical shift (right)

Input: $m = (m_{n-1} \cdots m_0)_b$, $x = (x_{n-1} \cdots x_0)_b$, with $x < m$, m odd
Output: $r = x \gg 1 \mod m$
1: if($x_0(0) \neq 0$)then
2: $x \leftarrow x + m$;
3: end if
4: for i=0 to n-1 do
5: $z_i \leftarrow x_{i+1}(0) \,||\, x_i((\log_2 b) - 1 \cdots 1)$;
6: end for
7: return(r);

Table 4.1: Multiplication by squaring

Operation	Algorithm
modular addition	3
modular subtraction	4
modular exponentiation	see section 4.3
division by 2^k	9

number of modular additions varies depending on the result of $((a+b)^2 - a^2 - b^2)$; if its least significant bit equals one, we must add the modulus to the result prior to shifting, otherwise we would be rounding down the result. In the case of Equation 4.3 we might need two extra additions, since we perform two shifts. Depending on the performance of the cryptographic coprocessor on a given card, it is possible to decide which one will be faster. However, due to the significant overhead caused by the JVM, Equation 4.2 is likely to outperform Equation 4.3 as most cards will perform a modular squaring faster than a modular right shift [82].

Table 4.2: comparison of the number of operations needed to perform modular multiplication

	number of executions	
Operation	equation 4.2	equation 4.3
modular addition/subtraction	3 to 4	3 to 5
modular exponentiation	3	2
1-bit modular right shift	1	2

Without prior knowledge of the reduction algorithm used by the cryptographic coprocessor, we cannot determine the computational efficiency of this method.

Comparison of the multiplication algorithms: Table 4.3 shows the theoretical number of multiplications and divisions required for the reduction operation only: they do not include the multiplications and divisions of the precalculation, any transformation or postcalcula-

Algorithm 10 squaring multiplication

```
Input: a = (a_{k-1} ··· a_0)_b, b = (b_{k-1} ··· b_0) and
n = (n_{k-1} ··· n_0)_b, where (a,b)<n
Output: res = a · b mod n
 1: res ← (a + b) mod n; % Algorithm 3
 2: res ← res^2 mod n; % RSA API
 3: aux ← a^2 mod n; % RSA API
 4: res ← (res − aux) mod n; % Algorithm 4
 5: aux ← b^2 mod n; % RSA API
 6: res ← (res − aux) mod n; % Algorithm 4
 7: if(LSB(res)=1) res ← res + n;
 8: res ← shiftRight(res);
 9: return(res);
```

tion [13]. The results refers to the reduction of a $2k$-digit number with a k-digit modulus m. The performance of the algorithms is attributed to the multiplications and divisions required, which are the most time consuming operations in the inner loop of the algorithms.

Table 4.3: Complexity of reduction algorithms in reducing a $2k$-digit number

Algorithm	Classical	Barrett	Montgomery
Multiplications	$k(k+2.5)$	$k(k+4)$	$k(k+1)$
Divisions	k	0	0
Precalculation	Normalization	$b^{2k}\, div\, m$	$-m_0^{-1}\, mod\, b$
Arg. transformation	None	None	m-residue
Postcalculation	Unormalization	None	Reduction
Restrictions	None	$x < b^{2k}$	$x < mb^k$

If only the reduction operation is considered, Montgomery's algorithm is the fastest. Nevertheless, the algorithms are quite close to each other in performance [13]. In their most naive implementation, both have a runtime of $\mathcal{O}(n^2)$, which is due to the use of Schoolbook Multiplication.

In order to improve performance of these modular reduction algorithms, enhancements to the original algorithms have been proposed, *e.g.* [44, 36, 43].

4.3 Large Number Exponentiation

Modular exponentiation is not only widely used in public key cryptosystems, but it is also the most computationally expensive modular operation. This is particularly true for smart cards,

where we have to consider a variety of factors such as memory availability, processing power or the existence of dedicated hardware for cryptographic operations.

In this section we first present our software implementation of the modular technique due to Montgomery. The choice is motivated by the fact that, for general modular exponentiation, the Montgomery's algorithm appears to have the best performance [13, 78]. However, even though this algorithm calculates modular exponentiation efficiently, the implementation in Java might not be fast enough for practical use. Therefore, we also show how the Java Card cryptographic library can be used to speed up this operation.

4.3.1 Montgomery exponentiation

Montgomery exponentiation is one of the most implemented techniques to compute $x^e \mod m$. It combines Montgomery multiplication (Algorithm 8) with the *right-to-left binary exponentiation* algorithm [5, 21] to give a Montgomery exponentiation algorithm (Algorithm 11).

Algorithm 11 Montgomery exponentiation

Input: $m = (m_{l-1} \cdots m_0)_b$, $R = b^l$, $m' = -m^{-1} \mod b$,
$e = (e_t \cdots e_0)_2$ with $e_t = 1$, and an integer x, $1 \leq x < m$.
$R \mod m$ and $R^2 \mod m$ may be provided as inputs.
Output: $A = (a_n \cdots a_0)_b = x^e \mod m$.
1: $\tilde{x} \leftarrow Mont(x, R^2 \mod m)$; % Algorithm 8
2: $A \leftarrow R \mod m$;
3: for i=t to 0 do
4: $\quad A \leftarrow Mont(A, A)$;
5: $\quad$ If $e_i \neq 0$ then
6: $\quad\quad A \leftarrow Mont(A, \tilde{x})$;
7: $\quad$ end if
8: end for
9: $A \leftarrow Mont(A, 1)$;
10: return A

Note that, aside from m', Montgomery exponentiation additionally requires pre-computation of the constants $R \mod m$ and $R^2 \mod m$. In any case, when the modulus changes infrequently these pre-computations are essentially free. For instance, these parameters could be computed externally, during the personalization phase of the card, where resource limitations are not a problem [12].

As we have already mentioned, our implementation of $x^e \mod m$ uses Montgomery exponentiation with the standard square-and-multiply algorithm. This method could benefit from the *p-ary* generalization [45, 21], in which instead of scanning a single bit of the exponent at a time, groups of p bits are scanned. The powerings and subsequent multiplications are performed according to a table of preprocessed values, which stores the powers of x. As the value of p increases, so does the average number of modular multiplications decreases [13]. However, this also increases the memory required to store the table of powers, which makes large values of p prohibitive for smart cards.

Computational efficiency of Montgomery exponentiation: The expected number of single-precision multiplications to compute $x^e \mod m$ by Algorithm 11 is $3l(l+1)(t+1)$, where t is the number of bits of the exponent and l the number of digits, base b of the modulus [5].

4.3.2 Chinese Remainder Theorem

The Chinese remainder theorem (CRT) is used to speed up modulo computations, thereby decreasing the computation time of private key operations in RSA by a factor of approximately four [33, 62, 5].

If the integers $n_1, n_2, \cdots, n_k$ are pairwise relatively prime, then the system of simultaneous congruences

$$\begin{cases} x \equiv & a_1 \mod n_1 \\ & \ldots \\ x \equiv & a_k \mod n_k \end{cases}$$

has a unique solution x, such that $0 \leq x < n = n_1 n_2 \cdots n_k$.

If the system of the linear congruences is soluble, then its solution x can be calculated as

$$x \equiv \sum_{i=1}^{k} a_i N_i N_i^{'} (mod\, n) \tag{4.4}$$

where $n = n_1 n_2 \cdots n_k$, $N_i = n/n_i$, $N_i{'} = N_i^{-1} (mod\, n_i)$, for $i = 1, 2, \cdots, k$.

The proof of CRT is available in most number theory books, e.g. [91].

The algorithm which is generally used to solve the Chinese remainder problem is Garner's algorithm for CRT, which is particularly efficient when dealing with large integers (*e.g.*, RSA). Given that the prime factors of the modulo, $n = pq$, are known, $x^d \mod n$ can be computed by first computing $x^{d_p} \mod p$ and $x^{d_q} \mod q$ (where $d_p = d \mod (p-1)$ and $d_q = d \mod (q-1)$), and then use Garner's algorithm to construct $x^d \mod pq$. Although this procedure takes two exponentiations, each is considerably more efficient because the modulus are smaller.

For further details on the Garner's algorithm, please refer to Chapter 14.5 of [5].

Example 4.3.1 *We can use the CRT to compute $m = c^d \mod n$ more efficiently.*
Suppose $p = 7$, $q = 11$, $e = 19$ and $d = e^{-1} \mod (p-1)(q-1) = 19$.
We begin by precomputing :

$$dP = e^{-1} \mod (p-1) = d \mod (p-1) = 19 \mod 6 = 1$$
$$dQ = e^{-1} \mod (q-1) = d \mod (q-1) = 19 \mod 10 = 9$$
$$qInv = q^{-1} \mod p = 11^{-1} \mod 7 = 2$$

Then we store our private key as the quintuple (p, q, dP, dQ, qInv).
To compute $s = m^d \mod pq$ we use Garner's algorithm:

$$s_1 = m^{dP} \mod p = 50^1 \mod 7 = 1$$
$$s_2 = m^{dQ} \mod q = 50^9 \mod 11 = 2$$
$$h = qInv \cdot (s_1 - s_2) \mod p = 2(1-2) \mod 7 = 5$$
$$s = s_2 + h \cdot q = 2 + 5 \cdot 11 = 57.$$

$s = m^d \mod pq = 50^{19} \mod 77 = 57$ ■

4.3.3 Java Card's RSA and exponentiation

Even if Montgomery's algorithm is an efficient algorithm for modular exponentiation, in the context of smart cards this does not hold true. In fact, it has already showed that even a highly optimised assembly implementation of a multi-exponentiation algorithm does not execute in reasonable time on an 8-bit microcontroller [7]. In addition, these complex mathematical operation will have to be executed in the Java Card VM, which has been proved to be inherently inefficient [82, 84, 9].

Based on the above, it becomes clear that the solution may lie in using the cryptographic coprocessor to speed up the computations. However, on currently available Java cards, it is only possible to access the cryptographic coprocessor through the Java Card application programming interface (API). The problem is that the API does not provide any support for large number arithmetic even though RSA, Diffie-Hellman and DSA are available through the API and internally perform these calculations.

The only exception is the *BigNumber* class, in the optional package *javacardx.framework.math*, which is only available in version 2.2.2 of the Java-Card API. Besides being an optional package and, therefore, rarely implemented, it only supports non-modular addition, subtraction and multiplication.

Nonetheless, the Java Card API does include support for RSA, where an encryption of the message m to the ciphertext c is executed by calculating $c = m^e \, mod \, n$. This is the same as calculating a modular exponentiation, which means that by setting all the values involved the RSA cipher can be tricked into performing modular exponentiation.

The RSA key types which allow us to set the modulus and the exponent are the *RSAPublicKey* and the *RSAPrivateKey*. The *RSAPrivateCrtKey* achieves faster decryption through the use of the Chinese Remainder Theorem (CRT), however to set up such key we would need to know the factorization of the modulus. Among the several available modes of operation for the RSA on the Java Card platform, only one is of interest to us; the *RSA_NOPAD* mode is the only mode which ensures that the input value is not padded. Therefore, the pair *(key_type,operation_mode)* that we are interested in is the *(RSAPrivateKey/RSAPublicKey, RSA_NOPAD)*; an example on how to perform modular exponentiation with the RSA cipher is depicted in Appendix A.

To ensure the correct computation of the modular exponentiation, a few restrictions must apply, though :

- The modulus must be at least 64 bytes long and have a maximum length of 244 bytes. It must also be odd, divisible by 4, and its most significant byte (MSB) must be nonzero (otherwise the bytes in the result array will be "shifted"').
- PublicKey size must be at most 10 bytes long, while private keys may be up to 244 bytes long. These restriction prevent us, for example, from tricking the coprocessor into

performing modular exponentiation or modular multiplications with a modulus smaller than 64 bytes.

- *RSA_NOPAD* requires that input data should be exactly the same size of the modulus (*i.e.*, the input array must have the same length as the modulus array; nevertheless, we can pad the MSBs of the input array with zeros).
- There is a limit for RSA public and private keys on how many times we can assign a cryptographic key to an RSA key object. The limit is different for public and private keys; while the former has a higher limit, the latter's is rather small. In Table 4.4 we present the upper bound for RSA Public Keys. This restriction appears to be a security mechanism specific to the card, as such feature has not been documented in similar projects that rely on JCOP Java Cards [82, 84, 9].

Table 4.4: RSA public key limitations

Key (bits)	512	768	1024	1280	1536	1792	2048
RSAPublicKey	519	392	314	260	221	192	169

RSA keys are meant to change infrequently. However, to design a general modular exponentiation routine, we need to be able to do so in order to change both exponent and modulus. A general modular exponentiation routine would do the following: at applet initialization a *RSAPublicKey* object would be created and assigned to an RSAPublicKey reference. Afterwards, during the applet's lifetime, whenever an exponentiation would be computed, another Key would be created using the *KeyBuilder.buildKey()* method. Afterwards, the modulus and exponent of the key would be set up and fed to the RSA cipher in order to compute the modular exponentiation.

However, with such restriction we cannot design a general modular exponentiation algorithm using RSA, without, at some point, having to re-issue the applet.

In addition, RSA keys are persistent objects, which would not only slow down the exponentiation routine, but would also have to rely on the efficiency of the garbage collector (if available).

Chapter 5

Implementing Authentication Protocols

> ”You can't mean this little hole! It isn't a window; it's a hole in my bed.”
> ”I did not say it was a window: I said it was my window.”
> ”But it can't be a window, because windows are holes to see out of.”
> ”Well, that's just what I made this window for.”
> ”But you are outside: you can't want a window.”
>
> ---
> At the Back of the North Wind
> GEORGE MACDONALD

In this chapter we address implementation issues and choices regarding the authentication schemes presented in chapter 2. Two classes of key-based encryption algorithms are implemented: symmetric and asymmetric ciphers. In the former, we consider both block and stream ciphers.

The most common operations performed by symmetric ciphers are bitwise operations. However, the architecture of the Java Card platform introduces extra overhead, which must be carefully addressed in order to minimize the performance penalizations. Public-key cryptosystems have the advantage of not requiring the entities to share a private key; however, this comes with a price: more complex and expensive computations are involved. When these cryptosystems are embedded in low resource devices, such as smart cards, an efficient implementation of modular arithmetic is fundamental, as we have already discussed in Chapter 4.

In sections 5.1, 5.2 and 5.3, we start by describing our implementation of the challenge-response protocols for AES, eLoBa and RSA, respectively. Sections 5.4.1 and 5.4.2 then descrive the

implementation of the Feige-Fiat-Shamir and Guillou-Quisquater identity protocols, respectively.

5.1 Block cipher AES

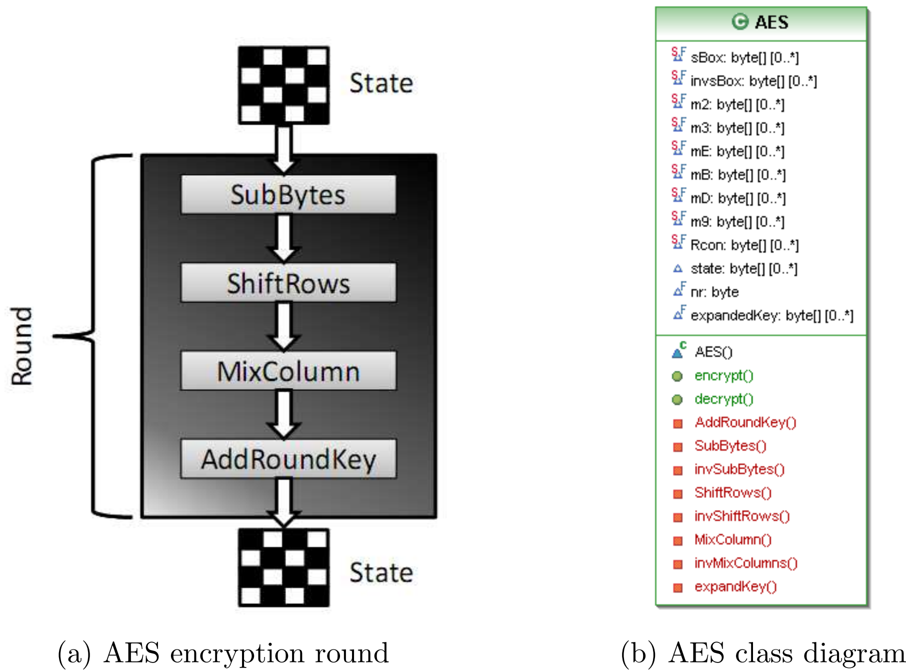
In this section we describe the main classes implemented for AES cryptosystem.

5.1.1 Architecture

The AES algorithm is iterative and every round operates on an entire data block called *State.* The input to the encryption and decryption algorithms is fixed-sized block (usually 128 bits, but AES is easily adaptable for a multiple 32-bit size block, such as 192 and 256 bits). Data is processed as a square matrix of bytes. However, the Java Card specifications do not support multi-dimensional arrays, which forces us to represent the state matrix as an array of $128/8 = 16$ Bytes. In order to speed up the transformations performed upon the state variable, we store the data as a transient byte array.

A working implementation of AES must support all the four transformations : *substitution, permutation, mixing and key adding.* The four main methods that are used for the encryption process are the *SubBytes, ShiftRows, MixColumns and AddRoundKey* functions; these operations are depicted in Figure 5.1(a). As for decryption, the sequence of method invocation is reversed, except for the AddRoundKey, whose inverse transformation is identical to the forward transformation, because the Xor operation is its own inverse. The AES Java class diagram depicted in Figure 5.1(b), where the algorithm's parameters and functions' names are according to the FIPS-197 recommendation [25].

Our implementation of AES supports three different key lengths: 128, 192 and 256 bits. The cipher key is loaded into the card at applet instantiation time and stored in static memory. The AES class constructor is invoked when the AES cipher is created at applet instantiation time; it takes a key as parameter and performs all the necessary memory allocation, key expansion and set up of the algorithm.



(a) AES encryption round (b) AES class diagram

Figure 5.1: AES round methods

SubBytes

The SubBytes transformation performs a simple byte substitution on each byte of the State using a substitution table, the substitution box (sBox), which contain the permutations of all 256 8-bit values. These boxes are constructed using defined transformation of values in $GF(2^8)$ and are loaded into the card as static final byte arrays (EEPROM) at applet instantiation time. Each of this tables is stored in the EEPROM and requires 256 bytes of memory.

The implementation of the inverted SubBytes function is straightforward, as is it processed exactly in the same way as the SubBytes operation, with the exception that the inverse substitution Box (invsBox) table is used. Consequently, a total of 512 bytes are needed for storing the S-BOX and the inverted S-BOX table, which is almost negligible for modern smart cards. The S-BOX and inverted S-BOX tables are depicted in Appendix B.1.

ShiftRows

The ShiftRows transformation consists of circular byte shifts, where each row is shifted over a different number of positions. The inverse shift row transformation (invShiftRows) performs the circular shifts in the opposite direction.

MixColumns

MixColumns is the most expensive operation, since it involves matrix multiplication in GF(2^8). GF(2^8) multiplication is defined with a carefully selected primitive polynomial $x^8 + x^4 + x^3 + x + 1$ to speed up computation.

In practice, Mix Columns can be implemented by expressing the transformation of each column as four equations to compute the new bytes for that column [79]. The computation only involves shifts, XORs and conditional XORs (for the modulo reduction). However, the decryption is slower due to the computation requiring the use of the inverse matrix, which has larger coefficients.

We can achieve a significant speed up by using lookup tables with all the precomputed multiplications in $GF(2^8)$. The additional 1536 bytes of EEPROM memory required to store the lookup tables does not comprise a problem, taking into account the considerable amount of memory currently available in smart cards. Lookup tables not only improve the performance of the AES algorithm but also make it more secure by making it less prone to timing and power attacks [41, 81, 10]. The multiplication tables are shown in Appendix B.2.

AddRoundKey

The AddRound function is straightforward: the 128 bits of State are bitwise XORed with the 128 bits of the round key.

KeyExpansion

The AES key expansion algorithm takes as input the key and produces the expanded key array, which ranges from 176 bytes to 240 bytes, depending on the size of the input key. The expanded key could be expanded each time it is being used, or be expanded once and stored in the static memory. We have followed the latter approach, since storing the key in static memory has a low memory footprint and decreases the time needed to perform either encryption or decryption.

The round constant array ($Rcon$) contains the values given by x^{i-1}, with x^{i-1} being powers of x in the field $GF(2^8)$. This array is stored in persistent memory and is depicted in Table 5.1.

Table 5.1: Rcon array

i	0	1	2	3	4	5	6	7	8	9
x^i	01	02	04	08	10	20	40	80	1b	36

5.1.2 Challenge-Response with AES

The challenge-response scheme with the AES cryptosystem, see section 2.4.2, is carried out between the host application (residing in the workstation) and the applet (residing in the smart card). We could have adapted our implementation of the AES cipher for the Windows/PC workstation, however we have decided to use Java Security Extension (JSE) to test interoperability between the 2 implementations. Moreover, since our implementation uses 8-bit arithmetic it would most certainly run slower than the JSE implementation.

The challenge-response authentication with AES proceeds as follows: in unilateral authentication the verifier generates a 16 byte random number for the claimant to encrypt with the shared secret key, whereas in mutual authentication each entity generates an 8 byte number, creating a 16 byte block which will be used as challenge. No identifier was used; although this approach is security weak - for example may suffer man-in-the-middle attacks, our focus is on the performance of the implementation of the cipher. Nevertheless, extending the protocol to incorporate the identifier is rather straightforward.

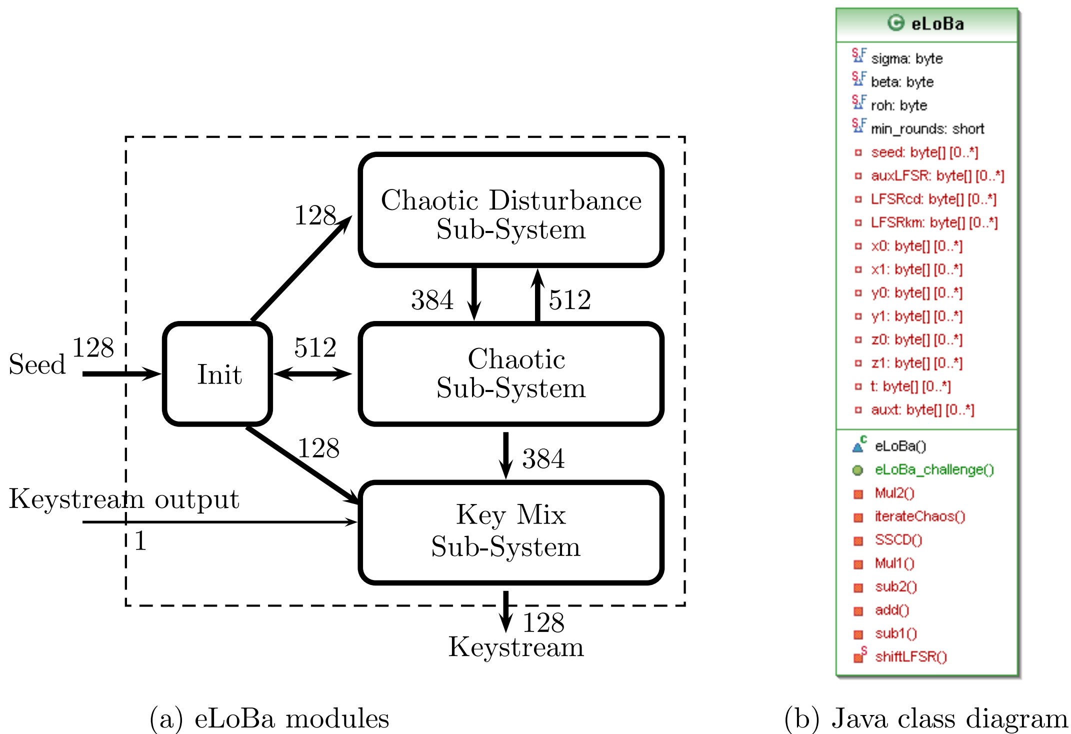
In order to communicate with the applet, we have defined four APDU commands: *UNILATERAL_AUTH_AES_CE, MUTUAL_AUTH_AES_CE*, *UNILATERAL_AUTH_AES_PJ* and *MUTUAL_AUTH_AES_PJ*. The first two commands request a unilateral and a mutual authentication, respectively, using the *coprocessor-enabled (CE) Java Card cryptographic API*, while the remaining two achieve the same functionality using our implementation of AES (fig. 5.1(b)).

5.2 Stream cipher eLoBa

In this section we describe the main classes implemented for eLoBa cryptosystem.

5.2.1 Architecture

The eLoBa cipher has a modular architecture comprised of four Sub-Systems, as shown in Figure 5.2(a). Relating to the overall functionality of eLoBa, the cipher is initialized by a 128 bit secret value, or *Seed*, which is processed through a sequence of steps. After system initialization, the complete process of key generation starts with the iteration of the *Chaotic Sub-System* after which the *Key Mix Sub-System* generates two 128 bit keys. The output of the *Chaotic Sub-System* gives also input to the *Chaotic Disturbance Sub-System* whose purpose is to provide feedback disturbance to the next iteration of the Chaotic System. Figure 5.2(b) shows the definition (attributes and methods) of the eLoBa class.



(a) eLoBa modules (b) Java class diagram

Figure 5.2: eLoBa architecture

The most common operations are straightforward: *bytewise XORs, shifts,* and *byte swapping.* This greatly simplifies the implementation of the system, as both the chaotic and key mix modules solely rely on these simple operations. Another such example are the 128-bit full cycle linear feedback shift registers (LFSRs), which are used in the initialization module as well as the Chaotic disturbance and Key mix sub-systems. These LFSRs are implemented in software by the *shiftLFSR(byte[] lfsr)* method, with a primitive polynomial given by the coefficients (128,7,2,1,0).

The most computationally demanding operations required by the system are, without doubt, the integer modular arithmetic performed by the Chaotic module. Modular addition and subtraction have already been discussed in section 4.1.2, and modular multiplication on section 4.2. Nonetheless, eLoBa works in 128 bit arithmetic, *i.e.,* it has a fixed modulus, which means that we will only need to consider the 16 least significant bytes of the result. Therefore, instead of the modular operations we use the classic methods. An example for modular multiplication is given in Appendix C.

An interesting alternative would be to use cards implementing version 2.2.2 or 3.0.1 of the Java Card standard, since it includes support for non-modular addition, subtraction and multiplication through the BigNumber class.

Chaotic Sub-System

The Chaotic Sub-System implements the Lorenz System of Equations and is parametrized for chaotic behaviour. Its behaviour is mathematically represented by Equation 5.1, where X, Y and Z are the coordinates of the chaotic system and Δt the integration step. The system parameters, obtained from [77], are fixed to $\sigma = 10$, $\rho = 28$, $\beta = 3$ and loaded into the card at applet instantiation time. Other values can be used as long as the system presents chaotic behaviour, which is achieved for $\sigma > \beta + 1$, $\rho > 0$ and $\rho > \frac{\sigma(\sigma+\beta+3)}{\sigma-\beta-1}$.

$$\begin{cases} X_{i+1} &= X_i + \Delta i(\sigma(y - x)) \\ Y_{i+1} &= Y_i + \Delta i(\rho x - y - xz) \\ Z_{i+1} &= Z_i + \Delta i(xy - \beta z) \end{cases} \tag{5.1}$$

The eLoBa cipher uses the Lorenz system of equations running in integer algebra under a 128 bit modular arithmetic. Therefore, each iteration to the Chaotic Sub-System requires eight 128-bit modular multiplications. Each modular multiplication is computed bytewise, hence each iteration requires a total of 16 bytewise multiplications.

It is still possible to reduce the calculation time by reducing the range of the integration step variable Δt [77]. In the reduction of the integration step, we substitute three 128 bit by 128 bit multiplications by three 8 bit by 128 bit multiplications, which has great impact in the performance. The reduced integration step is obtained by selecting a single byte from that

variable, according to the value of its own four bits of highest level. However, to keep the level of security, the integration step variable must keep its storage space of 128 bits.

Another feature of eLoBa's modular arithmetic is that, even though multiplications may produce a result of more than 16 bytes, only the 16 least significant bytes are kept and the remaining bytes are ignored. This greatly improves performance as it reduces both the memory and computation requirements. For instance, while a 16 bytes by 16 bytes multiplication requires 136 bitwise multiplications and 240 bitwise additions, a 16 bytes by 1 byte multiplication, requires only 16 bitwise multiplications and 15 bitwise additions.

The chaotic system is iterated through the *iterateChaos()* method; the *Add()*, *Sub1()* and *Mul1()* provide addition, subtraction and multiplication, respectively, for two 128 bits unsigned integers in the byte array format, while *Sub2()* and *Mul2()* provide subtraction and multiplication for the operations when one operand has 128 bits and the other 8 bits.

Chaotic Disturbance Sub-System

The Chaotic Disturbance module is responsible for the introduction of orbit changes in the Chaotic Sub-System, avoiding it to converge to one single value or enter in short-cycle length orbits. Its behaviour is implemented by the *SSCD()* method and mathematically represented by Equation 5.2. The $LFSR_{CD}$ is used to XOR its present state with the resulting Y and Z coordinates from the Chaotic Sub-System. The value of the integration step Δt is also changed through the XOR of its present value with the value of the present X coordinate that comes from the Chaotic Sub-System. The resulting Y_{k+1}, Z_{k+1} and $(\Delta t)_{k+1}$ values will be used as feedback to the Chaotic SubSystem and update the respective variables to be used in the next iteration of the chaotic system. After the new value for iteration step is computed, the reduced integration step byte is updated as well.

$$\begin{cases} Y_{k+1} &= Y_{k+1} \oplus LFSR_{CD_i} \\ Z_{k+1} &= Z_{k+1} \oplus LFSR_{CD_{i+1}} \\ \Delta t &= \Delta t \oplus x_{k+1} \end{cases} \tag{5.2}$$

Key Mix Sub-System

The Key Mix Sub-System receives as input a triple of coordinates *(x,y,z)* produced by the Chaotic Sub-System and generates as output two 128 bits keys, as shown in Equation 5.3. The purpose of equation 5.3 is to conceal the chaotic state to the outside world.

Regarding the notation, x_a means byte of order a in coordinate x; $y[a:b]$ corresponds to the bits a to b from the y coordinate; the ith state of the LFSR is represented by $LFSR_i$ and the symbol $A||B$ corresponds to the concatenation of A with B.

In order to generate the keys, bits from the three coordinates are concatenated into a block of 128 bits which is to be XORed with the $LFSR_{CD}$; the second key, on the other hand, is generated after iterating the $LFSR_{CD}$.

$$\begin{cases} Key_i & = [x_3 \,||\, y[1:3] \,||\, x_7 \,||\, y[5:7] \,||\, x_{11} \,||\, y[9:11] \,||\, x_{15} \,||\, y[13:15]] \oplus LFSR_{KM_i} \\ Key_{i+1} & = [x_2 \,||\, z[1:3] \,||\, x_6 \,||\, z[5:7] \,||\, x_{10} \,||\, z[9:11] \,||\, x_{14} \,||\, z[13:15]] \oplus LFSR_{KMD_{i+1}} \end{cases} \quad (5.3)$$

Initialization

The value of the *Seed* is processed by the initialization module, in order to initialize all the parameters of the three sub-systems:

1. Chaotic Sub-System : the x, y and z coordinates and integration step Δt
2. Chaotic Disturbance Sub-System : the $LFSR_{CD}$
3. Key Mix Sub-System: the $LFSR_{KM}$

The initialization of the Chaotic Sub-System can be summarized by Equation 5.4, where $LFSR_i$ denotes the state of the LFSR after the ith iteration, x_0 is the initial value of the x

coordinate and *F1*, *F2* and *F3* are functions that perform byte circular shifts.

$$\begin{cases} LFSR = & SEED \\ x_0 = & LFSR_0 \\ y_0 = & F1(LFSR_1) \\ z_0 = & F2(LFSR_2) \\ \Delta t_0 = & F3(LFSR_3) \end{cases} \tag{5.4}$$

The auxiliary LFSR, $auxLFSR$, as well as the x coordinate, are set to the value of the Seed, while the values of y_0, z_0 and Δt_0 are set to the value of the $auxLFSR$ after each of three iterations. Thereafter, the values of y, z and Δt are fed to the *F1*, *F2* and *F3* functions, respectively, thereby concluding the initialization of the Chaotic Sub-System.

The initialization of the remaining two Sub-Systems is straightforward: first, we initialize the Chaotic Disturbance Sub-System by iterating the Chaotic Sub-System; the resulting three new values for the coordinates, namely x_1, y_1 and z_1, are then XORed according to Equation 5.5.

$$\begin{cases} LFSR_{CD_0}[0:3] = & x_1[4:7] \oplus y_1[8:11] \oplus z_1[12:15] \\ LFSR_{CD_0}[4:7] = & x_1[8:11] \oplus y_1[12:15] \oplus z_1[0:3] \\ LFSR_{CD_0}[8:11] = & x_1[12:15] \oplus y_1[0:3] \oplus z_1[4:7] \\ LFSR_{CD_0}[12:15] & x_1[0:3] \oplus y_1[4:7] \oplus z_1[8:11] \end{cases} \tag{5.5}$$

The initialization of the Key Mix Sub-System and the Chaotic Disturbance Sub-System are similar; first the Chaotic Sub-System is iterated, resulting in three new values for the coordinates, namely x_2, y_2 and z_2. These are to be XORed according to Equation 5.6, thereby concluding system initialization.

$$\begin{cases} LFSR_{KM_0}[0:3] = & x_2[12:15] \oplus y_2[8:11] \oplus z_2[4:7] \\ LFSR_{KM_0}[4:7] = & x_2[0:3] \oplus y_2[12:15] \oplus z_2[8:11] \\ LFSR_{KM_0}[8:11] = & x_2[4:7] \oplus y_2[0:3] \oplus z_2[12:15] \\ LFSR_{KM_0}[12:15] = & x_2[8:11] \oplus y_2[4:7] \oplus z_2[0:3] \end{cases} \tag{5.6}$$

Note that the $LFSR_{CD}$ and the $LFSR_{KM}$ will still be used after the initialization process is complete, while the $auxLFSR$ is only necessary for system initialization. Nonetheless, the transient memory allocated for the $auxLFSR$ is still usable and RAM still remains a scarce resource on smart cards. Therefore, the memory allocated for the $auxLFSR$ in the initialization module is afterwards used for storing intermediary results.

5.2.2 Challenge-response with eLoBa

The overall functionality of our authentication scheme with the eLoBa cipher, depicted in figure 5.3 using SDL language [42], is divided in an initialization step and a minimum of 16 and up to 31 iterations to the Chaotic Sub-System. The 128 bit challenge is split between the number of rounds and the output key bit, while the remaining bits are XORed with the system's Seed and used to initialize the system. The process of key generation starts with the iteration of the Chaotic Sub-System after which the Key Mix Sub-System could produce the first pair of keys. However, since we are only interested in the system's internal state after a certain number of rounds, we can skip the key generation step until we reach the last round. In the last round we iterate the chaotic sub-system and the Key mix subsystem, but we eliminate the Chaotic Disturbance Sub-System iteration since it will not be necessary to further iterate the system. The process of key generation is carried out by the Key-Mix Sub-System, thereby constructing either key1 or key2, according to the parity of the output key bit. The generated key is then sent to the verifier in response to the challenge. The fact that we only need to generate the requested output key allows us to avoid unnecessary computations.

5.3 RSA

In this section we describe the components implemented for RSA cryptosystem.

5.3.1 Arquitecture

The RSA cryptosystem is a public-key cryptosystem that offers both encryption and digital signatures (for authentication). It uses a public modulus n, product of two large prime

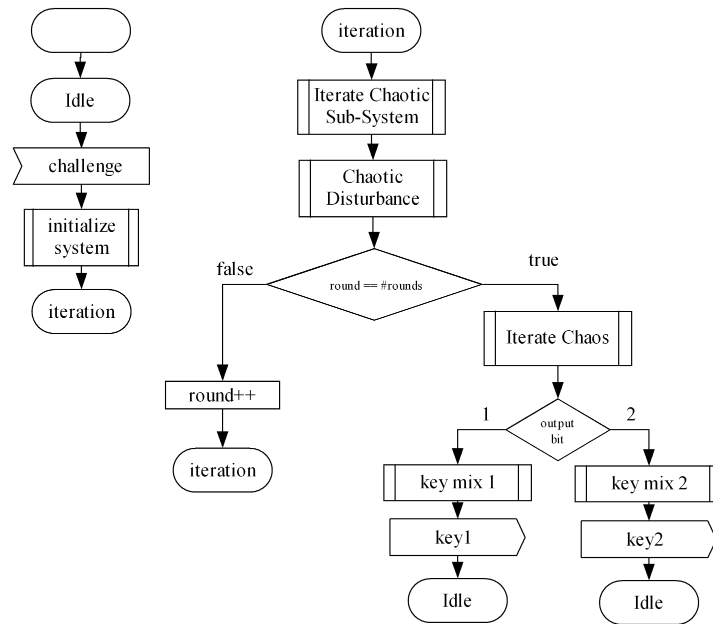


Figure 5.3: SDL description of eLoBa functionality

numbers p and q, a public exponent e, less than n and relatively prime to $\varphi(n) = (p-1)\cdot(q-1)$, and a private exponent $d = e^{-1}(mod\,\varphi(n))$. The values e and d are called the public and private exponents, respectively. The public key is the pair (n, e), while (n, d) is the private key. The factors p and q that constitute n must be kept secret, and allow us to use the Chinese Remainder Theorem(CRT) to speed up decryption/signing – see section 4.3.2. Additionally, when using Montgomery's algorithm for exponentiation, pre-computation and storage of m', R mod m and $R^2 \mod m$ is also required — see Algorithm 11. On the other hand, if Barrett's algorithm is used, only the pre-computation of $\mu = \lfloor b^{2k}/m \rfloor$ is required — see Algorithm 7.

The keys and other required parameters are computed externally, in the workstation with the *java.math.BigInteger* class, and loaded into the card at applet instantiation time. Once the RSA cryptosystem is set up, i.e., the modulus and the private and public exponents are determined and the public components have been published, both the operation of signing and verification can be performed with the computation of a modular exponentiation, $M^e\,(mod\,n)$, which we have already discussed in Chapter 4. The digital signature is created by exponentiating: $s = m^d \mod n$, where d and n are the signatory's private key, and m the message to be signed. The validation/verification of the signature is performed by $m = s^e \mod n$, where e and n are the signatory's public key.

5.3.2 Challenge-response with RSA

We compare our own implementation of RSA, which uses Barrett's exponentiation, against the one available in the Java Card crypto APIs, using CRT multiplication. No padding was added to the original messages, i.e., message length is equal to N size.
Our unilateral challenge-response protocol is based on RSA decryption and implements a simplified version of the protocol described in section 2.4.3. Figure 5.4 depicts RSA unilateral authentication.

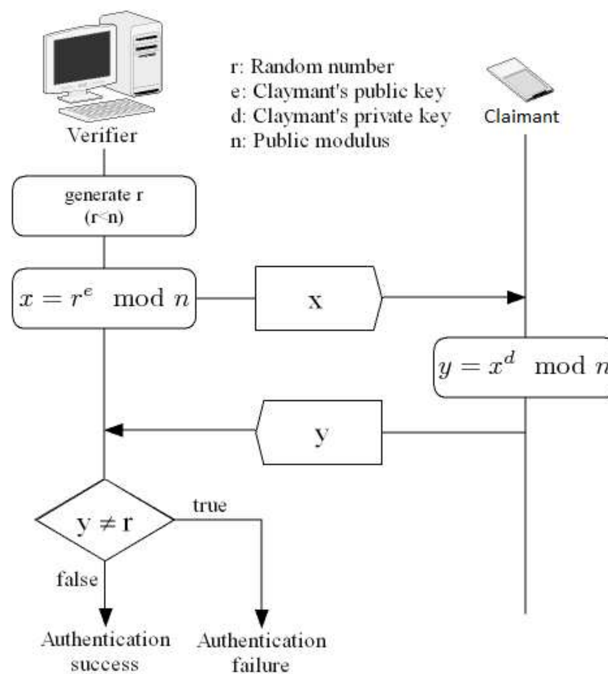


Figure 5.4: RSA-based challenge-response authentication

Note that we have omitted some features from the original protocol, namely the claimant's identifier and the message digest; we did so in order to focus on the performance of the cipher rather than on the protocol itself. Nevertheless, this approach provides us with the lower bound for the execution time of the complete protocol, and also allows a fair comparison between the authentications with AES and eLoBa.

Despite the fact that this basic scheme is not very secure, our implementation can be easily extended given that message digests (*e.g.*, SHA-1 [63]) are directly available on current smart cards.

5.4 Zero-knowledge protocols

In this section we describe the implementation of two zero-knowledge protocols: Feige-Fiat-Shamir, in section 5.4.1, and Guillou-Quisquater, in section 5.4.2.

5.4.1 Feige-Fiat-Shamir

The Feige-Fiat-Shamir (FFS) authentication scheme, discussed in section 2.5.2, uses a public-private key pair and is based on the difficulty of computing square roots modulo composite numbers. The claimant's public and private keys are generated in the workstation and loaded into the card at instantiation time. The protocol is repeated t times, where t=8 and the challenge is composed of k bits, where k=9.

In contrast to RSA, FFS is computationally much lighter; the only computations involved are modular multiplications, whereas RSA uses modular exponentiation. However the difficulties of implementing FFS on a Smart card are the limited computational resources and the inefficiency of Java (byte) code execution, as explained in chapter 4. Therefore, in order to achieve an acceptable performance we should rely as much as possible on the built-in cryptographic operations, which are executed on the dedicated coprocessor.

Table 5.2 depicts the mathematical operations needed for the FFS protocol. Random number generation is supported by the Java Card *RandomData* class and modular arithmetic has already been discussed on chapter 4. Naturally, our choice falls on the Java Card RSA interface, which benefit from hardware acceleration: modular multiplication is implemented through *multiplication by squaring* (section 4.2.3) and modular squaring is easily achieved through RSA exponentiation (section 4.3.3) with a fixed exponent 2.

A drawback of approach is that, even though modular multiplication partly executes on the coprocessor, it still remains the bottleneck in this protocol; *multiplication by squaring* involves modular exponentiation (for squaring) but also modular addition and subtraction, which must be computed on the JVM. In theory, FFS only requires a small fraction of the computations required by RSA, however, in practice, it is inherently less efficient than RSA because the latter can fully execute on the cryptographic coprocessor.

For comparison purposes, we also implement FFS without the coprocessor, using Barrett's algorithm for modular multiplication.

Table 5.2: Operations required by the FFS protocol

Mathematical operation	number of operations
Random number generation: r	1
Modular squaring: $x = r^2 \, mod \, n$	1
Modular multiplication: $y = (r \cdot s_1^{c_1} \cdots s_{k-1}^{c_{k-1}} \cdot s_k^{c_k}) \, mod \, n$	up to $k+1$

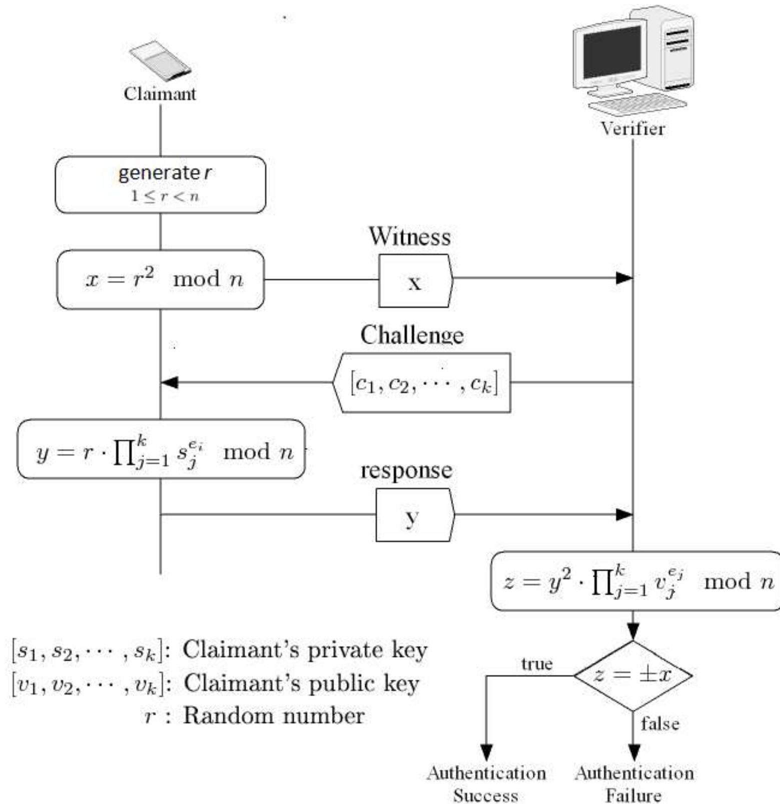


Figure 5.5: Feige-Fiat-Shamir authentication scheme

5.4.2 Guillou-Quisquater

As was already mentioned in section 2.5.3, the Guillou-Quisquater (GQ) protocol is an extension of the Fiat-Shamir protocol that limits the number of rounds, computations and the amount of memory requirements for user secrets. This protocol relies in the difficulty of extracting v^{th} root module n, where v is the security parameter and t determines the number of executions of the protocol.

No repetition of the procedure is needed as long as the size of the public exponent v is sufficient for the desired level of security. For instance, twenty to thirty bits are enough to ensure a secure remote authentication [34]. Therefore, we have chosen a 32 bit v and $t = 1$ (a single round), since these assure a good compromise between the level of security and the

execution speed. To simplify the implementation, we have also chosen a random value for the signature J_A (without actually computing the hash of an ID) and loaded the certificate $S_A \equiv (J_A)^{-1} mod\, n$ directly into the card.

We perform the system one-time setup and the selection of per-user parameters in the workstation, followed by the personalization of the card and applet instantiation.

As it can be seen from tables 5.2 and 5.3, and figures 5.5 and 5.6, the GQ protocol is very similar to the FFS protocol as it also involves modular multiplication and modular exponentiation. However, on one hand, an execution of the GQ protocol requires two modular exponentiations instead of one modular squaring, but on the other hand, GQ protocol requires only one single modular multiplication instead of $k+1$ multiplications.
Since we can compute modular exponentiations much faster than modular multiplications, thanks to the RSA API, we can expect GQ to outperform FFS which is much penalized by the modular multiplication operation, which must be implemented in software.

Table 5.3: Operations required by the GQ protocol

Mathematical operation	number of operations
Random number generation: r	1
Modular exponentiation: $x = r^v mod\, n$, $y = r \cdot S_A^e\, mod\, n$	2
Modular multiplication: $y = r \cdot S_A^e\, mod\, n$	1

The drawback of this approach are the severe restrictions of the RSA interface, which prevent us from fully exploiting it. For instance, the GQ protocol involves modular exponentiation with a random exponent (the challenge), which requires us to update the RSA keys frequently to obtain exponentiations with different values. Since RSA keys are stored in persistent memory (EEPROM), this will not only lead to premature wear of the memory but also prevent us from rapidly changing the exponent.

More importantly, the most severe limitation comes from the fact that our TOP GX4 card only allows a limited number of updates to the key object. Therefore, the number of modular multiplications our applet can execute before having to be re-installed, is also limited. We are able to maximize the number of updates to the RSA key by using a public key, instead of a private key, to store the 32 bits v. Yet, this hardly a solution to the problem and we have found no way to bypass this.

For comparison purposes, we also implement GQ without the coprocessor, using Barrett's algorithm for modular multiplication and modular exponentiation.

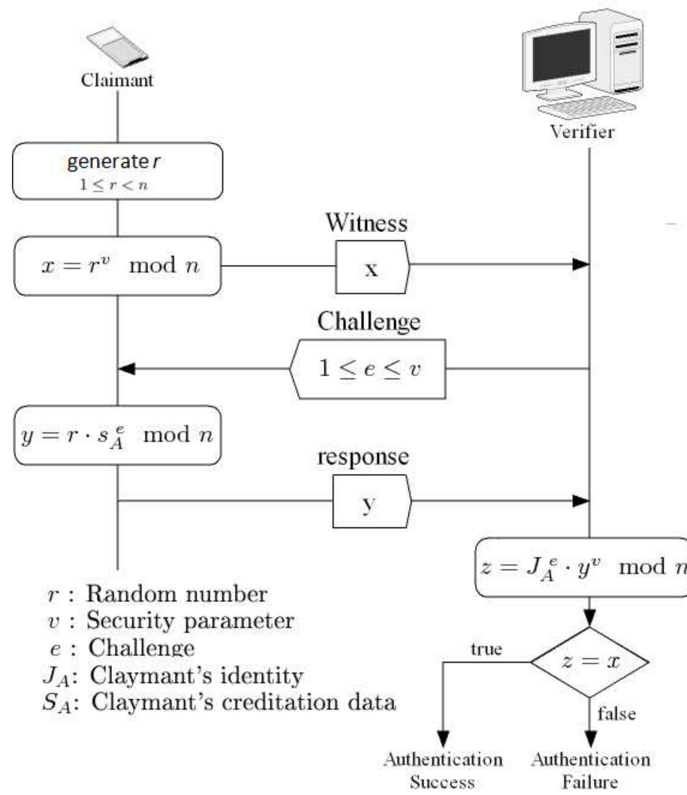


Figure 5.6: Guillou-Quisquater authentication protocol

Chapter 6

Result Analysis, Conclusions and Further Work

> Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.
>
> — Antoine de Saint-Exupéry

To analyse the performance of our implementation of the authentication protocols, we must first address the issue of performance measurement. For the code under analysis to be executed by the smart card's CPU, the command APDU must traverse several layers of software and hardware. To accomplish this, we must devise an adequate test framework to allow us to extract the proper execution time of our application. Unlike other projects [50, 67, 60, 61], our aim is not to provide a complete test framework to measure the performance of Java Card platforms. Therefore, we have decided to adopt a simpler approach, which we believe to be sufficiently accurate for the purpose of this work, and for the magnitude of the time measurements in cause.

In sections 6.1 to 6.2, we describe the configuration of the workspace and propose a general architecture for performance measurement. Performance results are depicted and discussed in section 6.3.

6.1 The development environment

For the development and testing of smart card applications, we adopted for the host computer an Intel Pentium IV 3GHz PC with 1,50 GB of RAM under Windows 7. Several tools are available to develop and load Java Card applets, and, even though there are interesting proprietary tools, we focused only on publicly available and open source software. The development environment has the following configuration:

- A Gemalto TOP GX4 [4] smart card is used in this project. It is Java Card 2.2.1 and Global Platform 2.1.1 compliant and has an approximate available memory size of 68K. Multiple cryptographic algorithms are supported, such as RSA (up to 2048 bit), AES (128, 192 and 256 bits) and SHA1. True random number generation and real garbage collector (JC 2.2.1 specification) are also available.

- Omnikey 3121 USB smart card reader will perform the tasks of a CAD.

- The workstation is a Windows/PC, which is used to develop the Java Card applets as well as running the host applications.

- Java code source file (.java) can be compiled via the Java Development Kit (JDK) into a class file (.class). Sun's Java Card Development Kit (JCDK) version 2.2.1 [1] is used for converting class files into converted applet files (.cap). The JCDK does not provide a visual development environment, therefore, the Eclipse IDE is used for developing Java Card applets through the JCDE [3] plugin, which integrates the functionality of the JCDK. The Eclipse integrated Java Card development Environment is depicted in Figure 6.1.

- In order to deploy the application, the CAP file must be loaded into the smart card, which can be achieved through the open source program GPShell 1.4.2 [30].

- Our implementation of the AES cryptosystem for the Java Card smart card supports three key sizes: 128, 192 and 256 bits. However, the standard Java SDK does not support the 192 and 256 key sizes due to export restriction policies. In order to access all sizes in the challenge-response, Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files need to be downloaded.

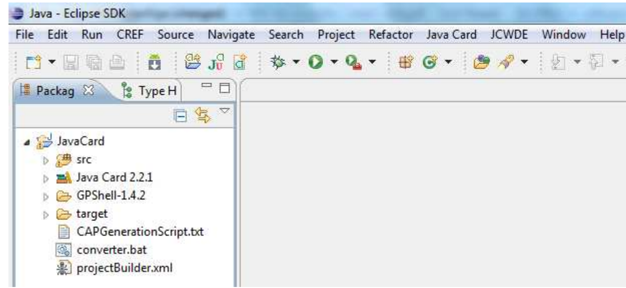


Figure 6.1: Eclipse integrated Java Card development Environment

- The *javax.smartcardio* package was included in Java 6 and allows Java programs to communicate with a Java Card smart card, using ISO/IEC 7816-4 APDUs. With the Java Smart Card I/O API it is also possible to detect card insertion/removal as well as to establish connection with a reader.

Prior to deployment, applications may be tested in the PC/Windows workstation, using the two simulators provided by the JCDK : JCWDE and CREF. Card simulator reveals several advantages, such as speeding up the development process of applets and not wearing out the card. However, these simulators also have several limitations; for instance, several cryptographic algorithms are not or only partially available (e.g. RSA keys are limited to 512 bits and NOPAD mode is not available). Moreover, accurate benchmarking can not be performed because the code is executed by the simulator running on a 0x86 CPU and can only provide estimate values. Emulators, on the other hand, provide more accurate results as their behaviour is similar to the physical cards. However these are only available on proprietary tools, such as JCOP tools [37].

Details of the setting up the development environment, may be consulted elsewhere [90].

6.2 Performance evaluation

The performance of authentication schemes on a smart card depends not only on the efficiency of the implementation, but also on the delays that can be encountered either in the application running in the smart card, or in the application residing in the workstation [50]. In order to accurately measure the performance of our authentication schemes, we must first identify what affects the execution time in order to allow the isolation of the execution time of the features of interest.

6.2.1 Execution time isolation

Smart cards have no internal clock, as a result we must deduce the execution time of our application from the elapsed time between sending a command APDU and receiving the respective response APDU. However, this is not enough due to the significant and non-predictable elapse of time between the beginning of the measure, characterized by the starting of the timer on the computer, and the actual execution of the byte-code of interest [61]. This non-predictability is mainly dependent on hardware characteristics of the benchmark environment (such as the card acceptance device (CAD), PC's hardware, etc), the OS level interferences, services and also on the PC's VM [67, 60].

To isolate the execution time of the on-card code of interest, we must remove the communication overhead, or in other words, the fraction of elapsed time that does not depend on the efficiency of our implementation. As a matter of fact, before the command can be processed on the card's CPU, several layers of software and hardware must be traversed, as depicted in Figure 6.2.

A command APDU needs to be sent from the workstation host application and transmitted through the PC/SC interface with the card reader, before the command APDU can be transmitted to the smart card. After reaching the smart card, the JCRE must yet forward the command to the applet and, only then, does the applet take control and process the command. After the command is executed, the applet sends a response APDU to the host application and the inverse communication path is traversed.

While the delay in the workstation is mainly due to sudden load changes within the OS, there is also the delay associated to the card reader (CAD) and its drivers, as well the APDU interpretation/encapsulation in the smart card.

The execution time for the code of our implementation corresponds to the t_4 frame depicted on Figure 6.2, i.e., the execution time of the command on the card's CPU. Therefore, we must exclude the *communication overhead* introduced by the application layer (t_1), the PC/SC interface and the card reader (t_2), as well as the on-card APDU interpretation (t_3). Additionally, if data is transmitted, it's contribution to the total execution time should be considered as well.

A first approach to estimate the *on-card execution time* of a target command is to execute the command several times in a loop, and for each iteration collect the *total execution time*. Due to variance in the measurements, a sufficiently large number of samples is required to compute

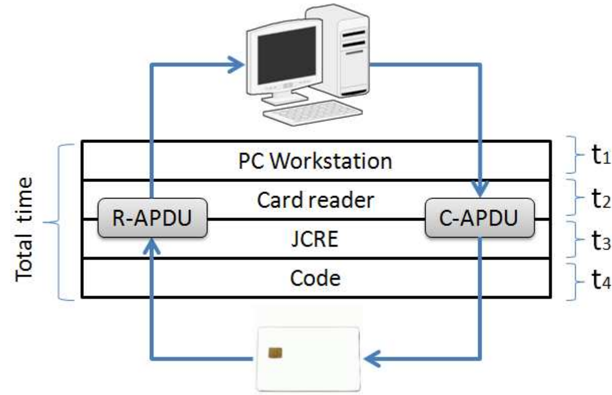


Figure 6.2: Communication path between the application on the computer and the card

the arithmetic mean and "filter" trustworthy results. The sampled values may also be stored persistently in a log file, allowing further access to the test data and statistical operations to be performed. With our proposed execution time measurement solution, we expect that the isolated execution time of the command of interest will stabilize after a certain loop size. This simplicity, however, requires that we perform a sufficiently large number of loops which may make the overall test tedious.

To compute the mean isolated execution time of the command of interest, we need to perform the following calculation:

$$\overline{M(command)} = \overline{m_L(TargetCommand)} - \overline{m_L(Overhead)} \tag{6.1}$$

where,

- $\overline{M(command)}$ is the estimation for the execution time of the code of interest.
- L represents the number of loop iterations, *i.e.*, the number of times the command under test is executed.
- $\overline{m_L(TargetCommand)}$ is the mean global execution time of the target command, including interferences coming from other operations performed during the measurement, both on the card and on the computer side, with respect to a loop size L.
- $\overline{m_L(Overhead)}$ represents the mean execution time of all the overhead functions: Smart card connection and disconnection, applet selection and data transmission — see section 6.3.1.

Despite the fact that the number of clock cycles are commonly used to compare the performance of different implementations, in this case it would not allow a fair comparison. The reason behind this is that interpreted languages, such as Java Card, have additional overhead caused by the complexity of the smart card operating system, making the execution of an application on top of the operating system much more expensive. For instance, in lightweight ciphers such as AES, the main source of computational cost may reside in the overhead caused by the operating system [41].

6.2.2 Test module configuration

Our test module consists of a script part and an applet part. While the script part, entirely written in Java, manages the execution of the tests and runs on a Windows/PC workstation, the applet is loaded on the smart card and implements the operations whose performance we want to measure.

The script makes use of the *javax.smartcardio* [53] package to exchange APDUs with the smart card and execute the target commands. Thereafter, the elapsed time between sending a command and receiving back the response is measured thanks to the *System.nanoTime* method.

In order to perform the tests required to solve equation 6.1 and evaluate the performance of our implementations, we have implemented the *GET_BYTES* function. The command *GET_BYTES* instructs the applet to receive the bytes contained in the data field of the command APDU, and to return back the same data field in the response APDU. If the data field of the APDU carries data, the applet must invoke the *apdu.setIncomingAndReceive()* method to receive the incoming data. In addition, the *apdu.sendBytesLong()* method must be used to return back the received data.

To measure the total execution time we set two timers: one to trigger before sending the first command and another to trigger after receiving the last response. Between the two timers the commands that constitute the test are executed. Card connection/disconnection and applet selection are included as well, since they increase the total execution time, thereby reducing the amount of standard deviation in the measurements.

6.3 Performance results

In this section we provide performance measurements related to the execution of authentication protocols. We begin by presenting performance results for communication overhead in section 6.3.1, followed by modular multiplication in section 6.3.2 and finally, authentication protocols in section 6.3.3.

6.3.1 Communication overhead

All values presented in this report refer to the *total time* needed to carry out an authentication protocol. Total time includes four overhead functions: (a) Smart card connection – 1,71ms, (b) Smart card disconnection – 0,20ms, (c) Applet selection – 3,21ms, and (d) Data transmission, where each value is obtained by computing the mean over 5000 executions.

In order to understand the impact of data transmission in the total execution time, the data transmission overhead values are depicted in Table 6.1, obtained by computing the mean over 5000 samples, for APDU with different lengths.

We consider a case 4 command APDU [66] where $L_e = L_c$, *i.e.* the amount of data contained in the C-APDU is the same amount of data which is to be returned in the R-APDU.

Table 6.1: Time spent for a data APDU

data field length(bytes)	0	16	64	96	128	160	192	224	255
time(ms)	2.98	6.98	10.99	15.00	17.62	21.00	24.02	27.75	30.01

Aside from the mandatory header and the L_c and L_e bytes, the APDU buffer may hold up to 255 bytes of data. When more than 255 bytes of data must be sent, the data must be split between several APDUs. Nevertheless, the P1 and P2 bytes can be used to provide extra input data. Note that these bytes are part of the header and therefore do not increase the data transmission time. For example, sending 256=2^8 Bytes requires one single APDU if we use P1 field.

Example 6.3.1 *Let the total time for an unilateral challenge-response authentication with AES, including card connection/disconnection and applet selection, equal 16,09 ms. If the challenge is composed of 16 bytes, the estimated on-card processing time is approximately:*
$16,09\,ms\,(total\,time) - 1,71\,ms\,(connect) - 0,20\,ms\,(disconnect) - 3,21\,ms\,(applet\,selection) - 6,98\,ms\,(data\,overhead) = 3,99\,ms.$

6.3.2 Modular Multiplication

Modular multiplication algorithms are described in section 4.2.3 and their average execution times are depicted in table 6.2, where execution times correspond to the mean over the values collected from 1000 iterations.

Tables 6.2a and 6.2b show that without coprocessor support, modular multiplication based on the algorithms originally proposed by Barrett and Montgomery exhibit similar performance. Both have approximately quadratic complexity and, therefore, are not fast enough for practical use.

On the other hand, squaring multiplication with coprocessor support, depicted in table 6.2c, exhibits superior performance results, even thought the algorithm is partly executed in the virtual machine. Note that the values depicted in table 6.2c correspond to the implementation of equation 4.2.

Table 6.2: Execution times for modular multiplication, with overheads excluded

(a) Montgomery Multiplication ($xyR^{-1} \mod n$)

Modulus (bits)	128	256	384	512
Time (s)	0.14	0.52	1.17	2.04

(b) Barrett Multiplication ($xy \mod n$)

Modulus (bits)	128	256	384	512
Time (s)	0.15	0.55	1.21	2.10

(c) Squaring Multiplication ($xy \mod n$)

Modulus (bits)	512	768	1024	1280
Time (ms)	65.87	91.15	123.84	162.43

6.3.3 Performance of Authentication Protocols

In this section we provide the average execution time for authentication protocols, with coprocessor support (C) and own implementation, *i.e.* programmed without coprocessor support (P), based on a total of 1000 measurements.

6.3.3.1 Authentication based on AES

The execution times for unilateral authentication are depicted in table 6.3, for coprocessor and own implementation.

The implementation of AES on the coprocessor runs virtually for free, which was expected since block ciphers are particularly efficient, especially when implemented on hardware. Even

Table 6.3: Execution times for AES unilateral authentication

Key size(bits)	128	192	256
Coprocessor(ms)	16.09	16.70	16.97
Programmed(ms)	68.93	79.94	90.97

though our implementation of AES is much slower than the coprocessor-enabled implementation, it still achieves acceptable execution times. This is probably due to the fact that AES consists mainly of logic operations and the most complex operations (multiplications in GF) can be replaced by lookup tables. The difference in execution times is most certainly due to the execution of byte code in the JVM instead of executing the code directly on the dedicated coprocessor.

6.3.3.2 Authentication based on RSA

The execution times for unilateral authentication with coprocessor support based on CRT, are depicted in table 6.4.

Table 6.4: Execution times for RSA

Modulus (bits)	512	768	1024	1280	1536	1792	2048
Coprocessor (ms)	159.08	253.23	436.18	602.48	786.51	994.54	1675.58

Without coprocessor, execution times for 64 and 128 bits are, respectively, 4.1s and 27.3s. These values are based on Barrett reduction. Nowadays, these keysizes are unsafe. However, these results indicate that safe keysizes would not be usable in practice, even if we were to resort to the CRT.

6.3.3.3 Authentication based on eLoBa

Execution times for eLoBa unilateral authentication vary according to the number of rounds. For a minimum of 16 rounds and a maximum of 31 rounds the execution times are 1.4s and 2.5s, respectively.

6.3.3.4 Authentication based on ZKP

Execution times required for unilateral authentication based on ZKP are depicted in table 6.5a and table 6.5b for coprocessor and own implementation, respectively.

Table 6.5: ZKP execution times

(a) with coprocessor

Modulus (bits)	512	768	1024	1280	1536	1792
FFS-C (s)	2.6	3.9	5.1	6.6	7.7	8.8
GQ-C (ms)	262.03	313.53	395.29	473.00	534.11	596.17

(b) without coprocessor

Modulus (bits)	64	128	192
FFS-P (s)	2.0	6.6	13.3
GQ-P (s)	2.3	7.0	14.8

6.3.3.5 Overall comparison

Figure 6.3 presents an overview of the execution times of the authentication protocols.

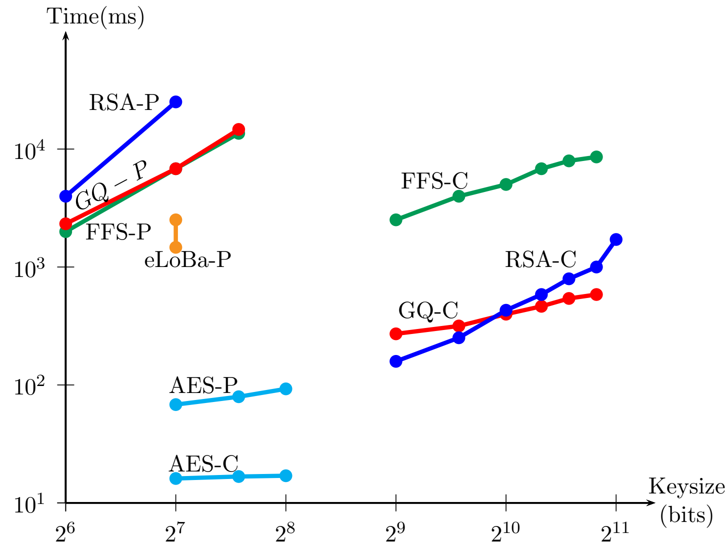


Figure 6.3: Comparison of authentication protocols

The AES cipher has the best performance among all the considered protocols, even without coprocessor support. The performance of eLoBa is excellent taking into consideration that it involves modular multiplications without resorting to the coprocessor.

Regarding RSA and ZKP, without coprocessor support FFS and GQ are faster than RSA. These results were expected as FFS and GQ involve operations which are computationally less expensive than RSA exponentiation. On the other hand, with coprocessor support we observe that RSA and GQ are faster than FFS. This contrast can be explained by the amount of operations that must execute on the VM. While RSA fully executes on the coprocessor, FFS and GQ require the computation of modular multiplications, which are only partially executed on the coprocessor. GQ outperforms FFS by the simple fact that only needs to perform a single modular multiplication.

It is also interesting to note that RSA is faster than GQ up until 896 bits. The explanation behind this behaviour is that as the module in the RSA-based authentication grows, so does

the exponent. On the other hand, GQ always uses an exponent that is bounded by the security parameter. This means that even though the size of the module increases, the cost of performing exponentiation does not grow at the same rate as in RSA.

6.4 Conclusions

In this thesis we have evaluated the performance of authentication schemes in a Java Card smart card. We have also implemented an authentication protocol based on a chaotic-based stream cipher and evaluated its performance against currently known cryptosystems [15].

We have shown that, due to the overhead of the Java Virtual Machine and the limited execution speed of the CPU, computations performed outside the coprocessor are too expensive. Java Card code is inherently inefficient and, as a result, reasonably fast implementations without hardware acceleration are only possible for very efficient algorithms, such as AES and eLoBa. We conclude that the performance of keystream chaotic system is only about 10% slower than AES, the fastest cryptosystem, and, therefore, is also suitable for authentication.

In order for RSA to remain presumably secure, the task of computing the private key given the public key must be computationally infeasible. However, larger key sizes require higher computational power and storage space, which are especially costly requirements for resource-constrained environments, such as smart cards.

The main performance limitation resides, however, in the Java-Card API, which restricts the access to the cryptographic coprocessor. API limitations force us to implement mathematically complex operations on the virtual machine, resulting in implementations that are too slow for practical use. Even efficient modular multiplication algorithms such as Barrett and Montgomery do not execute in reasonable time.

Nevertheless, it is possible to speed up the implementation of operations not directly offered by the Java Card API, through the clever usage of the available high-level cryptographic methods (*e.g.* RSA). By relying on the Java Card RSA interface, the *squaring multiplication* can be used to dramatically decrease the cost of modular multiplication, thus allowing FFS and GQ protocols to execute within acceptable time. However, the execution speed of the protocols is still bounded by the amount of computations that must be performed outside the coprocessor.

6.5 Further Work

As future work we propose to research optimizations to the authentication protocol based on the eLoBa cipher. A possible improvement would be to modify the authentication protocol in order to decrease the performance penalization of the initialization step.

To further improve the performance of protocols that rely on modular multiplication, the *squaring multiplication* method must also be enhanced. In this case, this is achieved by outsourcing more computation to the coprocessor, more specifically, by delegating the additions to the hardware acceleration. This should be possible by tunnelling additions through the RSA-CRT decryption operation [11].

Further work should also include a throughout analysis of the applet's memory requirements, making clear that memory consumption does not pose a problem for the chosen platform.

The implementation of algorithms using Elliptic Curve Cryptography (ECC) is also an interesting field. Compared to RSA, an equivalent level of security is achieved with smaller keys. Besides the performance advantage of ECC over RSA, the storage and transmission requirements are also considerably lower [35, 9].

Bibliography

[1] Java card development kit. http://java.sun.com/javacard/devkit/. [Online; accessed October 26, 2011].

[2] Pc/sc workgroup. http://www.pcscworkgroup.com/. [Online; accessed October 26, 2011].

[3] Sourceforge.net: Eclipsejcde. http://eclipse-jcde.sourceforge.net/.

[4] Top gx4 : Product information. http://www.gemalto.com/products/top_javacard/download/TOP_GX4_May10.pdf, May 2010. [Online; accessed October 26, 2011].

[5] Paul C. van Oorschot Alfred J. Menezes and Scott A. Vanstone. *Handbook of Applied Cryptography*. Crc Press Series on Discrete Mathematics and Its Applications, 1997.

[6] Michael Baentsch, Peter Buhler, Thomas Eirich, Frank Höring, and Marcus Oestreicher. Javacard-from hype to reality. *IEEE Concurrency*, 7:36–43, October 1999.

[7] Josep Balasch. Smart card implementation of anonymous credentials. Master's thesis, KU Leuven, 2007-2008.

[8] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Proceedings on Advances in cryptology—CRYPTO '86*, pages 311–323, London, UK, 1987. Springer-Verlag.

[9] L. Batina, J.-H. Hoepman, B. Jacobs, W. Mostowski, and P. Vullers. Developing efficient blinded attribute certificates on smart cards via pairings. In *9th IFIP WG 8.8/11.1 CARDIS 2010*, pages 209–222, April 2010.

[10] Jem E. Berkes. Side-channel monitoring of contactless java cards. Master's thesis, University of Waterloo, 2008.

[11] Patrik Bichsel, Jan Camenisch, Thomas Groß, and Victor Shoup. Anonymous credentials on a standard java card. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 600–610, New York, NY, USA, 2009. ACM.

[12] Arnaud Boscher and Robert Naciri. Optimal use of montgomery multiplication on smart cards. In *CARDIS*, pages 252–262, 2006.

[13] Antoon Bosselaers, René Govaerts, and Joos Vandewalle. Comparison of three modular reduction functions. In *In Advances in Cryptology-Crypto'93, LNCS 773*, pages 175–186. Springer-Verlag, 1994.

[14] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[15] Rui Gustavo Crespo and Jose Rafael Carvalho. Smartcard authentication scheme based on chaotic keystream cipher. In *e-Smart 2011 Proceedings*, September 2011.

[16] Des. Data encryption standard. In *In FIPS PUB 46, Federal Information Processing Standards Publication*, pages 46–2, 1977.

[17] Damien Deville, Antoine Galland, Gilles Grimaud, and Sébastien Jean. Smart card operating systems: Past, present and future. In *In Proceedings of the 5 th NORDU/USENIX Conference*, 2003.

[18] European Computer Manufacturers Association (ECMA). Standard ecma-335: Common language infrastructure (cli). http://www.ecma-international.org/publications/standards/Ecma-335.htm. [Online; accessed October 26, 2011].

[19] J. Elliot. The maos trap [smart card platforms]. *Computing Control Engineering Journal*, 12(1):4–10, feb 2001.

[20] EMVCo. Europay, mastercard and visa (emv). http://www.emvco.com/. [Online; accessed October 26, 2011].

[21] Çetin Kaya Koç. High-speed rsa implementation. Technical report, RSA Laboratories, 1994.

[22] European Telecommunications Standards Institute (ETSI). Mobile technologies gsm. http://www.etsi.org/WebSite/Technologies/gsm.aspx. [Online; accessed October 26, 2011].

[23] Council European Parliament. Directive 1999/93/ec of the european parliament and of the council of 13 december 1999 on a community framework for electronic signatures. http://europa.eu/legislation_summaries/information_society/other_policies/l24118_en.htm, December 1999. [Online; accessed October 26, 2011].

[24] Eurosmart. Market overview. url=http://www.eurosmart.com/index.php/publications/market-overview.html, December 2010. [Online; accessed October 26, 2011].

[25] Announcing The Federal. Federal information processing standards publication 197. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, November 2001. [Online; accessed October 26, 2011].

[26] U. Feige, A. Fiat, and A. Shamir. Zero-knowledge proofs of identity. *J. Cryptol.*, 1:77–94, August 1988.

[27] Amos Fiat and Adi Shamir. How to prove yourself: practical solutions to identification and signature problems. In *Proceedings on Advances in cryptology—CRYPTO '86*, pages 186–194, London, UK, 1987. Springer-Verlag.

[28] Patrick Gallagher, Deputy Director Foreword, and Cita Furlani Director. Fips pub 186-3 federal information processing standards publication digital signature standard (dss), 2009.

[29] P. Girard, K. Villegas, J.-L. Lanet, and A. Plateaux. A new payment protocol over the internet. In *Risks and Security of Internet and Systems (CRiSIS), 2010 Fifth International Conference on*, pages 1 –6, oct. 2010.

[30] GlobalPlatform. Gpshell. http://sourceforge.net/projects/globalplatform/. [Online; accessed October 26, 2011].

[31] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, STOC '85, pages 291–304, New York, NY, USA, 1985. ACM.

[32] E government II, C. Troncoso(Ed.), C. Diaz, V. Gornishki, V. Sucasas, M. Kohlweiss, M. Sterckx, and J. Balasch. Advanced applications for e-id cards in flanders adapid deliverable d15, January 2010.

[33] J. Groβchadl. The chinese remainder theorem and its application in a high-speed rsa crypto chip. In *Proceedings of the 16th Annual Computer Security Applications Conference*, ACSAC '00, pages 384–, Washington, DC, USA, 2000. IEEE Computer Society.

[34] L. C. Guillou and J.-J. Quisquater. A practical zero-knowledge protocol fitted to security microprocessor minimizing both transmission and memory. In *Lecture Notes in Computer Science on Advances in Cryptology-EUROCRYPT'88*, pages 123–128, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[35] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and rsa on 8-bit cpus. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 925–943. Springer Berlin - Heidelberg, 2004.

[36] William Hasenplaugh, Gunnar Gaubatz, and Vinodh Gopal. Fast modular reduction. In *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 225–229, Washington, DC, USA, 2007. IEEE Computer Society.

[37] IBM. Jcop tools 3.0. ftp://ftp.software.ibm.com/software/pervasive/info/JCOPTools3Brief.pdf.

[38] Government International Standards for Business and Society. Iso/iec 14443. http://www.iso.org/iso/search.htm?qt=14443&published=on&active_tab=standards. [Online; accessed October 26, 2011].

[39] Government International Standards for Business and Society. Iso/iec 7816. http://www.iso.org/iso/search.htm?qt=7816&published=on&active_tab=standards. [Online; accessed October 26, 2011].

[40] Government International Standards for Business and Society. Iso/iec 9798-2. http://www.iso.org/iso/search.htm?qt=9798&published=on&active_tab=standards, 15 2008. [Online; accessed October 26, 2011].

[41] Yiannakis Ioannou. A software implementation of aes for a multos smart card. Master's thesis, Royal Holloway, University of London, September 2006.

[42] ITU-T. *ITU-T Rec. Z.100 – Formal description techniques (FDT) – Specification and Description Language (SDL)*, 2007.

[43] Marcelo E. Kaihara and Naofumi Takagi. Bipartite modular multiplication. In *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2005, number 3659 in Lecture notes in Computer Science*, pages 201–210. Springer-Verlag, 2005.

[44] Miroslav Knezevic, Frederik Vercauteren, and Ingrid Verbauwhede. Faster interleaved modular multiplication based on barrett and montgomery reduction methods. *IEEE Trans. Comput.*, 59:1715–1721, December 2010.

[45] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1981.

[46] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[47] Hugo Krawczyk. Lfsr-based hashing and authentication. In *CRYPTO'94*, pages 129–139, 1994.

[48] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56:131–133, November 1995.

[49] MAOSCO Ltd. Multos developer's guide, 2006. [Online; accessed October 26, 2011].

[50] Konstantinos Markantonakis. Is the performance of smart card cryptographic functions the real bottleneck? In *Proceedings of the 16th international conference on Information security: Trusted information: the new decade challenge*, Sec '01, pages 77–91, Norwell, MA, USA, 2001. Kluwer Academic Publishers.

[51] Konstantinos Markantonakis and Keith Mayes. An overview of the globalplatform smart card specification. *Information Security Technical Report*, 8(1):17 – 29, 2003.

[52] Keith Mayes and Konstantinos Markantonakis. *Smart Cards, Tokens, Security and Applications*. Springer Science+Business Media, 2008.

[53] Sun Microsystems. Java smart card i/o api - package summary. http://download.oracle.com/javase/6/docs/jre/api/security/smartcardio/spec/javax/smartcardio/package-summary.html. [Online; accessed October 26, 2011].

[54] Victor S. Miller. Use of elliptic curves in cryptography. In *CRYPTO*, pages 417–426, 1985.

[55] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, April 1985.

[56] Michel Mouly and Marie-Bernadette Pautet. *The GSM System for Mobile Communications*. Telecom Publishing, 1992.

[57] Nadia Nedjah and Luiza de Macedo Mourelle. A review of modular multiplication methods ands respective hardware implementation. *Informatica (Slovenia)*, 30(1):111–129, 2006.

[58] NIST, editor. *An Introduction to Computer Security : The NIST Handbook*, Special Publication 800-12, October 1995.

[59] Tatsuaki Okamoto. Provably secure and practical identification schemes and corresponding signature schemes. In Ernest Brickell, editor, *Advances in Cryptology—CRYPTO 92*, volume 740 of *Lecture Notes in Computer Science*, pages 31–53. Springer Berlin / Heidelberg, 1993.

[60] Pierre Paradinas, Julien Cordry, and Samia Bouzefrane. Measurement analysis when benchmarking java card platforms. In *Proceedings of the 3rd IFIP WG 11.2 International Workshop on Information Security Theory and Practice. Smart Devices, Pervasive Systems, and Ubiquitous Networks*, WISTP '09, pages 84–94, Berlin, Heidelberg, 2009. Springer-Verlag.

[61] JulienCordry PierreParadinas and SamiaBouzefrane. Performance evaluation of java card bytecodes. *IFIP International Federation for Information Processing*, pages 127–137, 2007.

[62] Fachgebiet Kryptographische Protokolle, Dr. Tsuyoshi Takagi Junior professor, and Camille Vuillaume. Efficiency comparison of several rsa variants, 2002-2003.

[63] Federal Information Processing Standards Publications. Secure Hash Standard (SHS). Technical Report FIPS PUB 180-3, National Institute of Standards and Technology, October 2008.

[64] Jean-Jacques Quisquater, Myriam Quisquater, Muriel Quisquater, Michaël Quisquater, Louis C. Guillou, Marie Annick Guillou, Gaïd Guillou, Anna Guillou, Gwenolé Guillou, Soazig Guillou, and Thomas A. Berson. How to explain zero-knowledge protocols to your children. In *CRYPTO*, pages 628–631, 1989.

[65] W. Rankl and W. Effing. *Smart Card Handbook.* John Wiley & Sons, Inc., New York, NY, USA, 3 edition, 2003.

[66] Wolfgang Rankl and Kenneth Cox. *Smart Card Applications: Design models for using and programming smart cards.* John Wiley & Sons, Inc., New York, NY, USA, 2007.

[67] Karima Rehioui. Java card performance test framework. Master's thesis, Université de Nice - Sophia-Antipolis, 2005.

[68] R. Rivest. The md5 message-digest algorithm, 1992.

[69] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21:120–126, February 1978.

[70] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, Inc., New York, NY, USA, 1993.

[71] C. P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4:161–174, 1991. 10.1007/BF00196725.

[72] Claus P. Schnorr. Efficient identification and signatures for smart cards. In *Proceedings on Advances in cryptology*, CRYPTO '89, pages 239–252, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[73] RSA Data Security, 1987.

[74] Robert Sedgewick. *Algorithms in C - parts 1-4: fundamentals, data structures, sorting, searching (3. ed.).* Addison-Wesley-Longman, 1998.

[75] George Selimis, Apostolos Fournaris, George Kostopoulos, and Odysseas Koufopavlou. Software and hardware issues in smart card technology. *Communications Surveys Tutorials IEEE*, 11(3):143–152, 2009.

[76] R.M. Silva, R.G. Crespo, and M.S. Nunes. LoBa128, a Lorenz Based PRNG for Wireless Sensor Networks. *Special Issue of International Journal of Communication Networks*

and Distributed Systems on Emerging Security Issues in Communication Networks and Distributed Systems, 3(4):301–318, August 2009.

[77] Rui Miguel Soares Silva, Rui Gustavo Nunes Pereira Crespo, and Mario Serafim dos Santos Nunes. Enhanced chaotic stream cipher for wsns. *International Conference on Availability, Reliability and Security*, 0:210–215, 2010.

[78] Southern African Telecommunication Networks and Applications Conference (SATNAC). *Design and Implementation of Fast Multiplication Algorithms in Public Key Cryptosystems for Smart Cards*, 2003.

[79] William Stallings. *Cryptography and Network Security: Principles and Practice.* Prentice Hall, 4th edition, 2005.

[80] G. Starnberger, L. Froihofer, and K.M. Goeschka. Using smart cards for tamper-proof timestamps on untrusted clients. In *Availability, Reliability, and Security, 2010. ARES '10 International Conference on*, pages 96 –103, feb. 2010.

[81] Michael Sterckx. Implementation and side-channel analysis of anonymous credentials on java card platforms. Master's thesis, KU Leuven, 2008-2009.

[82] Michaël Sterckx, Benedikt Gierlichs, Bart Preneel, and Ingrid Verbauwhede. Efficient implementation of anonymous credentials on java card smart cards. In *1st IEEE International Workshop on Information Forensics and Security (WIFS 2009)*, pages 106–110, London,UK, 2009. IEEE.

[83] Douglas Stinson. *Cryptography: Theory and Practice.* Chapman & HALL/CRC, 3rd edition, 2006.

[84] Hendrik Tews and Bart Jacobs. Performance issues of selective disclosure and blinded issuing protocols on java card. In *Proceedings of the 3rd IFIP WG 11.2 International Workshop on Information Security Theory and Practice. Smart Devices, Pervasive Systems, and Ubiquitous Networks*, WISTP '09, pages 95–111, Berlin, Heidelberg, 2009. Springer-Verlag.

[85] Henk C.A. van Tilborg. *Encyclopedia of Cryptography and Security.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[86] Scott B. Guthery Timothy M. Jurgensen. *Smart Cards: The Developer's Toolkit (Essential Guide)*. Prentice Hall PTR, 2002.

[87] Joaquin Torres, Antonio Izquierdo, and Jose Maria Sierra. Advances in network smart cards authentication. *Comput. Netw.*, 51:2249–2261, June 2007.

[88] International Telecommunication Union. Recommendation x.800 : Security architecture for open systems interconnection for ccitt applications. http://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.800-199103-I!!PDF-E&type=items, 1991. [Online; accessed October 26, 2011].

[89] John Viega and Gary McGraw. *Building Secure Software - How to avoid security problems the right way*. Addison-Wesley Professional computing series, 2008.

[90] Jan Vossaert, Jorn Lapon, and Vincent Naessens. Developing secure java card applications. https://www.msec.be/jorn/, February 2011. [Online; accessed October 26, 2011].

[91] Song Y. Yan. *Number Theory for Computing*. Springer, 2nd edition, 2002.

Appendix A

RSA exponentiation

Step 1 - Create instance of a Cipher and key(s)

```
Cipher rsa;
RSAPublicKey pubKey;
try{
  rsa=javacardx.crypto.Cipher .getinstance(javacardx.Crypto.Cipher.ALG_RSA_NOPAD,false);
  pubKey=javacardx.security.KeyBuilder.buildkey();
}
 catch(){ /*RSA Crypto engine not supported by this card*/ }
```

Step 2 - Initialize key(s)

```
pubKey.setModulus(modulus,(short)0,modulus_len);
pubKey.setExponent(exponent,(short)0,exponent_len);
```

Step 3 - Initialize cryptographic engine

```
rsa.init(pubKey,MODE_ENCRYPT);
```

Step 4 - Perform encryption

```
rsa.doFinal(buffer2encrypt,(short)inOffset,(short)inLength,outputBuffer,(short)outOffset);
/*use rsa.update() to feed input data cumulatively, if the entire input data cannot be fit in
a byte array*/
```

Appendix B

AES tables

B.1 AES S-Boxes

Table B.1: AES forward S-box

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1x	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2x	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3x	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4x	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5x	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6x	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7x	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8x	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9x	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
ax	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
bx	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
cx	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
dx	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
ex	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
fx	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Table B.2: AES inverted S-box

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1x	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2x	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3x	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4x	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5x	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6x	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7x	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8x	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9x	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
ax	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
bx	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
cx	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
dx	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
ex	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
fx	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

B.2 AES multiplication tables

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	00	02	04	06	08	0a	0c	0e	10	12	14	16	18	1a	1c	1e
1x	20	22	24	26	28	2a	2c	2e	30	32	34	36	38	3a	3c	3e
2x	40	42	44	46	48	4a	4c	4e	50	52	54	56	58	5a	5c	5e
3x	60	62	64	66	68	6a	6c	6e	70	72	74	76	78	7a	7c	7e
4x	80	82	84	86	88	8a	8c	8e	90	92	94	96	98	9a	9c	9e
5x	a0	a2	a4	a6	a8	aa	ac	ae	b0	b2	b4	b6	b8	ba	bc	be
6x	c0	c2	c4	c6	c8	ca	cc	ce	d0	d2	d4	d6	d8	da	dc	de
7x	e0	e2	e4	e6	e8	ea	ec	ee	f0	f2	f4	f6	f8	fa	fc	fe
8x	1b	19	1f	1d	13	11	17	15	0b	09	0f	0d	03	01	07	05
9x	3b	39	3f	3d	33	31	37	35	2b	29	2f	2d	23	21	27	25
ax	5b	59	5f	5d	53	51	57	55	4b	49	4f	4d	43	41	47	45
bx	7b	79	7f	7d	73	71	77	75	6b	69	6f	6d	63	61	67	65
cx	9b	99	9f	9d	93	91	97	95	8b	89	8f	8d	83	81	87	85
dx	bb	b9	bf	bd	b3	b1	b7	b5	ab	a9	af	ad	a3	a1	a7	a5
ex	db	d9	df	dd	d3	d1	d7	d5	cb	c9	cf	cd	c3	c1	c7	c5
fx	fb	f9	ff	fd	f3	f1	f7	f5	eb	e9	ef	ed	e3	e1	e7	e5

Table B.3: AES m2 table

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	00	03	06	05	0c	0f	0a	09	18	1b	1e	1d	14	17	12	11
1x	30	33	36	35	3c	3f	3a	39	28	2b	2e	2d	24	27	22	21
2x	60	63	66	65	6c	6f	6a	69	78	7b	7e	7d	74	77	72	71
3x	50	53	56	55	5c	5f	5a	59	48	4b	4e	4d	44	47	42	41
4x	c0	c3	c6	c5	cc	cf	ca	c9	d8	db	de	dd	d4	d7	d2	d1
5x	f0	f3	f6	f5	fc	ff	fa	f9	e8	eb	ee	ed	e4	e7	e2	e1
6x	a0	a3	a6	a5	ac	af	aa	a9	b8	bb	be	bd	b4	b7	b2	b1
7x	90	93	96	95	9c	9f	9a	99	88	8b	8e	8d	84	87	82	81
8x	9b	98	9d	9e	97	94	91	92	83	80	85	86	8f	8c	89	8a
9x	ab	a8	ad	ae	a7	a4	a1	a2	b3	b0	b5	b6	bf	bc	b9	ba
ax	fb	f8	fd	fe	f7	f4	f1	f2	e3	e0	e5	e6	ef	ec	e9	ea
bx	cb	c8	cd	ce	c7	c4	c1	c2	d3	d0	d5	d6	df	dc	d9	da
cx	5b	58	5d	5e	57	54	51	52	43	40	45	46	4f	4c	49	4a
dx	6b	68	6d	6e	67	64	61	62	73	70	75	76	7f	7c	79	7a
ex	3b	38	3d	3e	37	34	31	32	23	20	25	26	2f	2c	29	2a
fx	0b	08	0d	0e	07	04	01	02	13	10	15	16	1f	1c	19	1a

Table B.4: AES m3 table

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	00	09	12	1b	24	2d	36	3f	48	41	5a	53	6c	65	7e	77
1x	90	99	82	8b	b4	bd	a6	af	d8	d1	ca	c3	fc	f5	ee	e7
2x	3b	32	29	20	1f	16	0d	04	73	7a	61	68	57	5e	45	4c
3x	ab	a2	b9	b0	8f	86	9d	94	e3	ea	f1	f8	c7	ce	d5	dc
4x	76	7f	64	6d	52	5b	40	49	3e	37	2c	25	1a	13	08	01
5x	e6	ef	f4	fd	c2	cb	d0	d9	ae	a7	bc	b5	8a	83	98	91
6x	4d	44	5f	56	69	60	7b	72	05	0c	17	1e	21	28	33	3a
7x	dd	d4	cf	c6	f9	f0	eb	e2	95	9c	87	8e	b1	b8	a3	aa
8x	ec	e5	fe	f7	c8	c1	da	d3	a4	ad	b6	bf	80	89	92	9b
9x	7c	75	6e	67	58	51	4a	43	34	3d	26	2f	10	19	02	0b
ax	d7	de	c5	cc	f3	fa	e1	e8	9f	96	8d	84	bb	b2	a9	a0
bx	47	4e	55	5c	63	6a	71	78	0f	06	1d	14	2b	22	39	30
cx	9a	93	88	81	be	b7	ac	a5	d2	db	c0	c9	f6	ff	e4	ed
dx	a0	03	18	11	2e	27	3c	35	42	4b	50	59	66	6f	74	7d
ex	a1	a8	b3	ba	85	8c	97	9e	e9	e0	fb	f2	cd	c4	df	d6
fx	31	38	23	2a	15	1c	07	0e	79	70	6b	62	5d	54	4f	46

Table B.5: AES m9 table

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	00	0b	16	1d	2c	27	3a	31	58	53	4e	45	74	7f	62	69
1x	b0	bb	a6	ad	9c	97	8a	81	e8	e3	fe	f5	c4	cf	d2	d9
2x	7b	70	6d	66	57	5c	41	4a	23	28	35	3e	0f	04	19	12
3x	cb	c0	dd	d6	e7	ec	f1	fa	93	98	85	8e	bf	b4	a9	a2
4x	f6	fd	e0	eb	da	d1	cc	c7	ae	a5	b8	b3	82	89	94	9f
5x	46	4d	50	5b	6a	61	7c	77	1e	15	08	03	32	39	24	2f
6x	8d	86	9b	90	a1	aa	b7	bc	d5	de	c3	c8	f9	f2	ef	e4
7x	3d	36	2b	20	11	1a	07	0c	65	6e	73	78	49	42	5f	54
8x	f7	fc	e1	ea	db	d0	cd	c6	af	a4	b9	b2	83	88	95	9e
9x	47	4c	51	5a	6b	60	7d	76	1f	14	09	02	33	38	25	2e
ax	8c	87	9a	91	a0	ab	b6	bd	d4	df	c2	c9	f8	f3	ee	e5
bx	3c	37	2a	21	10	1b	06	0d	64	6f	72	79	48	43	5e	55
cx	01	0a	17	1c	2d	26	3b	30	59	52	4f	44	75	7e	63	68
dx	b1	ba	a7	ac	9d	96	8b	80	e9	e2	ff	f4	c5	ce	d3	d8
ex	7a	71	6c	67	56	5d	40	4b	22	29	34	3f	0e	05	18	13
fx	ca	c1	dc	d7	e6	ed	f0	fb	92	99	84	8f	be	b5	a8	a3

Table B.6: AES mB table

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	00	0d	1a	17	34	39	2e	23	68	65	72	7f	5c	51	46	4b
1x	d0	dd	ca	c7	e4	e9	fe	f3	b8	b5	a2	af	8c	81	96	9b
2x	bb	b6	a1	ac	8f	82	95	98	d3	de	c9	c4	e7	ea	fd	f0
3x	6b	66	71	7c	5f	52	45	48	03	0e	19	14	37	3a	2d	20
4x	6d	60	77	7a	59	54	43	4e	05	08	1f	12	31	3c	2b	26
5x	bd	b0	a7	aa	89	84	93	9e	d5	d8	cf	c2	e1	ec	fb	f6
6x	d6	db	cc	c1	e2	ef	f8	f5	be	b3	a4	a9	8a	87	90	9d
7x	06	0b	1c	11	32	3f	28	25	6e	63	74	79	5a	57	40	4d
8x	da	d7	c0	cd	ee	e3	f4	f9	b2	bf	a8	a5	86	8b	9c	91
9x	0a	07	10	1d	3e	33	24	29	62	6f	78	75	56	5b	4c	41
ax	61	6c	7b	76	55	58	4f	42	09	04	13	1e	3d	30	27	2a
bx	b1	bc	ab	a6	85	88	9f	92	d9	d4	c3	ce	ed	e0	f7	fa
cx	b7	ba	ad	a0	83	8e	99	94	df	d2	c5	c8	eb	e6	f1	fc
dx	67	6a	7d	70	53	5e	49	44	0f	02	15	18	3b	36	21	2c
ex	0c	01	16	1b	38	35	22	2f	64	69	7e	73	50	5d	4a	47
fx	dc	d1	c6	cb	e8	e5	f2	ff	b4	b9	ae	a3	80	8d	9a	97

Table B.7: AES mD table

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	00	0e	1c	12	38	36	24	2a	70	7e	6c	62	48	46	54	5a
1x	e0	ee	fc	f2	d8	d6	c4	ca	90	9e	8c	82	a8	a6	b4	ba
2x	db	d5	c7	c9	e3	ed	ff	f1	ab	a5	b7	b9	93	9d	8f	81
3x	3b	35	27	29	03	0d	1f	11	4b	45	57	59	73	7d	6f	61
4x	ad	a3	b1	bf	95	9b	89	87	dd	d3	c1	cf	e5	eb	f9	f7
5x	4d	43	51	5f	75	7b	69	67	3d	33	21	2f	05	0b	19	17
6x	76	78	6a	64	4e	40	52	5c	06	08	1a	14	3e	30	22	2c
7x	96	98	8a	84	ae	a0	b2	bc	e6	e8	fa	f4	de	d0	c2	cc
8x	41	4f	5d	53	79	77	65	6b	31	3f	2d	23	09	07	15	1b
9x	a1	af	bd	b3	99	97	85	8b	d1	df	cd	c3	e9	e7	f5	fb
ax	9a	94	86	88	a2	ac	be	b0	ea	e4	f6	f8	d2	dc	ce	c0
bx	7a	74	66	68	42	4c	5e	50	0a	04	16	18	32	3c	2e	20
cx	ec	e2	f0	fe	d4	da	c8	c6	9c	92	80	8e	a4	aa	b8	b6
dx	0c	02	10	1e	34	3a	28	26	7c	72	60	6e	44	4a	58	56
ex	37	39	2b	25	0f	01	13	1d	47	49	5b	55	7f	71	63	6d
fx	d7	d9	cb	c5	ef	e1	f3	fd	a7	a9	bb	b5	9f	91	83	8d

Table B.8: AES mE table

Appendix C

Modular multiplication in eLoBa

- Multiplications are processed Byte by Byte.
- The operands, stored in a Byte array, are $a = (a_{15}, a_{14}, \cdots, a_0)_8$ and $b = (b_{15}, b_{14}, \cdots, b_0)_8$, while the product is stored in $r = (r_{15}, r_{14}, \cdots, r_0)_8$.
- The product of two Bytes, for example a_{12} and b_{07}, results in Bytes $12c07H||12c07L$, where '||' denotes the concatenation of two Bytes.

C.1 1 Byte by 16 Bytes modular multiplication

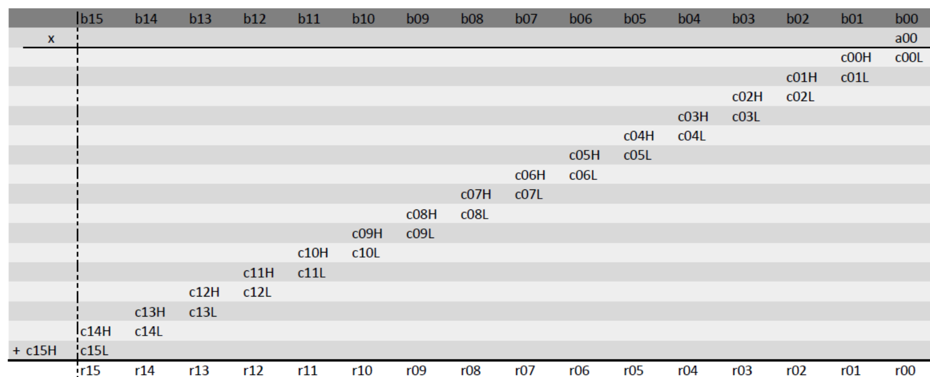


Figure C.1: 1 Byte by 16 Bytes modular multiplication

C.2 16 Bytes by 16 Bytes modular multiplication

	b15	b14	b13	b12	b11	b10	b09	b08	b07	b06	b05	b04	b03	b02	b01	b00
x	a15	a14	a13	a12	a11	a10	a09	a08	a07	a06	a05	a04	a03	a02	a01	a00
															00c00H	00c00L
														00c01H	00c01L	
													00c02H	00c02L		
												00c03H	00c03L	01c00H	01c00L	
											00c04H	00c04L	01c01H	01c01L		
										00c05H	00c05L	01c02H	01c02L	02c00H	02c00L	
									00c06H	00c06L	01c03H	01c03L	02c01H	02c01L		
								00c07H	00c07L	01c04H	01c04L	02c02H	02c02L	...	...	
							00c08H	00c08L	01c05H	01c05L	02c03H	02c03L	...	...		
						00c09H	00c09L	01c06H	01c06L	02c04H	02c04L	...	...			
					00c10H	00c10L	01c07H	01c07L	02c05H	02c05L	...	...				
				00c11H	00c11L	01c08H	01c08L	02c06H	02c06L	...	...					
			00c12H	00c12L	01c09H	01c09L	02c07H	02c07L	...	...						
		00c13H	00c13L	01c10H	01c10L	02c08H	02c08L	...	...							
	00c14H	00c14L	01c11H	01c11L	02c09H	02c09L	...	...								
00c15H	00c15L	01c12H	01c12L	02c10H	02c10L	...	...									
	01c13H	01c13L	02c11H	02c11L	...	...										
+ 01c14H	01c14L	02c12H	02c12L	...	...											
	r15	r14	r13	r12	r11	r10	r09	r08	r07	r06	r05	r04	r03	r02	r01	r00

Figure C.2: 16 Bytes by 16 Bytes modular multiplication