University of Minho
Engineering School
Master Course in Informatics

Master Thesis

2007/08

On the implementation of cryptographic algorithms in Java

Rui Miguel da Silva Abreu

Under supervison of: Professor Jose Carlos Bacelar Almeida, Department of Informatics, University of Minho

November, 2008

Acknowledgements

First of all, I would like to express my gratitude to my supervisor Jose Carlos Bacelar Almeida. Without his expertise, pacience and careful guidance, this work would not be possible.

I would also like to thank Luis Campos from PDM&FC and the WeDo Consulting, for letting me have some time to dedicate myself to this work.

Finally, I want to thank my family and friends for all the support. Writing a thesis while maintaining a full-time job at a software company requires a great will of mind and perserverance and the thesis development met some setbacks due to my professional situation.

To each of the above, I extend my deepest appreciation.

It is not the critic who counts, not the man who points out how the strong man stumbled, or where the doer of deeds could have done better. The credit belongs to the man who is actually in the arena, whose face is marred by dust and sweat and blood, who strives valiantly, who errs and comes short again and again, who knows the great enthusiasms, the great devotions, and spends himself in a worthy cause, who at best knows achievement and who at the worst if he fails at least fails while daring greatly so that his place shall never be with those cold and timid souls who know neither victory nor defeat.

Theodore Roosevelt

Abstract

Cryptography software has a critical nature because any failure can compromise the system's security. Therefore, its implementation must follow strict rules. These rules are defined by cryptography standards, which must serve as guidelines to programmers that implement the algorithms in the different programming languages.

The emergence of Java as one of the most popular programming languages makes it a natural choice for encoding cryptographic algorithms. The aim of this thesis is to analyse that particular use of Java. More concretely, we are interested in the following perspectives:

- functional correctness- judge how close actual implementations are with respect to funcional descriptions made available by cryptography standards.
- efficiency considerations- the appropriateness of pure Java implementations for computationally demanding cryptographic techniques, such as those based in elliptic curves over finite fields.

This work evaluates selected cryptographic algorithms from publicly available libraries. Their implementations were matched against relevant cryptography standards and efficiency compared with state-of-art C++ implementations. A solution based on extending JVM with selected operations was proposed and tested to obviate Java efficiency limitations.

Applications Areas: Cryptography.

Keywords: Java, C++, Cryptographic standards, Elliptic Curve Cryptography, Identity-Based Encryption, MIRACL.

Resumo

Título da tese: Implementação de algoritmos criptográficos em Java.

O *software* criptográfico tem uma natureza crítica, já que qualquer falha pode comprometer a segurança de um sistema. Assim sendo, a sua implementação tem que seguir regras rígidas. Estas regras são definidas nos *standards* de criptografia, que servem de guias para os programadores que implementam os algoritmos criptográficos nas diferentes linguagens de programação.

O facto de Java ter emergido como uma das mais populares linguagens de programação, faz com que seja uma escolha natural para codificar algoritmos de criptografia. O objectivo desta tese é analisar o uso de Java para esse fim particular. Mais concretamente, interessam as seguintes perspectivas:

- correcção funcional - avaliar o quão próximo das descrições funcionais dos *standards* se encontram implementações reais dos algoritmos.
- questões de eficiência - avaliar até que ponto implementações puras de Java são apropriadas para técnicas criptográficas exigentes em termos de recursos computacionais, tais como as técnicas baseadas em curvas elípticas sobre corpos finitos.

Este trabalho avalia alguns algoritmos criptográficos presentes em bibliotecas de domínio público. As suas implementações são confrontadas com *standards* de criptografia relevantes e a sua eficiência é comparada com implementações *state-of-art* em C++. Uma solução baseada em embeber C++ na JVM é proposta e testada para ultrapassar as limitações de eficiência de Java

Áreas de aplicação: Criptografia.

Keywords: Java, C++, *Standards* de criptografia, Criptografia baseada em curvas elípticas , Criptografia baseada em identidade, MIRACL.

Acronyms List

AES Advanced Encryption Standard

ANSI American National Standards Institute

API Application Programming Interface

BC Bouncy Castle

BDHP Bilinear Decisional Diffie-Hellman Problem

CC Common Criteria

CCRA Common Criteria Recognition Arrangement

CRT Chinese Remainder Theorem

CSE Communications Security Establishment

CSP Critical Security Parameter

DSA Digital Signature Algorithm

DSS Digital Siganature Standard

EAL Evaluation Assurance Level

ECC Elliptic Curve Cryptography

ECDLP Elliptic Curve Discrete Logarithm Problem

ECDSA Elliptic Curve Digital Signature Algorithm

ECSP-DSA Elliptic Curve Signature Primitive-Digital Signature Algorithm

ECSVDP-DH Elliptic Curve Secret Value Derivation Primitive-Diffie-Hellman

ECVP-DSA Elliptic Curve Verification Primitive-Digital Signature Algorithm

EMC Electromagnetic Compatibility

EMI Electromagnetic Interference

FIPS Federal Information Processing Standards

HMAC Hash Message Authentication Code

I2OSP Integer-to-Octet-String-Primitive

IBC Identity Based Cryptography

IBE Identity Based Encryption

IBS Identity Based Signature

IEEE Institute of Electrical and Electronics Engineers

ISO International Standards Organization

IT Information Technologies

J2ME Java 2 Micro Edition

JCA Java Cryptography Architecture

JCE Java Cryptography Extension

JNI Java Native Interface

JRE Java Runtime Environment

JVM Java Virtual Machine

MIRACL Multiprecision Integer and Rational Arithmetic C/C++ Library

MGF Mask Generation Function

NIST National Insitute of Standards for Technology

OAEP Optimal Asymmetric Encryption Padding

OCSP Online Certificate Status Protocol

OS2IP Octet-String-to-Integer Primitive

PKCS Public-Key Cryptography Standards

PKG Private Key Generator

PKI Public Key Infrastructure

PP Protection Profile

RSADP RSA Decryption Primitive

RSAEP RSA Encryption Primitive

RSAES RSA Encryption Scheme

SEC Standards for Efficient Cryptography

SHS Secure Hash Standard

SPI Service Provider Interface

ST Security Target

TLS Transport Layer Security

TOE Target of Evaluation

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivations

Cryptographic software often plays a critical role in computational systems. Any failure or deficiency in a cryptographic module can be exploited to compromise the system's safety, which can lead to catastrophic events from the security point of view.

Every system that runs cryptographic software, does it because it needs to protect information. From the email sent between friends, to industrial projects or government secrets, the need to hide data from an adversary is real and necessary. In the modern society, where the access and use of computer networks is something that grows everyday, the use of Cryptography is imperative.

When correctly implemented and used, cryptography is able to effectively protect data. Not because it is absolutely unbreakable, but because it makes unauthorised acquisition of the information computationally infeasible or cost-prohibitive.

A programmer willing to protect some sensitive data should choose the most appropriate cryptographic technique, find a sufficiently detailed description to permit its encoding in the target programming language, adopt formats for the data and be aware of different sorts of vulnerabilities and associated counter-measures. It is definitely a difficult task that demands deep knowledge in a wide range of areas. Some guidelines are needed and here's where the cryptography standards appear.

Cryptographic standards are specifications that describe every area related with the implementation of cryptosystems. Some, such as Federal Information Processing Standards 140 (FIPS-140) or the Common Criteria (CC), are high level standards, in the sense that they targeted towards

security evaluation and product certification. Others, like RSA's Public-Key Cryptography Standards are positioned at a lower level, providing specifications regarding the implementation of the algorithms and manipulation of sensitive data.

Adherence to the standards assures that a correct implementation and use of a cryptographic technique provides the associated security level. This is because those techniques have been previously tested and validated by renowned and trusted scientific organisations. This makes these standards one of the most valuable source for programmers of cryptographic modules.

Java, as one of the most used programming languages worldwide, has been adopted for cryptographic software development. Java was developed by Sun Microsystems and is recognised as a portable, elegant, secure, easy to learn and to maintain language. Two key aspects of Java stand out:

1. Its portability. Because Java programs run on top of a virtual machine, the Java Virtual Machine (JVM), they can be executed on every platform that has the Java Runtime Environment (JRE) installed. As Sun Microsystems likes to put it, Java is a language you can: write once, run everywhere.

2. The amount of APIs (Application Program Interfaces) available to a programmer. These APIs provide support for a rich set of data structures and allow programs to communicate with external systems such as databases, web-services, etc.

For the area of cryptography, Java offers the framework Java Cryptography Architecture (JCA). It was designed around a "provider" architecture that adheres to the principles of implementation independence, interoperability and extensibility. The standard Sun Microsystems provider already gives access to some of the most common cryptographic techniques, but the programmer can always install (or program) alternative providers that enlarges the scope of the API to new algorithms and/or implementations.

An example of such alternative provider is the Bouncy Castle JCA provider. It makes accessible a richer set of cryptographic algorithms, such as the ones based on elliptic curves. Elliptic Curve Cryptography (ECC) relies on the difficulty of solving the Elliptic Curve Discrete Logarithm Problem. Comparing with older algorithms, ECC offers the same level of security with much smaller keys, which can lead to less computational requirements.

But the adoption of Java might be problematic for some emergent cryptographic algorithms, such as those based on bilinear pairings, where efficiency becomes a major concern. In fact, programs written in Java are recognisably less efficient than those written in compiled languages like C or C++. It is not clear whether pure Java implementations or those implementations remain usable in real applications.

1.2 Objectives and Contributions

In this thesis, it is intended to analyse the usage of the Java as the target programming language for cryptographic modules. This study is spread along different perspectives, ranging from the applicability and validation of code against existing cryptography standards, to the measure of efficiency in computationally demanding techniques. More concretely, it is intended to:

- Evaluate, for a concrete ”industrial sized” cryptographic API, the extent to which the code follows the prescriptions made by the relevant cryptographic standards
- Check the viability of adopting Java for supporting computationally demanding cryptographic techniques, such as those based on bilinear pairings.

1.3 Document Structure

The rest document is organised as follows: in Chapter 2, is provided a brief overview of some of the available cryptography standards. In Chapter 3, is presented the Bouncy Castle JCA provider and investigated, for selected techniques, the extent to which the code conforms with the relevant standards. Chapter 4 is devoted to the presentation of a pure Java implementation of an Identity Based Encryption scheme - a technique based on bilinear pairings that brings to light the efficiency limitations of Java. In Chapter 5 is proposed a solution to overcome those efficiency limitations, by making available directly to the JVM selected elliptic curve operations. A discussion and direction for further work conclude the thesis.

1.4 Technical specifications

All the tests presented in this document were made on a computer with the following specifications:

- Ubuntu 8.04
- 2 GB RAM
- Intel T2300 Dual Core (1,66 GHz)
- C++ compiler: GCC
- Java version: 1.6.0.06

The file used in all tests to encrypt/decrypt, sign/verify is a 70 KB text file.

2

Overview of cryptographic standards

This chapter presents an overview of some publicly available cryptographic standards. The choice of the covered standards intends to put in evidence the difference of their scopes: Common Criteria and FIPS-140 are targeted to security evaluation of products. On the other hand, the RSA's PKCS family of standards and IEEE P1363 are oriented to be adopted as guidelines for the actual coding of the techniques, giving detailed descriptions of them and other implementations details.

Occasionally, during the presentation of the standards, description of particular techniques will be highlighted, namely the RSA encryption scheme from the PKCS#1 and the ECDSA signature scheme from P1363. These descriptions will support the conformance analysis to be performed in Chapter 3.

2.1 Common Criteria

Common Criteria (CC) [1, 2, 3, 4] for Information Technology Security Evaluation is an international standard for computer security.The goal is to provide an evaluation of the security capabilities of products in the area of information technologies.

Through an independent evaluation of a product's ability to be in conformity with security standards, the CC helps customers in the decision making process. They can have more confidence in the security of products evaluated by CC. Security demanding and dependent customers, such as the United States Federal Government, are increasingly requiring the CC certification on the products they consider purchase. Because the requirements for CC certification are public, vendors can target security needs to achieve CC certification. [5]

The standard is adopted by several nations (among others: Canada, France, Germany, United Kingdom, United States, Australia, New Zealand, Finland, Greece, Israel, Spain) and membership continues to expand. The standard has been renamed to Common Criteria Recognition Arrangement (CCRA) in the year 2000.

In the evaluation of a products security, there are two major issues: the clients needs and the products capabilities.To represent them, CC uses two key concepts: protection profiles and evaluation assurance levels. Next, the essential concepts of the standard are briefly presented.

Target of evaluation

A Target of Evaluation (TOE) is product or system that is the subject of the evaluation.

Protection Profiles

A Protection Profile (PP) defines a set of security requirements for a type of product (such as operating systems, databases, firewalls, etc.).

By making public required security features for product families, the Common Criteria allows products to state conformity to a relevant protection profile. During CC evaluation, the product is tested against a specific protection profile, making sure the product meets the requirements. If it does, a client has knowledge of the product's security features and can decide if these features meet its needs. The client can also compare the security features of validated products.

Evaluation Assurance Level

Evaluation Assurance Level (EAL) is an indication of the level of conformance to the vendor claims, that a products reveals. It is not an indication of security capabilities of the product, but an independent assessment based on tests against the vendor claims.

	EAL designation
EAL1	Functionally Tested
EAL2	Structurally Tested
EAL3	Methodically Tested & Checked
EAL4	Methodically Designed, Tested & Reviewed
EAL5	Semi-formally Designed & Tested
EAL6	Semi-formally Verified Design & Tested
EAL7	Formally Verified Design & Tested

Table 2.1: Evaluation Assurance Levels

source: [6]

Security Target

Security Target (ST) is a document that identifies the security properties of the TOE. The ST includes an overview of the product, potential security threats, detailed information on all security features and any claims of conformity against a PP at a specified EAL.

Examples of Security Targets (Product Specific)

- Oracle Database Management System
- Lucent, Cisco, Check Point Firewalls

Information Technologies (IT) security requirements

- Functional Requirements - define the security behaviour of the product. Once implemented, they become security functions. Examples:
  1. Identification & authentication
  2. User data protection
- Assurance Requirements - the purpose is to establish confidence in security functions:
  - Correctness of implementation
  - Effectiveness in satisfying security objectives

  Examples:
  1. Testing
  2. Vulnerability Analysis

The Certification Process

Product certification aims to provide customer a level a trust, letting them know that a product went through a reliable, objective and globally accepted process.

In the CC case, to have a product certificated the vendor must first specify a ST.Then, it must submit the ST to an accredited testing laboratory where the evaluation process will take place. The laboratory then tests the product to verify what is stated in the ST. If the evaluation is successful, an official certification of the product(s) against a specific PP at a specified EAL is released.

2.2 Federal Information Processing Standards

Federal Information Processing Standards (FIPS) publications are developed by the National Institute of Standards for Technology (NIST) for use by the government. These publications regulate the requirements for government security. FIPS validation assures users that a given product has passed thorough testing by an accredited third party lab and is suitable to secure sensitive information.

FIPS 140

FIPS 140 Publication Series coordinate the requirements and standards for cryptography modules, both hardware and software components. This validation is demanded by the Federal Government of the United States of America for the purchase of products implementing cryptography.

What is a cryptographic module? The set of hardware, software, and/or firmware that implements approved security functions (including cryptographic algorithms and key generation) and is contained within the cryptographic boundary (the physical bounds of a cryptographic module).

FIPS 140-2 [7] identifies eleven areas for a cryptographic module:

Cryptographic Module Specification
Cryptographic Module Ports and Interfaces
Roles, Services and Authentication
Finite State Model
Physical Security
Operational Environment
Cryptographic Key Management
Self Tests
Design Assurance
Mitigation of Other Attacks
Electromagnetic Interference/Electromagnetic Compatibility (EMI/EMC)

To evaluate the quality level of each one of the eleven areas, FIPS has four distinct levels of security, from 1 to 4 (1 is the lowest). In the end, an overall rating is assigned.

Security Level 1 *Security Level 1 provides the lowest level of security. Basic security requirements are specified for a cryptographic module (e.g., at least one Approved algorithm or Approved security function shall be used). No specific physical security mechanisms are required in a Security Level 1 cryptographic module beyond the basic requirement for production-grade components. An example of a Security Level 1 cryptographic module is a personal computer (PC) encryption board. 1 Security Level 1 allows the software and firmware components of a cryptographic module to be executed on a general purpose computing system using an unevaluated operating system. Such implementations may be appropriate for some low-level security applications when other controls, such as physical security, network security, and administrative procedures are limited or nonexistent. The implementation of cryptographic software may be more cost-effective than corresponding hardware-based mechanisms, enabling organisations to select from alternative cryptographic solutions to meet lower-level security requirements.*[7]

Security Level 2 *Security Level 2 enhances the physical security mechanisms of a Security Level 1 cryptographic module by adding the requirement for tamper-evidence, which includes the*

use of tamper-evident coatings or seals or for pick-resistant locks on removable covers or doors of the module. Tamper-evident coatings or seals are placed on a cryptographic module so that the coating or seal must be broken to attain physical access to the plaintext cryptographic keys and critical security parameters (CSPs) within the module. Tamper-evident seals or pick-resistant locks are placed on covers or doors to protect against unauthorised physical access. Security Level 2 requires, at a minimum, role-based authentication in which a cryptographic module authenticates the authorisation of an operator to assume a specific role and perform a corresponding set of services. [7]

Security Level 3 *In addition to the tamper-evident physical security mechanisms required at Security Level 2, Security Level 3 attempts to prevent the intruder from gaining access to Critical Security Parameters(CSP) held within the cryptographic module. Physical security mechanisms required at Security Level 3 are intended to have a high probability of detecting and responding to attempts at physical access, use or modification of the cryptographic module. The physical security mechanisms may include the use of strong enclosures and tamper detection/response circuitry that zeroizes all plaintext CSPs when the removable covers/doors of the cryptographic module are opened. Security Level 3 requires identity-based authentication mechanisms, enhancing the security provided by the role-based authentication mechanisms specified for Security Level 2. A cryptographic module authenticates the identity of an operator and verifies that the identified operator is authorised to assume a specific role and perform a corresponding set of services. Security Level 3 requires the entry or output of plaintext CSPs (including the entry or output of plaintext CSPs using split knowledge procedures) be performed using ports that are physically separated from other ports, or interfaces that are logically separated using a trusted path from other interfaces. Plaintext CSPs may be entered into or output from the cryptographic module in encrypted form (in which case they may travel through enclosing or intervening systems).*[7]

Security Level 4 *provides the highest level of security defined in this standard. At this security level, the physical security mechanisms provide a complete envelope of protection around the cryptographic module with the intent of detecting and responding to all unauthorised attempts at physical access. Penetration of the cryptographic module enclosure from any direction has a very high probability of being detected, resulting in the immediate zeroization of all plaintext CSPs. Security Level 4 cryptographic modules are useful for operation in physically unprotected environments. Security Level 4 also protects a cryptographic module against a security compromise due to environmental conditions or fluctuations outside of the module's normal operating ranges for voltage and temperature. Intentional excursions beyond the normal operating ranges may be used by an attacker to thwart a cryptographic module's defences. A cryptographic module is required to either include special environmental protection features designed to detect fluctuations and zeroize CSPs, or to undergo rigorous environmental failure testing to provide a reasonable assurance that the module*

will not be affected by fluctuations outside of the normal operating range in a manner that can compromise the security of the module.[7]

Validation Process

The validation process of FIPS 140 is described in the following steps (see figure 2.1):

- The vendor submits documentation and product for testing
- An accredited laboratory tests the product against the FIPS 140-2 Derived Test Requirements.
- The laboratory submits a draft certification report to NIST/CSE (CSE stands for Communications Security Establishment and is the Canadian counterpart to NIST).
- NIST/CSE elaborates questions on the certification report.
- Once these questions have been resolved with NIST/CSE, a FIPS 140 certificate is issued by NIST/CSE.
- The new certificate is made public to the Internet on the NIST FIPS 140-1 and FIPS 140-2 Cryptographic Modules Certification List web page.

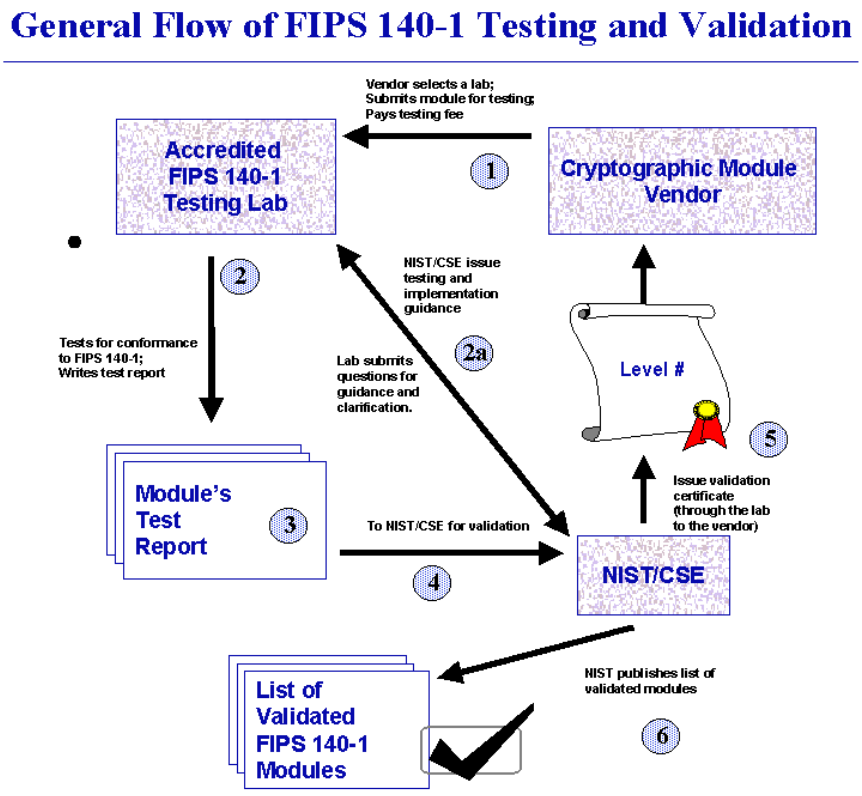

Figure 2.1: FIPS validation process

source:http://csrc.nist.gov.

Other cryptographic standards issued by FIPS

The FIPS series include several other standards directly concerned with cryptography. In particular, the standards that define important cryptographic techniques should be mentioned, such as the DES or AES symmetric ciphers, the SHA family of hash functions or the DSA signature scheme. Some of the relevant publications are:

- FIPS PUB 46-3 Data Encryption Standard (DES), 1999.
- FIPS PUB 81, DES Modes of Operation, 1980.
- FIPS PUB 180-2 Secure Hash Standard (SHS),2002, defines the Secure Hash Algorithm (SHA) family.
- FIPS PUB 186-2 Digital Signature Standard (DSS), 2000.
- FIPS PUB 196 Entity Authentication Using Public Key Cryptography, 1997.
- FIPS PUB 197 Advanced Encryption Standard (AES), 2001.
- FIPS PUB 198 The Keyed-Hash Message Authentication Code (HMAC), 2002 [8].

Relation between FIPS 140 and Common Criteria

FIPS 140 requires CC evaluations for the underlying operating system against specific protection profiles for level 2 (EAL 2), level 3 (EAL 3) and level 4 (EAL 4) [9, 7].

2.3 Public Key Cryptography Standards

The Public-Key Cryptography Standards (PKCS) are specifications produced by RSA Laboratories for the purpose of accelerating the deployment of public-key cryptography. PKCS defines various standards called PKCS#1 through #15. The covered areas are wide:

- RSA encryption and signature schemes — *PKCS #1 v2.1: RSA Cryptography Standard* [10].
- Key agreement — *PKCS #3 v1.4: Diffie-Hellman Key-Agreement Standard* [11]
- Password-based key-generation functions and encryptions schemes — *PKCS #5 v2.1: Password-Based Cryptography Standard* [12]
- Private-key information syntax and exchange — *PKCS #8: Private-Key Information Syntax Standard* [13] and *PKCS #12 v1.0: Personal Information Exchange Syntax* [14]
- Certificate request syntax and selected attribute types — *PKCS #10: Certification Request Syntax Standard* [15] and *PKCS #9: Selected Attribute Types* [16]

- Cryptographic token API and information syntax — *PKCS #11: Cryptographic Token Interface Standard* [17] and *PKCS #15: Cryptographic Token Information Format Standard* [18]

In the next section, *PKCS #1: RSA Cryptography Standard* will be the focus, as it will be the subject of a case study to be presented in Chapter 3.

2.3.1 PKCS #1 — RSA Cryptography Standard

As stated before, *PKCS #1 v2.1* comprises an encryption and a signature scheme. Only the encryption scheme will be highlighted.

Key types

To types of keys are employed in the primitive and schemes defined in this standard: *RSA public key* and *RSA private key*. Combined, they form a *RSA key pair*.

This specification supports "multi-prime" RSA where the modulus may have more than two prime factors. "Multi-prime" brings the benefit lower computational cost for the decryption and signature primitives, provided that the CRT (Chinese Remainder Theorem) is used. Quisquater and Couvreur [19] observed the benefit of applying the Chinese Remainder Theorem to RSA operations.

A *RSA public key* consists of two components:

n the RSA modulus, a positive integer.

e the RSA public exponent, a positive integer.

A *RSA private key* can have two different representations:

1. the pair (n, d), where the components have the following meanings:

   n the RSA modulus, a positive integer

   d the RSA private exponent, a positive integer

2. The second representation consists of a quintuple $(p, q, dP, dQ, qInv)$ and a (possibly empty) sequence of triplets (r_i, d_i, t_i), $i = 3, \ldots, u$, one for each prime not in the quintuple, where the components have the following meanings:

   p the first factor, a positive integer

   q the second factor, a positive integer

   dP the first factor's CRT exponent, a positive integer

   dQ the second factor's CRT exponent, a positive integer

qInv the (first) CRT coefficient, a positive integer

r_i the i^{th} factor, a positive integer

d_i the i^{th} factor's CRT exponent, a positive integer

t_i the i^{th} factor's CRT coefficient, a positive integer

Data conversion primitives

This standard defines two data conversion primitives:

- I2OSP — Integer-to-Octet-String primitive. Converts a nonnegative integer to an octet string of a specified length. an octet string is an ordered sequence of octets (eight-bit bytes).
- OS2IP — Octet-String-to-Integer primitive. Converts an octet string to a nonnegative integer.

Encryption and decryption primitives

Encryption and decryption primitives are operations that allow a plaintext to be transformed in ciphertext and *vice-versa*, respectively.

RSA defines:

- RSAEP - encryption primitive. Given RSA public key (n, e) and the message representative m, computes $c = m^e \mod n$.
- RSADP - decryption primitive. Given RSA private key K, where K has one of the following forms:

  1. a pair (n, d).
  2. a quintuple $(p, q, dP, dQ, qInv)$ and a possible empty sequence of triplets $(r_i, d_i, t_i), i = 3, ..., u$.

  must compute:

  1. If the first form (n, d) of K is used, $m = c^d \mod n$.
  2. If the second form $(p, q, dP, dQ, qInv)$ and (r_i, d_i, t_i) of K is used:
     (a) Let $m1 = c^{dP} \mod p$, $m2 = c^{dQ} \mod q$.
     (b) If $u > 2$, let $m_i = cd_i \mod r_i, i = 3 \ldots u$.
     (c) Let $h = (m1 - m2) \cdot qInv \bmod p$.
     (d) Let $m = m2 + q \cdot h$.
     (e) If $u > 2$, let $R = r_1$ and for $i = 3 \ldots u$ do:
        i. Let $R = R \cdot r_{i-1}$.
        ii. Let $h = (m_i - m) \cdot t_i \pmod{r_i}$.
        iii. Let $m = m + R \cdot h$.

RSA encryption schemes

An encryption scheme consists of an encryption operation followed by a decryption operation. A encryption operation produces a ciphertext from a initial message with a recipient's RSA public key. On the other hand, the decryption operation recovers the message from the ciphertext by using the corresponding RSA private key.

Two encryption schemes are described in PKCS #1 v2.1:

- RSAES-OAEP
  - RSAES-OAEP-ENCRYPT — Options: Hash function and the mask generation function (MGF).

    Input: (n, e) recipient's RSA public key (k denotes the length in octets of the RSA modulus n); M message to be encrypted, an octet string of length mLen, where $mLen \leq k - 2hLen - 2$; L optional label to be associated with the message (the default value for L, if L is not provided, is the empty string)

    Output: C ciphertext, an octet string of length k.

    RSAES-OAEP-ENCRYPT is comprised by the stages:

    1. Length checking. The length of the input parameters is validated.
    2. EME-OAEP encoding (see bellow).
    3. RSA encryption
       (a) Convert the encoded message to an integer message representative m (using data conversion primitive OS2IP).
       (b) Apply the RSAEP encryption primitive to the RSA public key (n, e) and the message representative m to produce an integer ciphertext representative c
       (c) Convert the ciphertext representative c to a ciphertext C of length k octets (using data conversion primitive I2OSP)
    4. Output the ciphertext C.

  - RSAES-OAEP-DECRYPT — Options: Hash function and the mask generation function (MGF).

    Input: K recipient's RSA private key (k denotes the length in octets of the RSA modulus n); C ciphertext to be decrypted, an octet string of length k, where $k \geq 2hLen + 2$; L optional label whose association with the message is to be verified (the default value for L, if L is not provided, is the empty string).

    Output: M message, an octet string of length mLen, where $mLen \leq k - 2hLen - 2$.

    RSAES-OAEP-DECRYPT is comprised by the stages:

    1. Length checking. The length of the input parameters is validated.

2. RSA decryption
   (a) Convert the ciphertext C to an integer ciphertext representative c (using data conversion primitive OS2IP)
   (b) Apply the RSADP decryption primitive to the RSA private key K and the ciphertext representative c to produce an integer message representative m.
   (c) Convert the message representative m to an encoded message of length k octets.
3. EME-OAEP decoding (where message M is extracted).
4. Output the message M.

- RSAES-PKCS1-v1.5

RSAES-OAEP is recommended for new applications; RSAES-PKCS1-v1.5 should only be used for compatibility with existing applications.

RSA padding methods

When using block cipher algorithms, padding is used to make the plaintext to be encrypted match the block size. This is done by adding a padding string.

PKCS #1 v2.1 provides two message padding methods:

- EME-OAEP
  - EME-OAEP encoding.
  - EME-OAEP decoding.
- EME-PKCS1-V1.5

EME-OAEP encoding:

1. If the label L is not provided, let L be the empty string. Let $lHash = Hash(L)$, an octet string of length hLen.
2. Generate an octet string PS consisting of $k - mLen - 2hLen - 2$ zero octets. The length of PS may be zero.
3. Concatenate $lHash$, PS, a single octet with hexadecimal value $0x01$, and the message$M$ to form a data block DB of length $k - hLen - 1$ octets as $DB = lHash \parallel PS \parallel 0 \times 01 \parallel M$.
4. Generate a random octet string seed of length $hLen$.
5. Let $dbMask = MGF(seed, k - hLen - 1)$.
6. Let $maskedDB = DB \oplus dbMask$.

7. Let $seedMask = MGF(maskedDB, hLen)$.

8. Let $maskedSeed = seed \oplus seedMask$.

9. Concatenate a single octet with hexadecimal value $0x00$, maskedSeed, and *maskedDB* to form an encoded message *EM* of length k octets as $EM = 0 \times 00 \parallel maskedSeed \parallel maskedDB$.

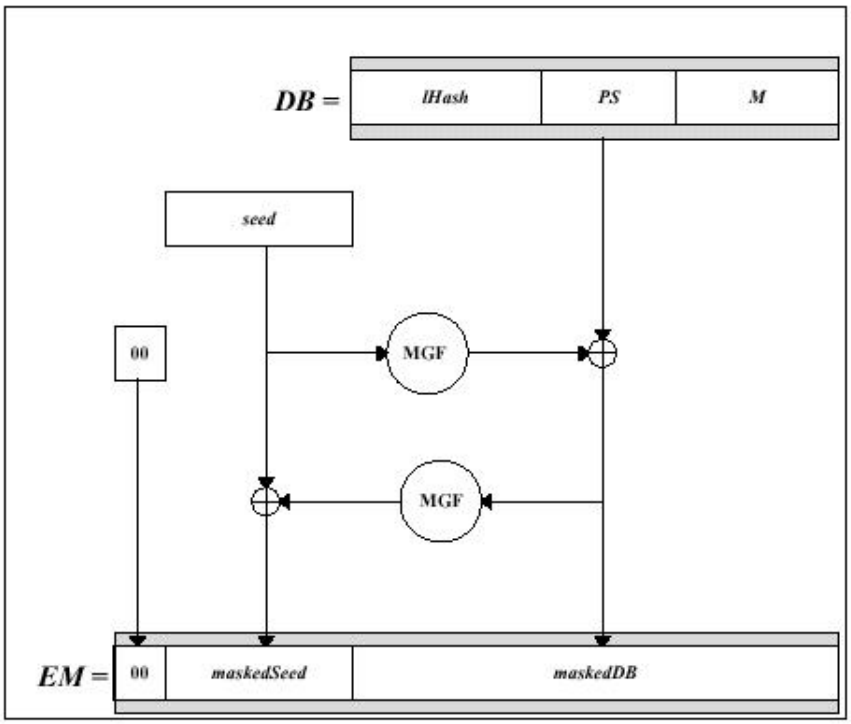

Figure 2.2: EME-OAEP encoding operation.
retrieved from PKCS#1 v2.1.

2.4 P1363

IEEE P1363 is a standard for public-key cryptography developed by the Institute of Electrical and Electronics Engineers (IEEE). The addressed areas:

- Traditional public-key cryptography –(IEEE Std 1363-2000 [20] and 1363a-2004 [21])
- Lattice-based public-key cryptography (P1363.1) –[22]
- Password-based public-key cryptography (P1363.2) –[23]

- Identity-based public-key cryptography using pairings –(P1363.3) (under development, first draft published in May 2008 [24]).

The focus of this section will be the *Traditional public-key cryptography*, as it us the most relevant for the purpose of this work. In fact, the *Identity-based public-key cryptography using pairings* would be relevant for the subject of Chapter 3, if it was not in such preliminary stage of development.

2.4.1 Traditional public-key cryptography

This specification covers a wide scope and includes the traditional areas of public-key cryptography:

- Key agreement;
- Signature schemes;
- Encryption schemes;

In P1363 [20] appeared a first set of such schemes and later, in P1363a [21] the set of covered techniques was extended and included additional information and corrections to the previous standard.

An interesting aspect of this standard is that it organises the above mentioned techniques according to the underlying mathematical *hard-problem*:

- Integer factorisation
- Discrete logarithm
- Elliptic curve discrete logarithm

In addition to cryptographic schemes presented in the body of the standard, it was included a rich set of *normative* and *informative* appendixes:

- Annex A *(Informative) Number-theoretic Background* –includes a comprehensive and complete background on the number-theoretic concepts and algorithms required by the techniques.
- Annex B *(Normative) Conformance* –provides implementers with a consistent language for claiming conformance with parts of this standard.
- Annex C *(Informative) Rationale* –this annex provides information about the standard in the form of questions and answers. It can be seen as a sort of Frequently Asked Questions.
- Annex D *(Informative) Security Considerations* –addresses security considerations for the cryptographic techniques that are defined in this standard.

- Annex E *(Informative) Formats* –P1363 does not specify how the mathematical and cryptographic objects are to be represented for communication or storage. This annex provides references to other relevant standards, such as the Abstract Syntax Notation 1 (ASN.1), and defines some recommended primitives for that purpose.

- Annex F *(Informative) Bibliography*.

In the following sections, a particular technique will be highlighted –The Elliptic Curve Digital Signature Algorithm –together with the supporting mathematical operations (elliptic curve operations). Again, this is motivated by the fact that this technique will be used in the case study presented in Chapter 3.

2.4.2 Elliptic Curve Cryptography Background

Because it is less widespread than other established public-key systems such as DSA and RSA, it is important to briefly introduced this technique.

Elliptic Curve Cryptography (ECC) is based on the analogue of the discrete logarithm problem in a finite field, but it uses a group of points on a elliptic curve (EC) defined over a finite field. The main advantage of ECC is that the best algorithm known for solving the elliptic curve discrete logarithm problem (ECDLP) takes exponential time. In comparison, the best algorithms known for solving the mathematical problem in which RSA or DSA are based upon, take sub-exponential time. In practical terms, this means ECC needs smaller parameters to achieve the same level of security. ECC can use keys five times smaller than the ones used by RSA and DSA. With smaller key sizes, it is possible to:

- perform faster computations.

- use less processing power, smaller storage space and bandwidth.

These features make ECC very attractive, particularly for constrained devices such as pagers, smart cards and cellphones.

The implementation of ECC requires some decisions to be made:

- Underlying finite field. The most common choices are prime fields ($\mathbb{F}_p$) or binary fields ($\mathbb{F}_{2^m}$).

- Type and parameters of the elliptic curve.

- Algorithms for the finite field arithmetic (addition of points and scalar multiplication).

These choices have a significant impact of overall ECC performance. Examples of schemes based on ECC (defined in [20]):

- Elliptic Curve Diffie-Hellman (ECSVDP-DH).

- Elliptic Curve Digital Signature Algorithm (ECSP-DSA and ECVP-DSA).

For further explanations on ECC, particularly about the mathematical foundations that support it, see [25].

2.4.3 ECC Arithmetic Operations

As mentioned, Annex A of P1363 [26] presents a fairly complete description of the mathematical concepts and algorithms required by the covered cryptographic techniques. In particular, it devotes a section to the description of algorithms for computing *elliptic curve operations*, namely the point-addition operation and scalar multiplication.

The attention will be restricted to curves over prime fields $\mathbb{F}_p$ (p is a prime number), defined by equations of the form:

$$y^2 = x^3 + ax + b \tag{2.1}$$

where a and b are the parameters of the curve. A point P is either O - the point at *infinity* - or a pair (xP, yP) of coordinates in $\mathbb{F}_p$ satisfying the defining equation.

Point Addition

The set of points of an elliptic curve form an algebraic group under the *point addition operation*. The addition of two points can be computed by the following algorithm (extracted from [26], *section A10.1*.

Input: a prime $p > 3$; coefficients *a, b* for an elliptic curve $E : y^2 = x^3 + ax + b$ modulo p; points $P_0 = (x_0, y_0)$ and $P_1 = (x_1, y_1)$ on E.

Output: the point $P_2 := P_0 + P_1$.

1. If $P_0 = O$ then output $P_2 \leftarrow P_1$ and stop.
2. If $P_1 = O$ then output $P_2 \leftarrow P_0$ and stop.
3. If $x_0 \neq x_1$ then
   (a) set $\lambda \leftarrow (y_0 - y_1)/(x_0 - x_1) \mod p$.
   (b) go to step 7.
4. If $y_0 \neq y_1$ then output $P_2 \leftarrow O$ and stop.
5. If $y_1 = 0$ then output $P_2 \leftarrow O$ and stop.
6. Set $\lambda \leftarrow 3x_1^2 + a)/(2y_1) \mod p$
7. Set $x_2 \leftarrow \lambda^2 - x_0 - x_1 \mod p$
8. Set $y_2 \leftarrow (x_1 - x_2)\lambda - y_1 \mod p$

9. Output $P_2 \leftarrow (x_2, y_2)$

The above algorithm requires 3 or 4 modular multiplications and a modular inversion. To subtract the point $P = (x, y)$, one adds the point $-P = (x, -y)$.

Scalar Multiplication

Elliptic Scalar Multiplication is the process of obtaining an elliptic curve point P_1, by multiplying an integer n by another curve point P_0. Both points belong to the same curve. The following algorithm is extracted from [26],*section A10.3*.

Input: an integer n and an elliptic curve point P. Output: the elliptic curve point nP.

1. If $n = 0$ then output O and stop.
2. If $n < 0$ then set $Q \leftarrow (-P)$ and $k \leftarrow (-n)$, else set $Q \leftarrow P$ and $k \leftarrow n$.
3. Let $h_l h_l - 1 \ldots h_1 h_0$ be the binary representation of $3k$, where the most significant bit h_l is 1.
4. Let $k_l k_l - 1 \ldots k_1 k_0$ be the binary representation of k.
5. Set $S \leftarrow Q$.
6. For i from $l - 1$ down to 1 do
   (a) Set $S \leftarrow 2S$.
   (b) If $h_i = 1$ and $k_i = 0$ then compute $S \leftarrow S + Q$ via the point addition algorithm (for prime case or binary case, as applies).
   (c) If $hi = 0$ and $k_i = 1$ then compute $S \leftarrow S - Q$ via the point addition algorithm (for prime case or binary case, as applies).
7. Output S.

2.4.4 Elliptic Curve Digital Signature Algorithm (ECDSA)

Elliptic Curve Digital Signature Algorithm (ECDSA) is a digital signature algorithm defined over an elliptic curve. It was first proposed in 1992 by Scott Vanstone [27] and in the following years accepted as a standard by organisations such as the International Standards Organisation (ISO) or the American National Standards Institute (ANSI), but is also included on P1363. In this section, the algorithm will be briefly presented. EC domain parameters (q, a, b, r, G) will be mentioned as input, where:

q is the prime number that defines the finite field.

a and b are the curve parameters.

G is the base point (or generator).

r is the order of G (the smallest non-negative numbersuch that $rG = O$)

The FE2IP (Field Element to Integer Conversion Primitive) will also be mentioned. It is defined in P1363[20] and its purpose is to convert finite field elements to non-negative integers

Signature (ECSP-DSA)

Input:

- The EC domain parameters (q, a, b, r, G) associated with the key private key s.
- The signer's private key s.
- The message representative, which is an integer $f \geq 0$.

Output: The signature, which is a pair of integers (c, d), where $1 \leq c < r$ and $1 \leq d < r$.
The signature (c, d) shall be computed by the following or an equivalent sequence of steps:

1. Generate a key pair (u, V) with the same set of domain parameters as the private key s.
2. Let $V = (x_V, y_V)$ ($V \neq O$ because V is a public key).
3. Convert x_V into an integer i with primitive FE2IP.
4. Compute an integer $c = i \mod r$. If $c = 0$, then go to step 1.
5. Compute an integer $d = u^{-1}(f + sc) \mod r$. If $d = 0$, then go to step 1.
6. Output the pair (c, d) as the signature.

Verification (ECVP-DSA)

Input:

- The EC domain parameters (q, a, b, r, G) associated with the key W.
- The signer's public key W.
- The message representative, which is an integer $f \geq 0$.
- The signature to be verified, which is a pair of integers (c, d).

Output: "Valid" if f and (c, d) are consistent, given the key and the domain parameters; "invalid" otherwise.

The output shall be computed by the following or an equivalent sequence of steps:

1. If c is not in $[1, r-1]$ or d is not in $[1, r-1]$, output "invalid" and stop.
2. Compute integers $h = d^{-1} \mod r$; $h_1 = fh \mod r$; $h2 = ch \mod r$.
3. Compute an elliptic curve point $P = h_1G + h_2W$. If $P = O$, output "invalid" and stop; otherwise, $P = (xP, yP)$.
4. Convert the field element xP to an integer i with primitive FE2IP.
5. Compute an integer $c' = i \mod r$.
6. If $c' = c$, then output "valid"; else, output "invalid."

2.4.5 Conformance

Conformance claims made by an implementation are mere claims, unless their accuracy can be assured by other means. Such other means may include, for example, implementation validation or assignment of legal liability to the implementer claiming conformance. They are outside the scope of this standard.

2.5 Conclusion

This chapter presents an overview of some cryptography standards. It should be noted the considerable overlap and cross-referencing across the standards. For instance, the RSA encryption and signature schemes described in PKCS #1 reappear in P1363 (including the corresponding message encoding operations).

The Standards for Efficient Cryptography [28], devoted to the standardisation of ECC and which was also subject of study in this work, should also be mentioned. They were not included in this survey because its content significantly overlaps with P1363.

3

Study of a cryptographic API for Java

A cryptographic Application Programming Interface (API) enables a programmer to have at his disposal the implementation of cryptographic techniques. It is an abstraction layer that isolates the programmer from the code used on the actual implementation.

There is a wide range of cryptography APIs available. Here are a few examples:

- Java Cryptography Architecture
- Microsoft Cryptography API
- Linux Kernel Cryptographic API
- International Crypto API for GNU/Linux
- Bouncy Castle Crypto APIs

The scope of some of these API's is broader than others. Some are only available in one programming language, like C#, therefore restricting them to be used only in certain environments. Others are written in popular and multi-platform languages like Java, which widespreads its use to almost every systems. Some are open-source, which means its source code is publicly available and any programmer can make changes to it.

Bouncy Castle Crypto APIs will be the focus of this chapter. Why Bouncy Castle? Primarily because it is an open-source Java based API, that covers the most common and significant cryprographic implementations.

3.1 Java Cryptography Architecture

Java Cryptography Architecture (JCA) and the the Java Cryptographic Extension (JCE) constitute a Java framework that provides a cryptographic API. The difference between JCA and JCE is merely historic, so it is common for the names to be used interchangeably or even together as JCA/JCE.

JCA makes use of a provider architecture. In the context of JCA, a provider is a concrete implementation of a set of cryptographic services. A provider can offer different services and different providers can be used in a program. The provider architecture makes JCA independent of the cryptographic techniques and their implementations. This way, is possible to include and use implementations that are not supported by Sun's official JCA distribution, which makes JCA also extensible.

One of the main advantages of JCA is the separation between concepts and implementations. There are classes that represent a cryptographic concept, like the class Cipher for example. JCA enables the programmer to use a cipher without having to worry about the underlying details. They are all encapsulated. The programmer just has to decide the algorithm to be used (AES or DES, for example). These also means that applications can work with different cryptographic algorithms without modifying the code.

The Provider Architecture of JCA has the Engine Classes on its core. These classes provide an interface to the functionality of a cryptographic service. Examples of Engine Classes are:

MessageDigest generates an object that implements a message digest algorithm.

Signature generates an object for signature and verification.

CertificateFactory generates certificate, certification path and certificate revocation list objects.

Cipher generates an object for encryption and decryption.

Each Engine Class has a correspondent Service Provider Interface (SPI) class, which is an abstract class that defines the methods that cryptographic service providers must implement.Therefore, a concrete implementation of an algorithm must be a subclass of the corresponding SPI class and must implement all the abstract methods. With this provider based architecture, it possible to use several different cryptographic service providers. An application requests the type of object (such as Cipher) and the associated algorithm (RC4, for example) and an appropriate object is returned to it. It is possible to specify the providers preference order which is the order providers are searched for requested services when no specific provider is specified. It is also possible to explicitly select a specific provider. An example of a JCA service request is given in code excerpt 3.1.

It is possible to see how a programmer, having more than one provider installed on the system, can simultaneous use the providers, while the providers are totally independent of one another. It

```
 (....)
/* Get a cipher instance from the default provider
Cipher a = Cipher.getInstance("RC4");

/* Get the same cipher, but from the installed provider XPTO, which applies a
    different technique
Cipher b = Cipher.getInstance("RC4","XPTO");

(....)
```

Code excerpt 3.1: Example of JCA providers use

is also possible to see how the JCA offers the separation between concepts and implementations. The programmer just indicated the algorithm (RC4) and it gets an object representing it.

The use of JCA cryptographic services follows a pattern:

- Instance creation - through the method *getInstance*. The programmer specifies the algorithm it wants and possibly the provider.
- Initialisation - through the use of the *init* method.
- Use - through the use of each Engine Class methods: *upDate*, *doFinal*, *sign*, etc.

For detailed information on JCA, including instructions on the installation of providers, consult the "Java Cryptographic Architecture Reference Guide" [29]

3.2 Bouncy Castle Crypto APIs

Bouncy Castle Crypto APIs are developed by Legion of the Bouncy Castle (BC) team and have their website at `http://www.bouncycastle.org`. BC is an open-source project that offers a very complete API for cryptographic operations for the programming languages Java and C#.

Regarding the Java API, BC provides:

- A lightweight cryptographic API, which implements all the techniques supported by BC and that can be used in all environments, including those that not directly support JCA, like Java 2 Micro Edition (J2ME).
- A JCA/JCE provider that offers the additional infrastructure that connects JCA and the implementations available by the light-weight cryptographic API.
- A JCE *clean room* implementation, which is a rewriting of the original JCE methods that dates back to when the Sun JCE API was not available outside of the United States.
- Specific API's for several protocols, such as Online Certificate Status Protocol (OCSP) or the Transport Layer Security (TLS), that are outside the scope of this work.

```
//EXAMPLE 1
Cipher rsaCipher = Cipher.getInstance("RSA/NONE/PKCS1Padding","BC");

//EXAMPLE 2
Cipher rsaCipher = Cipher.getInstance("RSA/NONE/OAEPWithSHA1AndMGF1Padding","BC
    ");

// EXAMPLE 3
 Cipher rsaCipher = Cipher.getInstance("RSA/NONE/NoPadding","BC");
```

Code excerpt 3.2: Obtaining an RSA object

The focus of this chapter is the study of the JCA/JCE provider (together with the correspondent lightweight API). In direct comparison with the standard Sun's provider, BC offers a bigger number of cryptographic techniques. One of the most interesting features of BC is the support for techniques based on elliptic curves.

3.2.1 Installation

Bouncy Castle can be used as a JCA provider. To use Bouncy Castle Crypto APIs as a cryptographic provider in Java, there are two choices:

1. Dynamic installation: The provider is loaded at runtime through an instruction in the program code.
2. Static installation: Modify the Java Runtime Environment (JRE) options and make Bouncy Castle one of the JCA providers.

3.3 RSA Cipher Implementation

The purpose of the this section is to study the BC implementation of the RSA cipher, in particular if it's easy to check for conformance with PKCS#1.

The study will focus on the functional aspects of the implementation, leaving aside details such as the parameters specification or the Abstract Syntax Notation 1 (ASN.1) codifications.

3.3.1 Implementation

An RSA cipher object is obtained by using the *getInstance* method from the class Cipher. Examples of invocations follow.

On the first example, the padding specified in PKCS#1v1.5 is used; on the second the OAEP is used, as specified in the PKCS#1v2; on the third, no padding is used.

The JCA selection mechanism will select the appropriate class to satisfy the request. On BC, for RSA that class is org.bouncycastle.jce.provider.JCERSACipher which is a subclass of the abstract class javax.crypto.CipherSpi (through the class org.bouncycastle.jce.provider.WrapCipherSpi) and it will be responsible for supplying the required functionality for the cipher object.

JCERSACipher class acts as a stub between JCA and the required functionality from the BC lightweight API. In this class, the cipher object will be instantiated as an instance of a class from the API. In the RSA case, those classes are:

- org.bouncycastle.crypto.engines.RSACoreEngine - offers the basic functionality from the RSA primitives.
- org.bouncycastle.crypto.engines.RSAEngine offers the basic functionality from the RSA primitives with no padding. It's a stub for RSACoreEngine.
- org.bouncycastle.crypto.engines.RSABlindedEngine - implements the RSA Blinding on the decipher operation. The blinding operation is discussed further ahead.
- org.bouncycastle.crypto.encodings.OAEPEncoding - implements the padding OAEP.
- org.bouncycastle.crypto.encodings.PKCS1Encoding - implements the padding scheme from PKCS#1v1.5.

Because it is recommended by the latest PKCS version, RSA with OAEP will be discussed.

As required by PKCS#1, BC allows different versions of cryptographic hash functions to be used on the codification of the padding OAEP:

- SHA1
- SHA 224
- SHA256
- SHA384
- SHA512

This parameterisation is done on class OAEPParameterSpec.

Once all the parameterisations are complete, the cipher object will be instantiated as an object of class OAEPEncoding. The constructor of this class, in turn, receives an instance of the underlying primitive –an object of class RSABlindedEngine. In short, the JCERSACipher will be instantiated as:

```
// This code is simplified for the sake of illustration. The hash function can
    be specified. By omission, the SHA1 is used

cipher=new OAEPEncoding(new RSABlindedEngine());
```

Lets analyse the OAEPEncoding class. This class implements the EME-OAEP scheme by closely following the description present in PKCS#1. The implementation of support operations can be found in the class, such as:

- OS2IP conversion
- I2OSP conversion
- MGF1 - Mask Generation Function

Also, the encryption and the decryption operations:

- RSAES-OAEP-ENCRYPT - method encodeBlock
- RSAES-OAEP-DECRYPT - method decodeBlock

```
public byte[] encodeBlock(
        byte[]  in,
        int     inOff,
        int     inLen)
        throws InvalidCipherTextException
    {
        byte[]  block = new byte[getInputBlockSize() + 1 + 2 * defHash.length];

        // copy in the message

        System.arraycopy(in, inOff, block, block.length - inLen, inLen);

        // add sentinel

        block[block.length - inLen - 1] = 0x01;


        // as the block is already zeroed - there's no need to add PS (the >= 0
             pad of 0)

        // add the hash of the encoding params.

        System.arraycopy(defHash, 0, block, defHash.length, defHash.length);

        // generate the seed.

        byte[]  seed = new byte[defHash.length];

        random.nextBytes(seed);

        // mask the message block.
```

```
        byte[]  mask = maskGeneratorFunction1(seed, 0, seed.length, block.
            length - defHash.length);

        for (int i = defHash.length; i != block.length; i++)
        {
            block[i] ^= mask[i - defHash.length];
        }

        // add in the seed

        System.arraycopy(seed, 0, block, 0, defHash.length);


        // mask the seed.

        mask = maskGeneratorFunction1(
                        block, defHash.length, block.length - defHash.length,
                            defHash.length);

        for (int i = 0; i != defHash.length; i++)
        {
            block[i] ^= mask[i];
        }

        return engine.processBlock(block, 0, block.length);
    }
```

Code excerpt 3.3: encodeBlock method from class OAEPEncoding

Code excerpt 3.3 contains the code of the method encodeBlock as distributed in BC. All the steps contain comments from the programmers, which helps understanding all the phases. By analysing the code and comparing to what PKCS#1 establishes for EME-OAEP, as can be seen in figure 2.2, it is easy to establish a correlation.

The last line of the code fragment invokes the method processBlock of the variable *engine*. This variable is initialised with the cipher object passed to the constructor as a parameter. In this case, the cipher is an object from RSABlindedEngine.

As so, it would be expected to have RSAEP and RSADP implemented in the class RSABlindedEngine. But, when analysing the class RSABlindedEngine, it is possible to conclude that it does not implement the RSAEP and RSADP primitives described on PKCS#1. Those are implemented in the class RSACoreEngine. What happens in class RSABlindedEngine is that, before those primitives are called in the code, a blinding mechanism is implemented.

The blinding mechanism purpose is to protect the decryption primitive from ”timming attacks”, like those described in [30]. Here’s a brief overview of the blinding mechanism:

1. Generate a secret random number r between 0 and $n - 1$.

2. Compute $x' = xr^e \bmod n$, where e is the public exponent.

3. Compute $y' = (x')^d \bmod n$ with the ordinary RSA decryption primitive.

4. Compute $y = y'r^{-1} \bmod n$. The result is as expected because $r^{ed} = r \bmod n$.

Since the mechanism involves a secret x', its running time cannot be correlated with cryptogram x, so a timing attack would fail on gaining information regarding the private key.

Regarding the implementation of the primitives RSAEP and RSADP, they are implemented on the method processBlock of the class RSACoreEngine, as shown in code excerpt 3.4

```
public BigInteger processBlock(BigInteger input)
    {
        //DECRYPTION
        if (key instanceof RSAPrivateCrtKeyParameters)
        {
            //
            // we have the extra factors, use the Chinese Remainder Theorem -
                the author
            // wishes to express his thanks to Dirk Bonekaemper at rtsffm.com
                for
            // advice regarding the expression of this.
            //
            RSAPrivateCrtKeyParameters crtKey = (RSAPrivateCrtKeyParameters)key
                ;

            BigInteger p = crtKey.getP();
            BigInteger q = crtKey.getQ();
            BigInteger dP = crtKey.getDP();
            BigInteger dQ = crtKey.getDQ();
            BigInteger qInv = crtKey.getQInv();

            BigInteger mP, mQ, h, m;

            // mP = ((input mod p) ^ dP)) mod p
            mP = (input.remainder(p)).modPow(dP, p);

            // mQ = ((input mod q) ^ dQ)) mod q
            mQ = (input.remainder(q)).modPow(dQ, q);

            // h = qInv * (mP - mQ) mod p
            h = mP.subtract(mQ);
            h = h.multiply(qInv);
            h = h.mod(p);               // mod (in Java) returns the positive
                residual

            // m = h * q + mQ
```

```
            m = h.multiply(q);
            m = m.add(mQ);

            return m;
        }
        else
        { //ENCRYPTION
            return input.modPow(
                    key.getExponent(), key.getModulus());
        }
    }
```

Code excerpt 3.4: processBlock method from RSACoreEngine

It is easy to correlate what is implemented by the code with what PKCS#1 establishes to RSAEP and RSADP, as can be seen is section 2.3.1.

After studying the BC implementation of the RSA cipher, one can conclude that it almost matches the specification presented in PKCS#1. Nevertheless, it introduces a feature not present in PKCS#1: the blinding mechanism on the decryption phase, to protect against "timming attacks".

The study also demonstrated that in order to validate the implementation against a standard, it is necessary to consider a big number of interconnected classes, pinpointing the relevant parts.

3.4 ECDSA Signature

In this section the BC implementation of the digital signature ECDSA will be studied, in particular if it's easy to check for conformance with P1363. Again, the study will focus on the functional aspects of the implementation.

3.4.1 Implementation

An ECDSA object is obtained by calling the *getInstance* method from the engine class Signature.

```
// Getting an ECDSA object

//EXAMPLE 1
 Signature ecdsa = Signature.getInstance("ECDSA", "BC");

//EXAMPLE 2
ecdsa = Signature.getInstance("SHA256/ECDSA", "BC");
```

Code excerpt 3.5: Obtaining an ECDSA object

Code excerpt 3.5 shows an example of how to obtain ECDSA objects. The examples differ on the cryptographic hash function used:

- Example 1: SHA-1
- Example 2: SHA-256 (SHA-2)

Like in the RSA case, JCA selection mechanism will select the appropriate class to satisfy the request. In BC and for ECDSA, that class is ecDSA, a sub-class of JDKDSASigner which, in turn, is a sub-class of SignatureSpi –the abstract class responsible to provide all the required functionalities for the signature object. The ecDSA class connects the JCA mechanism to the BC lightweight API. Regarding ECDSA, the signature and verification operations are implemented in class org.bouncycastle.signers.ECDSASigner. This last class is invoked inside JDKDSASigner nested classes.

```
static public class ecDSA
        extends JDKDSASigner
    {
        public ecDSA()
        {

            super(new SHA1Digest(), new ECDSASigner());
        }
    }


    static public class ecDSA256
        extends JDKDSASigner
    {
        public ecDSA256()
        {
            super(new SHA256Digest(), new ECDSASigner());
        }
    }


    }
```

Code excerpt 3.6: JDKDSASigner nested classes

As can be seen in code excerpt 3.6, class ECDSASigner is used as a parameter to constructors of class JDKDSASigner, together with a class implementing a cryptographic hash algorithm. JDKDSASigner will compute the parts of the ECDSA algorithm related with the hash function and will use the class ECDSASigner to compute the signature and verification primitives, by calling the ECDSASigner methods *generateSignature* and *verifySignature*, respectively.

```
public BigInteger[] generateSignature(
        byte[] message)
    {
        BigInteger n = key.getParameters().getN();
        BigInteger e = calculateE(n, message);
```

```
        BigInteger r = null;
        BigInteger s = null;

        // 5.3.2
        do // generate s
        {
            BigInteger k = null;
            int        nBitLength = n.bitLength();

            do // generate r
            {
                do
                {
                    k = new BigInteger(nBitLength, random);
                }
                while (k.equals(ZERO));

                ECPoint p = key.getParameters().getG().multiply(k);

                // 5.3.3
                BigInteger x = p.getX().toBigInteger();

                r = x.mod(n);
            }
            while (r.equals(ZERO));

            BigInteger d = ((ECPrivateKeyParameters)key).getD();

            s = k.modInverse(n).multiply(e.add(d.multiply(r))).mod(n);
        }
        while (s.equals(ZERO));

        BigInteger[]  res = new BigInteger[2];

        res[0] = r;
        res[1] = s;

        return res;
    }

    public boolean verifySignature(
        byte[]       message,
        BigInteger  r,
        BigInteger  s)
    {
        BigInteger n = key.getParameters().getN();
        BigInteger e = calculateE(n, message);
```

```
        // r in the range [1,n-1]
        if (r.compareTo(ONE) < 0 || r.compareTo(n) >= 0)
        {
            return false;
        }

        // s in the range [1,n-1]
        if (s.compareTo(ONE) < 0 || s.compareTo(n) >= 0)
        {
            return false;
        }

        BigInteger c = s.modInverse(n);

        BigInteger u1 = e.multiply(c).mod(n);
        BigInteger u2 = r.multiply(c).mod(n);

        ECPoint G = key.getParameters().getG();
        ECPoint Q = ((ECPublicKeyParameters)key).getQ();

        ECPoint point = ECAlgorithms.sumOfTwoMultiplies(G, u1, Q, u2);

        BigInteger v = point.getX().toBigInteger().mod(n);

        return v.equals(r);
    }
```

Code excerpt 3.7: ECDSA signature and verification

Code excerpt 3.7 contains the signature and verification primitives. It is easy to correlate the steps of each method with what SEC establishes to ECDSA, as can bee seen in section 2.4.4.

ECDSA signature and verification primitives require arithmetic over an elliptic curve: point addition and scalar multiplication.

BC uses the class org.bouncycastle.math.ec.ECPoint to represent a point on a elliptic curve. Inside this class, it defines two nested classes:

- ECPointFp
- ECPointF2m

ECPointFp and ECPointF2m refere to operations on $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$, respectively.
Each of these classes implements these operations:

- add;
- subtract;

- negate;
- twice;

Lets focus on the addition and point doubling on a curve over $\mathbb{F}_p$, for example.

```
public ECPoint add(ECPoint b)
        {
            if (this.isInfinity())
            {
                return b;
            }

            if (b.isInfinity())
            {
                return this;
            }

            // Check if b = this or b = -this
            if (this.x.equals(b.x))
            {
                if (this.y.equals(b.y))
                {
                    // this = b, i.e. this must be doubled
                    return this.twice();
                }

                // this = -b, i.e. the result is the point at infinity
                return this.curve.getInfinity();
            }

            ECFieldElement gamma = b.y.subtract(this.y).divide(b.x.subtract(
                this.x));

            ECFieldElement x3 = gamma.square().subtract(this.x).subtract(b.x);
            ECFieldElement y3 = gamma.multiply(this.x.subtract(x3)).subtract(
                this.y);

            return new ECPoint.Fp(curve, x3, y3);
        }

public ECPoint twice()
        {
            if (this.isInfinity())
            {
                // Twice identity element (point at infinity) is identity
                return this;
            }
```

```
        if (this.y.toBigInteger().signum() == 0)
        {
            // if y1 == 0, then (x1, y1) == (x1, -y1)
            // and hence this = -this and thus 2(x1, y1) == infinity
            return this.curve.getInfinity();
        }

        ECFieldElement TWO = this.curve.fromBigInteger(BigInteger.valueOf
            (2));
        ECFieldElement THREE = this.curve.fromBigInteger(BigInteger.valueOf
            (3));
        ECFieldElement gamma = this.x.square().multiply(THREE).add(curve.a)
            .divide(y.multiply(TWO));

        ECFieldElement x3 = gamma.square().subtract(this.x.multiply(TWO));
        ECFieldElement y3 = gamma.multiply(this.x.subtract(x3)).subtract(
            this.y);

        return new ECPoint.Fp(curve, x3, y3, this.withCompression);
    }
```

Code excerpt 3.8: Addition on class ECPointFp

The operations are done by methods *add* and *twice*. Although P1363 represents the two operations together and the BC code splits them in two different operations, the presented operations can be successfully matched against those established in P1363, for elliptic curves arithmetics over $\mathbb{F}_p$, as can be seen in section 2.4.3.

A note on a safety issue: after studying the code related with operations with elliptic curve points, it becomes clear that a test to guarantee the two points belong to the same curve isn't being done, which can lead to errors if the parameterisations are incorrect. Regarding the scalar multiplication on a elliptic curve, it is implemented in classes:

- org.bouncycastle.math.ec.FpNafMultiplier
- org.bouncycastle.math.ec.WTauNafMultiplier

The two classes use different algorithms. Because FpNafMultiplier is the done used by default on BC, lets analyse it.

```
package org.bouncycastle.math.ec;

import java.math.BigInteger;

/**
```

```
 * Class implementing the NAF (Non-Adjacent Form) multiplication algorithm.
 */
class FpNafMultiplier implements ECMultiplier
{
    /**
     * D.3.2 pg 101
     * @see org.bouncycastle.math.ec.ECMultiplier#multiply(org.bouncycastle.
         math.ec.ECPoint, java.math.BigInteger)
     */
    public ECPoint multiply(ECPoint p, BigInteger k, PreCompInfo preCompInfo)
    {
        // TODO Probably should try to add this
        // BigInteger e = k.mod(n); // n == order of p
        BigInteger e = k;
        BigInteger h = e.multiply(BigInteger.valueOf(3));

        ECPoint neg = p.negate();
        ECPoint R = p;

        for (int i = h.bitLength() - 2; i > 0; --i)
        {
            R = R.twice();

            boolean hBit = h.testBit(i);
            boolean eBit = e.testBit(i);

            if (hBit != eBit)
            {
                R = R.add(hBit ? p : neg);
            }
        }

        return R;
    }
}
```

Code excerpt 3.9: ECC Multiplication on Bouncy Castle

Code excerpt 3.9 represents the FpNafMultiplier class on Bouncy Castle. By comparing the P1363 Scalar Multiplication steps, as presented in section 2.4.3, with the code from class FpNafMultiplier, it is possible to conclude that the method is correctly implemented in the class.

In the same code excerpt, it can be seen that one of the parameters of the method *multiply* is an object of the class PreCompInfo, although the object is not used. BC defines the interface PreCompInfo as a way to classes store precomputation data for multiplication.

After going through the BC ECDSA Signature implementation, it is possible to conclude that it follows what is established in the standard SEC and also (in the scalar multiplication case) on

P1363.

Another conclusion is that BC ECDSA does not use a big number of classes on the implementation of its core and the connections between those classes aren't difficult to assess. It is easy to follow the ECDSA implementation form the top class JDKDSASigner to classes like ECPoint that implement mathematical operations.

3.5 Performance Testing

To see how the Bouncy Castle Crypto APIs behaves in practical terms, some tests were conducted. It was decided to test one of the ECC based algorithms to check how a pure Java implementation of ECC behaves in practical terms. The chosen algorithm was ECDSA.

Bouncy Castle comes with a predefined set of elliptic curves. These curves come described in cryptographic standards because they were tested and proved to be efficient and safe to use in ECC. In this test, it was used the curve *Curve P-192*, as described in FIPS 186-2 [31] (see table 3.1).

Two different scenarios are presented in this section: single execution and multi-execution. The single execution scenario corresponds to a simple execution of the signature and verification operations. The multi-execution scenario corresponds to the repeated execution of the signature and verification operations, using the same parameters and keys, with the same message. The idea is to create a hypothetic scenario where the operation is repeatedly executed and to investigate if there are any performance gains.

curve parameter	value in hexadecimal
p	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFF (192 bits)
a	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFC
b	64210519E59C80E70FA7E9AB72243049FEB8DEECC146B9B1
Gx	188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012
Gy	07192b95ffc8da78631011ed6b24cdd573f977a11e794811

Table 3.1: Curve P-192 parameters

	Signature	Verification
1	79	56
2	51	63
3	109	103
4	99	142
5	82	85
6	115	111
7	110	68
8	82	121
9	94	104
10	97	88
Average	91,8	94,1

Table 3.2: ECDSA single execution times in milliseconds

	Signature	Verification
1	169	62
2	29	28
3	22	23
4	34	27
5	22	31
6	21	26
7	20	25
8	28	28
9	22	28
10	20	25
Average	38,7	30,3

Table 3.3: ECDSA multi-execution times in milliseconds

Bouncy Castle provider was dynamically installed using the package *bcprov-jdk15-139.jar*. This package is signed by Sun Microsystems.

Table 3.2 and 3.3 contains the measured times of single executions of ECDSA and multi-execution of ECDSA, respectively. Each scenario was repeated several times in order to reduce the influence of the randomised nature of the algorithms. One is able to conclude that in the multi-execution scenario there is a decrease of 57,8% on the Signature time and a decrease of 67,6% in the Verification time. The execution times do not consider input and output operations or represent the total times of execution of the programs. The only operations that were considered were those directly related with the algorithms of signature and verification.

So, in fact there are performance gains when ECDSA is repeatedly executed. This difference is mostly due to the class loading mechanism of Java, but a deeper study of this speedup falls out of the scope of this project.

4

Identity-Based Cryptography

The purpose of this chapter is to introduce an implementation in Java of Identity-Based Cryptography, in particular an Identity-Based Encryption (IBE) scheme. Implementations of Identity-Based Cryptography are usually based on the mathematical concept of bilinear pairings, so the basic concepts of bilinear pairings are also presented.

4.1 Identity-Based Encryption

Identity-Based Cryptography was introduced by Shamir [32] back on the year of 1984. The proposed concept is quite innovative. Instead of using the public key contained on a digital certificate for encryption and signature verification, information related with the identity of the user can be used as the public key. This means that the complexity involved with the Public Key Infrastructure (PKI) can be avoided.

A simplified outline of the steps involved in a IBE encryption scheme is presented. The involved parts are a Private Key Generator (PKG) which is a trusted third-party, the sender and the receiver.

Setup PKG generates its private key (known as master key) and public key.

Encryption Using the receiver's identity and the PKG's public key, the sender is able to encrypt the plaintext M and obtain the ciphertext C.

Private Key Extraction The receiver of the message authenticates itself to the PKG and receives its own private key associated with his identity. This identity can be, as mentioned earlier, a

string such as email.

Decryption When the receiver gets the ciphertext C from the sender, uses its private key to recover the original message M.

4.1.1 Advantages and drawbacks

As stated before, IBE can be used as an alternative to the traditional PKI which uses digital certificates. Many of the problems inherent with managing certificates can be eliminated. In IBE it is also possible to encrypt a message even before the person for whom the message is destinated has generated a key pair. The person that sends the message can include in the identification string a set of conditions that must be met by the receiver before the PKG issues the private key. On the other hand, IBE forces the existence of a PKG that generates the private keys. As a result, the PKG can decrypt or sign any messages. Because of this, the level of trust on this third party has to be absolute. This is called the key escrow problem . As a possible solution to this, Boneh and Franklin [33] proposed a solution that involves sharing the master secret key of the PKG between PKGs using Shamir's secret sharing technique [34]. The user would obtain partial private key shares from each one of them and then reconstruct the whole key. This solution isn't ideal since the multiple authentications to PKGs generate big communication and computational costs [35]. Another two drawbacks of the IBE scheme are the need of a secure channel between the user and the PKG, so the private key can be safely transmitted and the potential heavy workloads imposed on a single PKG, though a solution to this problem has been suggested by Gentry and Silverberg by the means of a hierarchical IBE. [36]. .

4.2 Bilinear pairings

Bilinear pairings over elliptic curves were initially used in cryptography as an attack that rendered supersingular curves insecure [37]. Later, it was found that the same technique can be employed in the construction of cryptographic schemes

Joux [38] proposed bilinear pairings to solve the problem of a three-party one-round key agreement that is secure against eavesdroppers [39]. This breakthrough called the attention of cryptographers to the use of pairings in other applications. Boneh and Franklin [33] proposed an identity-based encryption scheme using pairings and Boneh, Lynn and Shacham proposed a short signature scheme [40] also based on pairings.

This introduction is adapted from Menezes [39]. A bilinear pairing on (G_1, G_T) is a map:

$$\hat{e} : G_1 \times G_1 \rightarrow G_T \tag{4.1}$$

where n is a prime number, $G_1 = \langle P \rangle$ an additively-written group of order n with identity ∞ and G_T a multiplicatively-written group of order n with identity 1, such that the following

conditions hold:

1. *(bilinearity)* For all $R, S, T \in G_1, \hat{e}(R+S, T) = \hat{e}(R, T)\hat{e}(S, T)$ and $\hat{e}(R, S+T) = \hat{e}(R, S)\hat{e}(R, T)$
2. *(non-degeneracy)* $\hat{e}(P, P) \neq 1$
3. *(computability)* $\hat{e}$ can be efficient computed

Fromm the above properties, one can easily derive:

1. $\hat{e}(S, \infty) = 1$ and $\hat{e}(\infty, S) = 1$
2. $\hat{e}(S, -T) = \hat{e}(-S, T) = \hat{e}(S, T)^{-1}$
3. $\hat{e}(aS, bT) = \hat{e}(S, T)^{ab}$ for all $a, b \in \mathbb{Z}$
4. $\hat{e}(S, T) = \hat{e}(T, S)$
5. If $\hat{e}(S, R) = 1$ for all $R \in G_1$, then $S = \infty$

The security of many pairing-based protocols relies on the intractability of the Bilinear Decisional Diffie-Hellman Problem (BDHP):

- Let $\hat{e}$ be a bilinear pairing on (G_1, G_T). The Bilinear Decisional Diffie-Hellman Problem is the following: Given (P, aP, bP, cP) compute $\hat{e}(P, P)^{abc}$

4.2.1 Boneh and Franklin Scheme

Shamir [32] constructed an identity based signature (IBS) scheme, but the construction of a identity based encryption (IBE) scheme was left as an open problem. This problem was independently solved by Boneh and Franklin [33] and Cocks [41] in 2001.

The following mathematical description is adapted from Menezes [39]. The scheme proposed by Boneh and Franklin [33] uses a bilinear pairing $\hat{e}$ on (G_1, G_T) for which the BDHP is intractable and two hash functions:

$$H_1 : \{0,1\}^* \rightarrow G_1 \backslash \{\infty\} \tag{4.2}$$

$$H_2 : G_T \rightarrow \{0,1\}^l \tag{4.3}$$

where l is the bitlength of the plaintext. The PKG randomly selects an integer $t \in [1, n-1]$ and computes its public key $T = tP$, where P is a generator of G_1 Lets imagine an interaction between two people, Alice and Bob (Alice and Bob are typically used to describe cryptographic algorithms). Alice requests her private key d_a. The PKG creates Alice's identity string ID_A and

computes $d_a = tH_1(ID_A)$, delivering it to Alice through a secure channel. To encrypt a message $m \in \{0,1\}^l$ for Alice (using the basic Boneh-Franklin scheme [33]), Bob computes:

$$Q_A = H_1(ID_A) \tag{4.4}$$

$$r \in [1, n-1] \tag{4.5}$$

$$R = rP \tag{4.6}$$

$$c = m \oplus H_2(\hat{e}(Q_A, T)^r) \tag{4.7}$$

After this, the ciphertext (R, c) is transmitted to Alice. To decrypt the ciphertext, Alice computes

$$m = c \oplus H_2(\hat{e}(d_A, R)) \tag{4.8}$$

because:

$$\hat{e}(d_A, R) = \hat{e}(tQ_A, rP) = \hat{e}(Q_A, tP)^r = \hat{e}(Q_A, T)^r \tag{4.9}$$

Anyone, other than Alice, who wishes to recover m from (R, c) most compute $\hat{e}(Q_A, T)^r$ given (P, Q_A, T, R). This is an instance of the BDHP and precisely where the cryptographic strength of IBE resides.

The principles of Identity-Based Encryption can be extended to an Identity-Based Signature scheme, as described in *A Survey of Identity-Based Cryptography* [35].

4.3 IBE in Java

Owens, Duffy and Downling [42] have implemented the IBE system of Boneh and Franklin [33] scheme, entirely in Java. This is something new, because the previous known implementations of IBE were developed using C/C++. Shamus Software's MIRACL [43] is one of those implementations. On the website of the project [44], the source code is available, just as all the jar files needed to use the software. The program is designed as a JCA provider and because it implements a new Cipher class, the jar file that contains the project's compiled code is signed by Sun. This is a Sun's requirement to protect sensitive parts of the framework (see [29] for more informations).

The authors of the project used an existing elliptic curve cryptosystem implementation [45] and concentrated the efforts on the implementation of the bilinear pairing that IBE uses.

parameter	value in hexadecimal
p	D535ED880BA061BB05A34E83D82E6271B2ED7D8207C0EF5C9CE7F72A6BB-E5C1EF1F5956CB0BCB629D0F64608333C87662CEA3F08-D229BE8FA209861E66B3BBB (512 bits)
q	800000000000000000000000000000000020001
l	1AA6BDB101740C3760B469D07B05CC4E365D451514E69CB95FD7642DE-C2CBD00169F791958529BA7B6EF33BBC

Table 4.1: IBE parameters

4.3.1 Performance testing

To test this implementation, one hybrid cryptosystem was developed. The idea is to:

1. Use a symmetric key to cipher the data
2. Use the IBE encryption scheme to cipher the symmetric key

The selected algorithm for symmetric key cryptography is AES using 128 bits keys.

To create the test class, it was used the sample code suggested by the authors in the website [42], with some minor modifications. One of them is the hashing algorithm: SHA-256 was used. The bilinear pairing used was Tate (implemented in the class ModifiedTatePairing).

The algorithm used a super-singular curve of form $y^2 = x^3 + 1$. Table 4.1 contains the parameters related to the underlying field F_p according to the formula $p = lq - 1$. The modulus of p is 512 bits. These parameters were used in this test because they come by default in the IBE implementation of MIRACL, which will be addressed further ahead in this document. The parameters were supplied to the program by using in the code, the constructor *ModifiedTatePairing(BigInteger p, BigInteger q, BigInteger l)* available in the class *ModifiedTatePairing*.

The execution times do not consider input and output operations or represent the total times of execution of the programs. The only operations that were considered were those directly related with encryption and decryption.

Table 4.2 shows the times the IBE scheme in Java took to cipher and decipher an 128 bytes AES key in isolated executions. Table 4.3 contains the times obtained on a multi-execution scenario. The performance increase is not significant: 5,6% on the encryption stage and 3,1% on the decryption counterpart. The heavy workload that the calculations of pairings imposes doesn't allow higher performances even on a multi-execution scenario, where some optimisations are likely to appear, as seen in section 3.5.

Another interesting point are the values of the execution times. The average times obtained can be considered very high, although is important to bear in mind that the project is still in the early stages and many improvements can be done.

Results apart, the authors of the project should be praised for the initiative because is groundbreaking.

	Encryption	Decryption
1	5260	3454
2	5161	3416
3	5115	3438
4	5119	3447
5	5095	3424
6	5092	3419
7	5092	3390
8	5099	3459
9	5270	3476
10	5117	3456
Average	5142	3447,9

Table 4.2: IBE single execution times in milliseconds

	Encryption	Decryption
1	5021	3396
2	4845	3348
3	4811	3356
4	4855	3332
5	4773	3322
6	4843	3346
7	4796	3317
8	4843	3327
9	4846	3326
10	4849	3326
Average	4848,2	3339,6

Table 4.3: IBE multi-execution times in milliseconds

5

Searching for efficiency

Ever since the use of the Java language began to increase, its performance during heavy calculations has been questioned by many. Typically, scientific calculations require heavy computer work. These programs are normally developed in Fortran or C/C++. So, the use of Java to implement cryptographic algorithms that use heavy mathematical primitives, just as IBE does, can in fact raise some questions related with performance. The purpose of this chapter is not to discuss if and why Java is slower than other languages. That's still part of a debate in the programmers community. But for that matter, there are many investigations, for example [46, 47] that propose ways of improving the Java Virtual Machine (JVM) performance. In the following sections, will be presented tests made with implementations of cryptographic algorithms in C++ and a possibility of increasing Java based implementations performance using C++.

5.1 Testing with C++

It was decided to use cryptographic software based on C/C++ to conduct some tests. As stated earlier, the family C/C++ has a good reputation on computing hard calculations. Therefore, it was needed some cryptographic implementation preferably released as open-source, thus enabling the access to the source code. The choice was MIRACL, presented in the next section.

5.1.1 MIRACL

Shamus Software MIRACL [43] is a Big Number Library designed to support Big Number Cryptography. It has it's website at `http://www.shamus.ie`.

> It is primarily a tool for cryptographic system implementors. RSA public key cryptography, Diffie-Hellman Key exchange, DSA digital signature, they are all just a few procedure calls away (...) MIRACL now provides more support for conventional cryptography. The latest version implements the Advanced Encryption Standard (AES), Modes of Operation, and the new hashing standards SHA-160/256/384/512. *[43]*

The installation instructions can be found within the zip file that can be downloaded with all the source code. User and Reference manuals can also be found on the website.

MIRACL comes with ready to execute programs for the Windows platform and more than 25 example programs provided in both C/C++ versions. Two of these example programs are of special interest within scope of his work:

- ECDSA implementation
- IBE implementation

It was decided to do the tests with the C++ versions. No performance issues were involved in this decision.

The ECDSA implementation is organized like this (these comments may be found on the file's headers):

common.ecs contains the domain information (p, A, B, q, x, y), where A and B are curve parameters, (x, y) is a point of order q and p is the prime modulus (base field).

ecsgen.cpp generates one set of public and private keys in files public.ecs and private.ecs respectively.

ecsign.cpp signs a file whose name <file> is inputed by the user and outputs the signature to a file <file>.ecs. Need the private key file and common.ecs

ecsver.cpp This program verifies the signature given to a <file> in <file>.ecs generated by program ecsign. Needs the files public.ecs and common.ecs

Each one of this .cpp files is dependent of files:

big.cpp This class is used to represent big numbers.

ecn.cpp This class is used to implement elliptic curves and it's arithmetics.

The IBE scheme is organized as so (these comments may be found on the file's headers):

ibe_set.cpp Creates the files common.ibe and master.ibe.

common.ibe This file contains the parameters used on the IBE scheme.

- Size of prime modulus in bits
- Prime p
- Prime q (divides p+1)
- Point P - x coordinate
- Point P - y coordinate
- Point Ppub - x coordinate
- Point Ppub - y coordinate
- Cube root of unity in Fp2 - x component
- Cube root of unity in Fp2 - y component

master.ibe Contains the PKG master key.

ibe_ext.cpp This program extracts a private (secret) key from the identity string inputed by the user and stores it in the file private.ibe

ibe_enc.cpp Generates a random AES session key, and uses it to encrypt a file. Outputs ciphertext to <file>.ibe. The session key is IBE encrypted, and written to <file>.key. The user inputs a string that will be used to compute the public key, according to the IBE scheme.

ibe_dec.cpp Decrypts a file <file>.ibe. Finds the session key by IBE decrypting the file <file>.key. Uses this session key to AES decrypt the file. The IBE private key used is in private.ibe.

Besides big.cpp and ecn.cpp, the above .cpp files are dependent of:

zzn.cpp Definition of class ZZn (Arithmetic mod n), using Montgomery's Method for modular multiplication

zzn2.cpp Arithmetic over n^2

5.1.2 MIRACL results

The idea was to measure the performance times of MIRACL during:

- ECDSA signature generation
- ECDSA signature verification
- IBE encryption of a 128 bytes AES key
- IBE decryption of the same key

Tests were made with single executions and with multi-executions. Again, the execution times do not consider input and output operations or represent the total times of execution of the programs. The only operations that were considered were those directly related with the algorithms of signature and verification in the ECDSA case and encryption and decryption in the IBE case.

	Signature	Verification
1	13,5	14,3
2	13,4	14,2
3	13,4	14,0
4	13,3	14,0
5	13,5	16,6
6	13,6	14,2
7	20,2	14,2
8	13,6	14,1
9	8,1	14,4
10	13,5	14,0
Average	13,62	14,37

Table 5.1: MIRACL ECDSA single execution times in milliseconds

	Signature	Verification
1	13,6	14,2
2	13,2	14,1
3	13,6	13,1
4	8,0	8,4
5	8,2	8,5
6	8,0	8,5
7	8,0	8,5
8	8,0	8,5
9	8,0	8,6
10	8,1	8,4
Average	9,67	10,1

Table 5.2: MIRACL ECDSA multi-execution times in milliseconds

Table 5.1 shows the measured times of single executions of ECDSA. Table 5.2 contains the execution times measured for ECDSA in the multi-execution scenario. By analysing the data, it is possible to verify that in the multi-execution scenario there is a decrease of about 29% and 25% on the average time, for Signature and Verification, respectively.

	Encryption	Decryption
1	21,8	19,4
2	21,9	19,6
3	21,8	19,6
4	21,6	19,8
5	21,9	19,5
6	21,7	19,6
7	21,7	19,7
8	21,6	19,6
9	21,6	19,6
10	21,7	19,6
Average	21,73	19,6

Table 5.3: MIRACL IBE single execution times in milliseconds

	Encryption	Decryption
1	21,6	19,5
2	21,6	19,4
3	21,6	19,5
4	21,7	19,4
5	21,6	19,4
6	21,6	19,4
7	21,6	20,5
8	21,6	19,9
9	21,6	19,4
10	21,6	19,5
Average	21,6	19,4

Table 5.4: MIRACL IBE multi-execution times in milliseconds

Table 5.3 shows the measured times of single executions of IBE using MIRACL. Table 5.4 has the times measured for the same algorithm, but for multi-executions achieved using a cycle in the source code. By analysing both tables, it can been seen that there is almost no variation of execution times in the multi-executions scenario.

Again, without going into details that are outside the scope of this work, it is fair to conclude that the heavy calculations on the IBE scheme,specially those related with pairings, don't allow a increase of performance even when in a multi-execution scenario, contrarily to the ECDSA case where there is a decrease of times when the algorithm is repeatedly executed.

5.1.3 Comparison with Java results

After finishing the tests, it is clear that the MIRACL implementation is much more faster than the one written in Java. The results displayed in this section are only those regarding single executions. However, there are somethings to bear in mind when analysing the results:

- MIRACL is optimised. It is designed to be fast, to achieve good performances.
- The IBE implementation in JAVA is still at the early stages. The authors [42] emphasise the fact that there's plenty of room for improvements.

Figure 5.1 shows how MIRACL is faster than Bouncy Castle. Nevertheless, Bouncy Castle can still be considered as having a reasonable performance. Figure 5.2 is quite clear about the differences between the performances of the IBE implementations. While encrypting and decrypting an 128 bits AES key is almost instantaneous on MIRACL, it takes a few seconds to do it with the IBE Java implementantion. In fact, the IBE Java implementation results leave a good room for improvements.

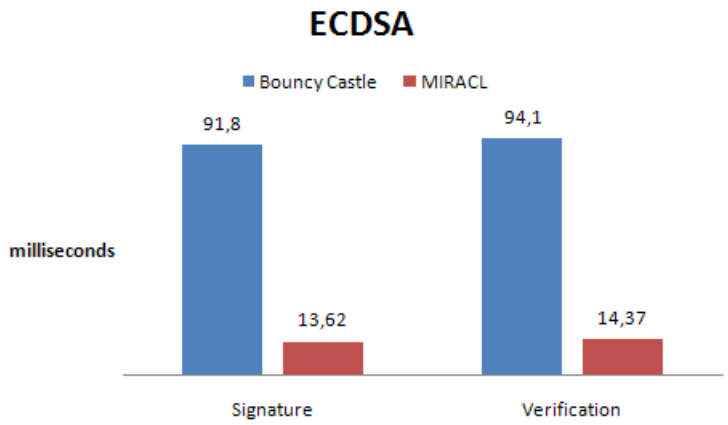

Figure 5.1: ECDSA comparison

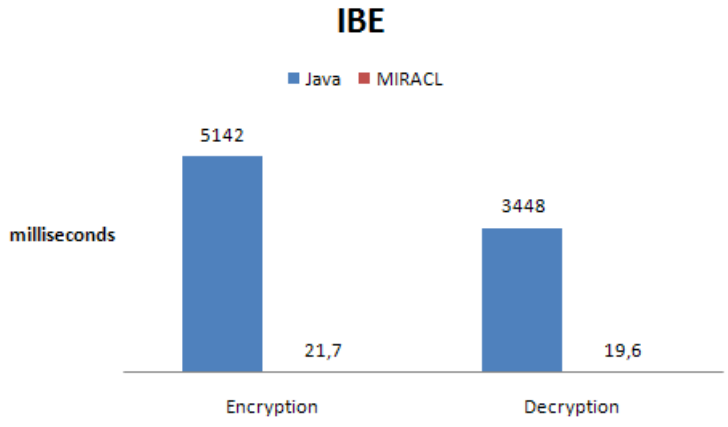

Figure 5.2: IBE comparison

5.2 Embedding C++ into Java

After conducting the tests presented in the previous sections and analysing the results, it was decided to embed some of the C++ MIRACL functions into the Java programs: the IBE scheme and Bouncy Castle's ECDSA. The objective was to achieve greater performance in Java by using the power of MIRACL to compute the operations that impose greater workload on the computer:

- Bilinear pairings calculations
- Elliptic curves arithmetics

While making the preparations to edit the source code of the IBE scheme on Java, a setback emerged. JCA requires that providers that implement Cipher objects to be digitally signed by Sun Microsystems. So, it was impossible to develop and test a new provider from the source code because a new Cipher instance was to be created and therefore a signature for the provider was needed. This requirement and the process to obtain the signed provider are described in [48]. Instead of using the JCA provider architecture, it is possible to manually instantiate the classes and call all the necessary methods, something that JCA mechanism does automatically, but this solution is more complicated from a programmer's point of view and was initially intended to use the program as a JCA provider. Due to time constraints, an alternative solution was sought. It was found that there are *clean-room implementations* of javax.crypto packages (BC distributes one). By using this packages, one is able to replace or supercede the jce.jar file that comes with the JVM, thus being able to use non-signed providers. The packages made available by the GNU Crypto project [49] were used for this purpose. Unfortunately, this alternative also raised some compilation problems. And again due to time constraints, embedding C++ into the Java IBE scheme had to be put postponed.

The problems raised by the IBE scheme don't apply to the Bouncy Castle ECDSA. Since it is a signature algorithm, the JVM doesn't need to authenticate the provider, unlike what happens with ciphers. So the tests that were made consist on embedding C++ into the Bouncy Castle ECDSA. This was achieved by using the Java Native Interface (JNI).

Note that, even if IBE testing was postponed, it is expectable that the performance gains obtained for IBE to be significantly more substantial since, as it was shown above, the gap between the performance of the C++ and Java implementations is much bigger.

5.2.1 Java Native Interface

Java Native Interface is a framework that allows Java code to call and be called by native applications or libraries written in other languages. The most complicated part of using JNI is mapping the types between the different languages. In this particular case it is needed to call a C++ procedure in a Java class. The JNI engine will automatically convert the Java native types to its own internal types, but on the C++ function being called it is necessary to convert from JNI types to

C++ native types and return the result on a JNI type. Again, JNI will automatically convert its types to the Java native ones and so Java and C++ are able to pass data among them. For more information about JNI, please refer to [50].

5.2.2 Calling the C++ procedures

The first thing to do is to identify which operation of the ECDSA algorithm will be computed by the MIRACL. By analysing the ECDSA specification, as defined in section 2.4.4 it was decided to pass to C++ the burden of computing elliptic curve arithmetics.

In the signature part:

- Compute kG where k is a random integer $1 \leq k \leq n-1$ and G is the generator point of the elliptic curve. This operation corresponds to Step 1 of ECSP-DSA, as seen in section 2.4.4.

In the verification part:

- Compute $X = u_1G + u_2Q_A$ where $u_1 = ew \mod n$, $u_2 = rw \mod n$, $w = s^{-1}$, $e = hash(m)$. s is part of the signature (r, s), m is the message being signed and Q_A is the signer's public key. This operation corresponds to Step 3 of ECVP-DSA, as seen in section 2.4.4.

The steps needed to be taken to call the C++ functions are presented next.

JNI is used to generate a header file. This file will be used by the C++ file that implements the code. The file is generated once the methods are declared on the Java class, by calling the method:

- *javah -jni <java class name>*

The most complex part is done in the C++ side.

1. Include the libraries *jni.h* and the header file generated by JNI.
2. Codifying the body of the functions declared on the Java side.
3. Make the necessary type conversion between JNI types and C++ native types.
4. Once the code is complete, create a library so that the Java class can access the methods. In Windows, this library will be a .DLL (Dynamic-Link Library) file. On Unix based systems, a .SO file (shared object). Note that by creating libraries according to the operating system, the Java program that uses them will loose its mobility. For example. if it uses a .SO library, it will not run on a Windows platform.

```
JNIEXPORT jobjectArray JNICALL
    Java_org_bouncycastle_crypto_signers_ECDSASigner_kG
  (JNIEnv *env, jobject obj, jstring a, jstring b, jstring q, jstring gx,
     jstring gy, jstring k)
```

```
{
miracl *mip = &precision;
jobjectArray ret;

char str_res_x[100000];
char str_res_y[100000];

Big big_a,big_b,big_q,big_gx,big_gy,big_k,res_x,res_y;
ECn G,W;
const char *str_a,*str_b,*str_q,*str_gx,*str_gy,*str_k;

str_a = (env)->GetStringUTFChars (a, NULL);
str_b = (env)->GetStringUTFChars (b, NULL);
str_q = (env)->GetStringUTFChars (q, NULL);
str_gx = (env)->GetStringUTFChars (gx, NULL);
str_gy = (env)->GetStringUTFChars (gy, NULL);
str_k = (env)->GetStringUTFChars (k, NULL);

ret= (jobjectArray)env->NewObjectArray(2,
         env->FindClass("java/lang/String"),
         env->NewStringUTF(""));


 mip->IOBASE=16;
  // x=str;
  big_a=(char*)str_a;
  big_b=(char*)str_b;
  big_q=(char*)str_q;
  big_gx=(char*)str_gx;
  big_gy=(char*)str_gy;
  big_k=(char*)str_k;

 mip->IOBASE = 10;
 ecurve (big_a, big_b, big_q, MR_PROJECTIVE);
 // G=ECn(big_gx,big_gy);

 if (!G.set(big_gx,big_gy))
    {
        cout << "Problem - point (x,y) is not on the curve" << endl;
       // return 0;
    }

 W=G;
 W *= big_k;

 mip->IOBASE=16;
 W.get(res_x,res_y);
```

```
 str_res_x << res_x;
 str_res_y << res_y;


 env->SetObjectArrayElement(ret,0,env->NewStringUTF(str_res_x));
 env->SetObjectArrayElement(ret,1,env->NewStringUTF(str_res_y));

(env)->ReleaseStringUTFChars (a, str_a);
(env)->ReleaseStringUTFChars (b, str_b);
(env)->ReleaseStringUTFChars (q, str_q);
(env)->ReleaseStringUTFChars (gx, str_gx);
(env)->ReleaseStringUTFChars (gy, str_gy);
(env)->ReleaseStringUTFChars (k, str_k);

return ret;
```

Code excerpt 5.1: Implementing the method that computes kG

Code excerpt 5.1 illustrates the mapping that is made from JNI types to C++ native types and *vice-versa*. Once in possession of the parameters in the C++ types, the computations are done in a normal fashion. Then the result is passed again to a JNI type, to be returned to the calling Java class.

On the Java side we need to modify the class ECDSASigner, particularly the methods *generateSignature* and *verifySignature*, because as seen in section 3.4, it implements the ECDSA signature and verification primitives:

1. Call the external library on the Java class. This the library that contains the MIRACL functions.

2. Declare the new methods that will be called. These methods will be implemented on the C++ side.

3. If necessary, make type conversions to fit the methods being called.

```
static {
        System.loadLibrary("miracl");
        }

    public native String[] kG(String a,String b,String p,String Gx,String Gy,
        String k);

    public native String[] U2PubU1G(String a,String b,String p,String Gx,String
         Gy,String u2,String pubX,String pubY, String u1);
```

Code excerpt 5.2: Declarations on the Java class

In the code excerpt 5.2 it is possible to see the method that calls the external library and also the declarations of the methods that will be used. The body of this method will be on the C++ side, as implemented in code excerpt 5.1. The methods pass to the MIRACL side the curve parameters, public key parameters, as well the elements of the computations that will be done. Method *kG* will be used on the signature part; *U2PubU1G* will be used on the verification part.

```
BigInteger bi_q= ((ECCurve.Fp)key.getParameters().getCurve()).getQ();
String q=bi_q.toString(16);
String Gx = key.getParameters().getG().getX().toBigInteger().toString(16);
String Gy = key.getParameters().getG().getY().toBigInteger().toString(16);
String a = key.getParameters().getCurve().getA().toBigInteger().toString(16);
String b = key.getParameters().getCurve().getB().toBigInteger().toString(16);
String kapa = k.toString(16);

String obj[]=new String[2];
obj=kG(a,b,q,Gx,Gy,kapa);

BigInteger fast_gx = new BigInteger(obj[0],16);
BigInteger fast_gy = new BigInteger(obj[1],16);

ECFieldElement.Fp fex = new ECFieldElement.Fp (bi_q,fast_gx);
ECFieldElement.Fp fey = new ECFieldElement.Fp (bi_q,fast_gy);

ECPoint.Fp p = new ECPoint.Fp (key.getParameters().getCurve(),fex,fey);
```

Code excerpt 5.3: Conversions in the Java class

In code excerpt 5.3 it is possible to see some type conversions needed inside the Java class (in the method *generateSignature*, in this case). All the parameters are represented as BigInteger. So, it was decided to convert them to hexadecimal strings to be passed to the C++. It can be seen the method *kG* being called and the result being obtained. Each invocation of the method *kG* is stateless. The parameters are always passed to the C++ side which constructs its own representation of the elliptic curve. There is no state that is saved in the C++ side from one invocation to another. A few more conversions follow and then finally the point P is obtained as the result of computing *kG* on the C++ side. After this point, the Java method resumes execution and so the remaining steps of the algorithm will be executed.

```
public BigInteger[] generateSignature(
        byte[] message)
    {

        BigInteger n = key.getParameters().getN();
        BigInteger e = calculateE(n, message);
        BigInteger r = null;
        BigInteger s = null;
```

```
// 5.3.2
do // generate s
{
    BigInteger k = null;
    int        nBitLength = n.bitLength();

    do // generate r
    {
        do
        {
            k = new BigInteger(nBitLength, random);
        }
        while (k.equals(ZERO));

     BigInteger bi_q= ((ECCurve.Fp)key.getParameters().getCurve()).getQ
         ();

     String q=bi_q.toString(16);
     String Gx = key.getParameters().getG().getX().toBigInteger().
         toString(16);
     String Gy = key.getParameters().getG().getY().toBigInteger().
         toString(16);
     String a = key.getParameters().getCurve().getA().toBigInteger().
         toString(16);
     String b = key.getParameters().getCurve().getB().toBigInteger().
         toString(16);
     String kapa = k.toString(16);

     String obj[]=new String[2];
     obj=kG(a,b,q,Gx,Gy,kapa);


     BigInteger fast_gx = new BigInteger(obj[0],16);
     BigInteger fast_gy = new BigInteger(obj[0],16);

    ECFieldElement.Fp fex = new ECFieldElement.Fp (bi_q,fast_gx);
    ECFieldElement.Fp fey = new ECFieldElement.Fp (bi_q,fast_gy);

    ECPoint.Fp p = new ECPoint.Fp (key.getParameters().getCurve(),fex,
        fey);


        // 5.3.3
        BigInteger x = p.getX().toBigInteger();

        r = x.mod(n);
    }
    while (r.equals(ZERO));
```

```
            BigInteger d = ((ECPrivateKeyParameters)key).getD();

            s = k.modInverse(n).multiply(e.add(d.multiply(r))).mod(n);
        }
        while (s.equals(ZERO));

        BigInteger[]  res = new BigInteger[2];

        res[0] = r;
        res[1] = s;

        return res;
    }
```

Code excerpt 5.4: New version of method generateSignature

The full code of the modified method *generateSignature* is in code excerpt 5.4. This can be compared to the original code showed in code excerpt 3.7. Apart from the type conversions, the code remains the same.

The code excerpts only illustrate the function *kG* and the modified method *generateSignature* of class ECDSASigner, but the same method and principles were applied to the function *U2PubU1G* and to the method *verifySignature* of the same class.

5.3 Results

In this section are displayed the results of the tests with Bouncy Castle with C++ embedded.

	Signature	Verification
1	7	21
2	7	21
3	9	21
4	28	55
5	32	40
6	8	45
7	7	42
8	16	20
9	8	22
10	8	20
Average	13	30,7

Table 5.5: BC with MIRACL single execution times in milliseconds

	Signature	Verification
1	23	19
2	5	4
3	3	4
4	3	10
5	4	4
6	4	4
7	3	4
8	20	10
9	5	14
10	9	4
Average	7,9	7,7

Table 5.6: BC with MIRACL multi-execution times in milliseconds

Tables 5.5 and 5.6 represent the tests made with single and multi-execution scenarios.

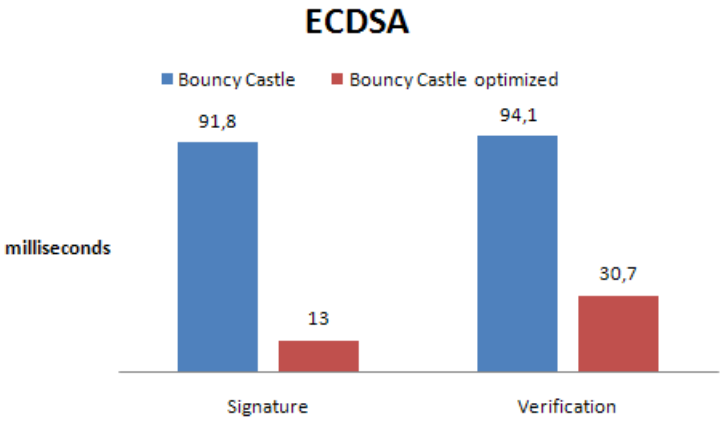

Figure 5.3: Bouncy Castle versus Bouncy Castle optimised

Figure 5.3 illustrates the performance increase that took place by using C++ in the Bouncy Castle ECDSA. The data comes from tables 3.2 and 5.5. There was a decrease of 85% in the signature time and a decrease of 67,3% on the verification time. These values leave no margin for error: despite the necessary type conversions, it pays off greatly to let MIRACL compute the heavier calculations. As stated before, the same principle can theorically be applied to the IBE implementation and bigger performance gains can be expected.

6

Conclusions and further work

In this chapter, the conclusions of this work are presented, as well the suggestions for further work.

6.1 Conclusions

As closing remarks of this work, some solid conclusions may be drawn from the results presented on the previous chapters. Bouncy Castle is an example of how cryptographic API's written in Java can be well structured and organised, while offering not only the most common and used cryptographic algorithms, but emerging techniques such as those based on elliptic curves. The design of these API's as a Java Cryptography Architecture Provider broadens their availability to Java programmers.

Bouncy Castle is also an example of how the *low level* cryptographic standards such as PCKS, SEC and P1363 can guide the implementation. And by following the standards as technical guides, the conformance to those standards will be a consequence. In particular and as an example, we showed how Bouncy Castle follows the standards guidelines on the implementation of RSA encryption and ECDSA signature schemes.

Techniques based on bilinear-pairings are something relatively new on the Cryptography world. One of the most interesting concepts proposed on this area is the Identity Based Encryption (IBE), which can be faced as a future alternative to the current Public Key Infrastructure. We presented a Java implementation of an Identity Based Encryption System. While its a project on its early stages, it shows a path to the use of bilinear pairing on Java that others can follow.

Throughout this document, performance tests were presented. Two scenarios were showed:

single and multi-executions. The single execution times correspond to the normal use of a cryptographic algorithm, which is precisely a single execution each time. The multi-execution scenario serves the purpose of showing how the compilers plus the computer architecture can optimise the performance of some cryptographic algorithms, when the algorithm is repeatedly executed. In particular, the Elliptic Curve Digital Signature Algorithm (ECDSA) shows significant performance increase, while the Identity Based Encryption gains are much smaller. This should be a consequence of the heavier computations that bilinear pairings require.

We also showed how the use of C++ embedded into Java implementations of cryptographic algorithms, through the use of the Java Native Interface (JNI), can decrease the execution times of the Java implementations. We made tests by using C/C++ library MIRACL and the Bouncy Castle's ECDSA. The results are illustrative of the performance gains, and theorically the same principle can be extended to the IBE implementation. It is also interesting to verify that even with time lost during the type mapping between Java / JNI / C++, the end result is positive in terms of performance gains.

The quantitative results should not be regarded as something definitive. There's a margin of error inherent to measuring execution times and also ECDSA uses random numbers, which can make times differ from one execution to another. Regarding this subject, what can be seen as the major conclusion is the fact that the use of *low-level* languages like C++ embedded into Java, can be regarded as a way to increase performance of Java based cryptographic algorithms.

6.2 Further work

As future extensions to this work, it will be very interesting to embed C/C++ into the Java IBE implementation presented in this document, like was done for Bouncy's Castle ECDSA. The objective is to investigate if it is possible to have some performance gains by letting some of the mathematical operations involved in bilinear pairings be done by a *low-level* language.

Bibliography

[1] Common Criteria Maintenance Board. *Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and general model*, version 3.1 revision 1 edition, 2006.

[2] Common Criteria Maintenance Board. *Common Criteria for Information Technology Security Evaluation, Part 2: Security functional components*, version 3.1 revision 2 edition, 2007.

[3] Common Criteria Maintenance Board. *Common Criteria for Information Technology Security Evaluation, Part 3: Security assurance components*, version 3.1 revision 2 edition, 2007.

[4] Common Criteria Maintenance Board. *Common Criteria for Information Technology Security Evaluation, Evaluation methodology*, version 3.1 revision 2 edition, 2007.

[5] Apple Inc. *Common Criteria Certification: Apple's Ongoing Commitment to Security*.

[6] Stuart Katzke. The common criteria (cc) paradigm. `www.ieeeia.org/iasw/Katzke-CC.ppt`.

[7] National Institute of Standards Information Technology Laboratory and Technology. *FIPS PUB 140-2, Federal Information Processing Standards Publication, Security Requirements for Cryptograpic Modules*, May 2001.

[8] National Institute of Standards Information Technology Laboratory and Technology. *FIPS PUB 198-1, Federal Information Processing Standards Publication, The Keyed-Hash Message Authentication Code (HMAC)*, June 2007.

[9] Karl Scheibelhofer. Security evaluations.

[10] RSA Laboratories. *PKCS #1 v2.1: RSA Cryptography Standard*, June 2002.

[11] RSA Laboratories. *PKCS #3 v1.4: Diffie-Hellman Key-Agreement Standard*, November 1993.

[12] RSA Laboratories. *PKCS #5 v2.1: Password-Based Cryptography Standard*, October 2006.

[13] RSA Laboratories. *PKCS #8 v1.2: Private-Key Information Syntax Standard*, November 1993.

[14] RSA Laboratories. *PKCS #12 v1.0: Personal Information Exchange Syntax*, June 1999.

[15] RSA Laboratories. *PKCS #10 v1.7: Certification Request Syntax Standard*, May 2000.

[16] RSA Laboratories. *PKCS #9: Selected Object Classes and Attribute Types*, February 2000.

[17] RSA Laboratories. *PKCS #11 v2.20: Cryptographic Token Interface Standard*, June 2004.

[18] RSA Laboratories. *PKCS #15 v1.1: Cryptographic Token Information Syntax Standard*, June 2000.

[19] J-J. Quisquater and C. Couvreur. Fast decipherment algorithm for rsa public-key cryptosystem, 1982.

[20] Institute of Electrical and Electronics Engineers. *IEEE Std 1363-2000: IEEE Standard Specifications for Public-Key Cryptography*, 2004.

[21] Institute of Electrical and Electronics Engineers. *IEEE Std 1363a-2004: IEEE Standard Specifications for Public-Key Cryptography - Amendment 1: Additional Techniques*, June 2000.

[22] Institute of Electrical and Electronics Engineers. *IEEE P1363.1/D9: Draft Standard for Public-Key Cryptographic Techniques Based on Hard Problems over Lattices (Draft D9)*, January 2007.

[23] Institute of Electrical and Electronics Engineers. *IEEE P1363.2/D26: Draft Standard for Specifications for Password-based Public Key Cryptographic Techniques (Draft D26)*, September 2006.

[24] Institute of Electrical and Electronics Engineers. *IEEE P1636.3/D1 Draft Standard for Identity-based Public-key Cryptography Using Pairings*, April 2008.

[25] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[26] Institute of Electrical and Electronics Engineers. *IEEE P1363 / D13 Standard Specifications for Public-Key Cryptography: Annex A, Number-Theoretic Background.*, 1999.

[27] Scott Vanstone. Responses to nist's proposal, 1992.

[28] Certicom Research. *SEC 1: Elliptic Curve Cryptography*, September 2000.

[29] Java cryptographic architecture reference guide. `http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html`.

[30] David Brumley and Dan Boneh. Remote timing attacks are practical. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2003. USENIX Association.

[31] National Institute of Standards Information Technology Laboratory and Technology. *FIPS PUB 140-2, Federal Information Processing Standards Publication, Digital Signature Standards (DSS)*, January 2000.

[32] Adi Shamir. Identity-based cryptosystems and signature schemes. In *Proceedings of CRYPTO 84 on Advances in cryptology*, pages 47–53, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[33] Dan Boneh and Matthew Franklin. Identity-based encryption from the weil pairing. *SIAM J. Comput.*, 32(3):586–615, 2003.

[34] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[35] CJoosang Baek;Jan Newmarch;Reihaneh Safavi-Naini;Willy Susilo. A survey of identity-based cryptography. In *Proceedings of Australian Unix Users Group Annual Conference*, 2004.

[36] Craig Gentry and Alice Silverberg. Hierarchical id-based cryptography. In *ASIACRYPT '02: Proceedings of the 8th International Conference on the Theory and Application of Cryptology and Information Security*, pages 548–566, London, UK, 2002. Springer-Verlag.

[37] Manuel Bernardo Barbosa. Identity based cryptography from bilinear pairings. 2005.

[38] Antoine Joux. A one round protocol for tripartite diffie-hellman. In *ANTS-IV: Proceedings of the 4th International Symposium on Algorithmic Number Theory*, pages 385–394, London, UK, 2000. Springer-Verlag.

[39] Alfred Menezes. An introduction to pairing-based cryptography, 2005.

[40] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *ASIACRYPT '01: Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security*, pages 514–532, London, UK, 2001. Springer-Verlag.

[41] Clifford Cocks. An identity based encryption scheme based on quadratic residues. In *Proceedings of the 8th IMA International Conference on Cryptography and Coding*, pages 360–363, London, UK, 2001. Springer-Verlag.

[42] Louise Owens, Adam Duffy, and Tom Dowling. An identity based encryption system. In *PPPJ '04: Proceedings of the 3rd international symposium on Principles and practice of programming in Java*, pages 154–159. Trinity College Dublin, 2004.

[43] Shamus Software. Miracle. http://www.shamus.ie/.

[44] NUI Maynooth Crypto Group. Nui maynooth crypto group website. http://www.crypto.cs.nuim.ie/software/eyebee.

[45] Andrew Burnett, Keith Winters, and Tom Dowling. A java implementation of an elliptic curve cryptosystem. In *PPPJ '02/IRE '02: Proceedings of the inaugural conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate representation engineering for virtual machines, 2002*, pages 83–88, Maynooth, County Kildare, Ireland, Ireland, 2002. National University of Ireland.

[46] James Parnis and Gareth Lee. Exploiting fpga concurrency to enhance jvm performance. In *ACSC '04: Proceedings of the 27th Australasian conference on Computer science*, pages 223–232, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[47] Iffat H. Kazi, Howard H. Chen, Berdenia Stanley, and David J. Lilja. Techniques for obtaining high performance in java programs. *ACM Comput. Surv.*, 32(3):213–240, 2000.

[48] How to implement a provider in the java cryptogaphic architecture. http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/HowToImplAProvider.html.

[49] Gnu crypto. http://www.gnu.org/software/gnu-crypto.

[50] Sun. Java native interface. http://java.sun.com/javase/6/docs/technotes/guides/jni/index.html.