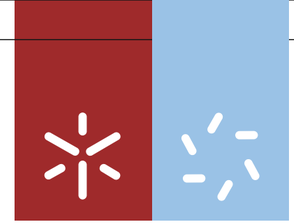


Universidade do Minho
Escola de Ciências

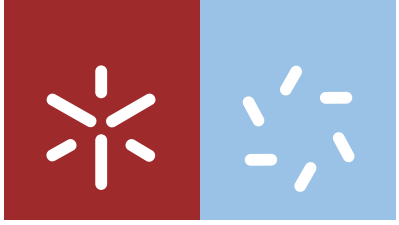
Ana Catarina Pereira Correia

Public Keys Quality

outubro de 2015

Public Keys Quality
Ana Catarina Pereira Correia

UMinho | 2015



Universidade do Minho
Escola de Ciências

Ana Catarina Pereira Correia

Public Keys Quality

Dissertação de Mestrado
Mestrado em Matemática e Computação

Trabalho realizado sob orientação de:
Orientador: **Professor Doutor José Pedro Miranda Mourão Patrício**
Co-orientador: **Professor Doutor José Carlos Bacelar Almeida**
Tutor: **Engenheiro José Martins**

outubro de 2015

Declaração

Nome: Ana Catarina Pereira Correia
Endereço electrónico: catarina.correia.m@gmail.com
Número do Cartão de Cidadão: 13781321

Título dissertação:

Public Keys Quality

Orientador: Professor Doutor José Pedro Miranda Mourão Patrício,
Co-orientador: Professor Doutor José Carlos Bacelar Almeida
Tutor: Engenheiro José Martins

Ano de conclusão: 2015

Designação do Mestrado: Mestrado em Matemática e Computação

Universidade do Minho, _____ / _____ / _____

Assinatura: _____

Acknowledgements

Firstly I would like to thank my course directors and MULTICERT for giving me this amazing opportunity.

I would like to thank professor Pedro Patrício and Professor José Carlos Bacelar for the good theoretical orientations and discussions about the "to dos" in this project. It was really important to have their background support during my internship in MULTICERT.

I definitely own a huge thanks to José Martins and Nuno Martins for accepting the challenge of tutoring me at MULTICERT. José Martins, although not professionally, you are a great teacher. You have a special ability to anticipate the difficulties and the following steps during a project. You not only orientated me into my thesis but you also guided me into my growth as a mathematician developer. I lived in Spring all seasons of this year! Nuno Martins, thank you for your inspirations, for your big dreams and mostly for believing in my project. It was really important having you as my non official tutor. I also would like to thank Pedro Cunha for the beautiful discussions we had about Java and programming . I would like to thank all of the people working in MULTICERT. Thanks for the laughs, for the team work, for the happy lunches, for the help and for making me feel like I was part of your work family. I had the opportunity to meet amazing persons.

There would be no chance to forget to acknowledge and thank Professor Eduardo Nunes-Pereira for his sincere advises, for showing me how sometimes we need to look at the "big picture". It was a great experience be in an amazing world of light and flights. I still carry with me all of the knowledge. Thank you for the amazing tutor and human person you are.

I would like to thank Ana Coroas and Nuno Viana for the comments about my work. Sorry for being a terrible friend. A special thank to my family friends. I know you love to listen me talking for hours about things you don't care about.

A lovely thanks to Carlos Sá. Our paths converged more to the same path than what we expected to and this makes me love us even more. Thanks for taking this journey with me and for being by my side.

I would like to thank my family for the great support. A special thanks to my mother, my father, my sister and my grandmother. Not only for the support you gave me this year but for all the support you gave me my entire life, encouraging me to conquer my dreams and to go after to what I love in in this life. This work is also yours.

Abstract

The RSA cryptosystem, invented by Ron Rivest, Adi Shamir and Len Adleman ([Rivest et al., 1978]) is the most commonly used cryptosystem for providing privacy and ensuring authenticity of digital data. RSA is usually used in contexts where security of digital data is priority. RSA is used worldwide by web servers and browsers to secure web traffic, to ensure privacy and authenticity of e-mail, to secure remote login sessions and to provide secure electronic credit-card payment systems.

Given its importance in the protection of digital data, vulnerabilities of RSA have been analysed by many researchers. The researches made so far led to a number of fascinating attacks. Although the attacks helped to improve the security of this cryptosystem, showing that securely implementing RSA is a nontrivial task, none of them was devastating.

This master thesis discusses the RSA cryptosystem and some of its vulnerabilities as well as the description of some attacks, both recent and old, together with the description of the underlying mathematical tools they use. Although many types of attacks exist, in this master thesis only a few examples were analysed. The ultimate attack, based in the batch-GCD algorithm, was implemented and tested in the RSA keys produced by a certificated Hardware Security Modules Luna SA and the results were commented.

The random and pseudorandom numbers are fundamental to many cryptographic applications, including the RSA cryptosystems. In fact, the produced keys must be generated in a specific random way. The National Institute of Standards and Technology, responsible entity for specifying safety standards, provides a package named "A Statistical Test Suit for Random and Pseudorandom Number Generators for Cryptography Applications" which was used in this work to test the randomness of the Luna SA generated numbers. All the statistical tests were tested in different bit sizes number and the results commented.

The main purpose of this thesis is to study the previous subjects and create an applications capable to test the Luna SA generated numbers randomness, a well as evaluate the security of the RSA.

This work was developed in partnership with University of Minho and Multicert.

Resumo

O RSA, criado por Ron Rivest, Adi Shamir e Len Adleman ([Rivest et al., 1978]) é o sistema criptográfico mais utilizado para providenciar segurança e assegurar a autenticação de dados utilizados no mundo digital. O RSA é usualmente usado em contextos onde a segurança é a grande prioridade. Hoje em dia, este sistema criptográfico é utilizado mundialmente por servidores web e por *browsers*, por forma a assegurar um tráfego seguro através da Internet. É o sistema criptográfico mais utilizado na autenticação de *e-mails*, nos inícios de sessões remotos, na utilização de pagamentos através de cartões multibanco, garantindo segurança na utilização destes serviços.

Dada a importância que este sistema assume na proteção da informação digital, as suas vulnerabilidades têm sido alvo de várias investigações. Estas investigações resultaram em vários ataques ao RSA. Embora nenhum destes ataques seja efetivamente eficaz, todos contribuíram para um aumento da segurança do RSA, uma vez que as implementações de referência deste algoritmo passaram a precaver-se contra os ataques descobertos.

Esta tese de mestrado aborda o sistema criptográfico RSA, discutindo algumas das suas vulnerabilidades, assim como alguns ataques efetuados a este sistema, estudando todos os métodos matemáticos por estes usados. Embora existam diversos ataques, apenas alguns serão abordados nesta tese de mestrado. O último ataque, baseado no algoritmo *batch-GCD* foi implementado e foram feitos testes em chaves RSA produzidas por um *Hardware Security Module Luna SA* certificado e os resultados obtidos foram discutidos.

Os números aleatórios e pseudoaleatórios são fundamentais a todas as aplicações criptográficas, incluindo, portanto, o sistema criptográfico RSA. De facto, as chaves produzidas deverão ser geradas com alguma aleatoriedade intrínseca ao sistema. O Instituto Nacional de Standards e Tecnologia, entidade responsável pela especificação dos *standards* de segurança, disponibiliza um pacote de testes estatísticos, denominado por "A Statistical Test Suit for Random and Pseudorandom Number Generators for Cryptography Applications". Estes testes estatísticos foram aplicados a números gerados pelo Luna SA e os resultados foram, também, comentados.

O objetivo desta tese de mestrado é desenvolver capacidade de compreensão sobre os assuntos descritos anteriormente e criar uma aplicação capaz de testar a aleatoriedade dos números gerados pelo Luna SA, assim como avaliar a segurança do sistema criptográfico RSA.

Este foi um trabalho desenvolvido em parceria com a Universidade do Minho e com a Multicert.

Contents

Acknowledgements iii

Abstract v

Resumo vii

1 Introduction 1
- 1.1 Cryptography . . . 1
- 1.2 Public Key Cryptography . . . 3
- 1.3 Diffie-Hellman Algorithm . . . 4

2 The RSA Public-Key Cryptosystem 7
- 2.1 RSA Key Generation, Message Encryption and Decryption . . . 7
- 2.2 RSA vulnerabilities . . . 8
  - 2.2.1 The RSA problem . . . 8
  - 2.2.2 Large integer numbers factorization . . . 9
  - 2.2.3 Common modulus . . . 10
  - 2.2.4 Blinding . . . 11

3 The RSA security and cryptanalysis 13
- 3.1 Small public exponent . . . 13
  - 3.1.1 Hastad's broadcast attack . . . 14
  - 3.1.2 Franklin-Reiter Related Message Attack . . . 17
  - 3.1.3 Coppersmith's Short Pad Attack . . . 17
  - 3.1.4 Partial Key Exposure Attack . . . 18
- 3.2 Low private exponent . . . 19
  - 3.2.1 Wiener's Attack . . . 19
- 3.3 Bleichenbacher's Attack on PKCS#1 . . . 21

4 Factorization 23
- 4.1 Trial Factorization . . . 23
- 4.2 Fermat Factorization . . . 24
- 4.3 Continued Fraction Method . . . 24
- 4.4 Pollard's $p-1$ Factorization Algorithm . . . 25

4.5 Elliptic Curve Method . . . 27
4.6 Quadratic Sieve . . . 28
4.7 General Number Field Sieve . . . 28
4.8 RSA Modulus Factorization . . . 29

5 Recent Attacks on RSA Keys 31
5.1 How the attack works . . . 32
5.1.1 Batch-GCD . . . 33
5.1.2 Coppersmith-style attacks . . . 34

6 Randomness 37
6.1 Random and Pseudorandom Numbers . . . 38
6.2 Random Number Generators (RNGs) . . . 39
6.3 Pseudorandom Number Generators (PRNGs) . . . 39
6.3.1 Failed Algorithms . . . 41
6.3.2 Cryptographically Strong Sequences . . . 42
6.4 Theoretical Constructions of Pseudorandom Objects . . . 43
6.5 Testing Randomness . . . 51
6.5.1 How a Statistical Test Works . . . 52
6.5.2 Frequency Test . . . 54
6.5.3 Frequency Test Within a Block . . . 55
6.5.4 Runs Test . . . 55
6.5.5 Test for the Longest Run of Ones in a Block . . . 56
6.5.6 Binary Matrix Rank Test . . . 57
6.5.7 Discrete Fourier Transform (Spectral) Test . . . 58
6.5.8 Non-overlapping Template Matching Test . . . 58
6.5.9 Overlapping Template Matching Test . . . 59
6.5.10 Maurer's "Universal Statistical" Test . . . 60
6.5.11 Linear Complexity Test . . . 61
6.5.12 Serial Test . . . 63
6.5.13 Approximate Entropy Test . . . 63
6.5.14 Cumulative Sums Test . . . 64
6.5.15 Random Excursions Test . . . 65
6.5.16 Random Excursions Variant Test . . . 66
6.6 Recommendations for Random Numbers Generation. Randomness Requirements for Security. . . . 67
6.6.1 Entropy Sources . . . 68
6.6.2 De-skewing . . . 71
6.6.3 Mixing . . . 73
6.7 The Blum Blum Shub Generator Example . . . 75

7 Quality analysis of generated numbers 77
7.1 Quality analysis of a Hardware Security Module (HSM) generated number of 10^6 bits . . . 77
7.1.1 Frequency Test . . . 78
7.1.2 Frequency Test Within a Block . . . 78
7.1.3 Runs Test . . . 79
7.1.4 Test for the Longest Run of Ones in a Block . . . 79
7.1.5 Binary Matrix Rank Test . . . 79
7.1.6 Discrete Fourier Transform (Spectral) Test . . . 80
7.1.7 Non-overlapping Template Matching Test . . . 81
7.1.8 Overlapping Template Matching Test . . . 82
7.1.9 Maurer's "Universal Statistical" Test . . . 82
7.1.10 Linear Complexity Test . . . 83
7.1.11 Serial Test . . . 84
7.1.12 Approximate Entropy Test . . . 84
7.1.13 Cumulative Sums Test . . . 85
7.1.14 Random Excursions Test . . . 85
7.1.15 Random Excursions Variant Test . . . 85
7.2 Quality analysis of a Hardware Security Module (HSM) generated number of 10^9 bits . . . 86
7.2.1 Frequency Test . . . 86
7.2.2 Frequency Test Within a Block . . . 87
7.2.3 Runs Test . . . 88
7.2.4 Test for the Longest Run of Ones in a Block . . . 88
7.2.5 Binary Matrix Rank Test . . . 88
7.2.6 Non-overlapping Template Matching Test . . . 89
7.2.7 Overlapping Template Matching Test . . . 90
7.2.8 Maurer's "Universal Statistical" Test . . . 90
7.2.9 Linear Complexity Test . . . 91
7.2.10 Serial Test . . . 91
7.2.11 Approximate Entropy Test . . . 93
7.2.12 Cumulative Sums Test . . . 93
7.3 Quality analysis of a Hardware Security Module (HSM) set of generated numbers 94
7.4 Conclusions . . . 94

8 MQualityTester Application 97

9 Conclusions and Future Work 103
9.1 Conclusions . . . 103
9.2 Future Work . . . 104

A Coppersmith's Theorem 105

B Continued Fraction Method Fundamental Concepts **107**
B.1 Continued Fractions 107
B.2 Factor Basis and Smooth Numbers 109

C Tables **111**

List of Tables

6.1 True status of the data available for analysis and the conclusion arrived by the usage of the testing procedure. An important observation is that the status of the data available for analysis is unknown in almost all the cases, in fact, that is why the statistical tests are made. 53
6.2 Values of N and K according to the values of M. 56
6.3 Acceptable values for M according to the sequence to be tested minimun length, n. 57
6.4 Representative table of how the values of L, Q and n should be chosen. 62
6.5 Method of recording the values of T_i in $v_0, v_1, \ldots, v_6$. 62
6.6 Bit pairs probability. 72

7.1 Statistics table of the Frequency Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 78
7.2 Statistics table of the Frequency Test Within a Block for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 78
7.3 Statistics table of the Runs Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 79
7.4 Statistics table of the Test for the Longest Run of Ones in a Block for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 79
7.5 Statistics table of the Binary Matrix Rank Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 80
7.6 Statistics table of the Discrete Fourier Transform (Spectral) Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 80
7.7 Statistics table of the Non-Overlapping Template Matching Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). The complete table is in C, table C.1 81

7.8 Statistics table of the Overlapping Template Matching Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 82
7.9 Statistics table of the Maurer's "Universal Statistical" Test for a 1000000 bits number. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 83
7.10 Statistics table of the Linear Complexity Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 83
7.11 Statistics table of the Serial Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 84
7.12 Statistics table of the Approximate Entropy Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 84
7.13 Statistics table of the Cumulative Sums (forward) Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 85
7.14 Statistics table of the Cumulative Sums (reverse) Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 85
7.15 Statistics table of the Random Excursions Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 86
7.16 Statistics table of the Random Excursions Variant Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 87
7.17 Statistics table of the Frequency Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 87
7.18 Statistics table of the Frequency Test Within a Block for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 88
7.19 Statistics table of the Runs Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 88
7.20 Statistics table of the Test for the Longest Run of Ones in a Block for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 89
7.21 Statistics table of the Binary Matrix Rank Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 89

7.22 Statistics table of the Non-Overlapping Template Matching Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). The complete table is in C, table C.2 90
7.23 Statistics table of the Overlapping Template Matching Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 91
7.24 Statistics table of the Maurer's "Universal Statistical" Test for a 10^9 bits number. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 91
7.25 Statistics table of the Linear Complexity Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 92
7.26 Statistics table of the Serial Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 93
7.27 Statistics table of the Approximate Entropy Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 93
7.28 Statistics table of the Cumulative Sums (forward) Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 94
7.29 Statistics table of the Cumulative Sums (reverse) Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). 94
7.30 Pass rates test for each one of the NIST Statistical Test Suit [Rukhin et al., 2010] based on 100 tests for 100 10^6 bits number generated by a Luna SA HSM. 96

C.1 Statistics table of the Non-Overlapping Template Matching Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hipothesis (the hipothesis that states the sequence is random). 115
C.2 Statistics table of the Non-Overlapping Template Matching Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hipothesis (the hipothesis that states the sequence is random). 119
C.3 Pass rates test for each one of the NIST Statistical Test Suit [Rukhin et al., 2010] based on 100 tests for 100 10^6 bits number generated by a Luna SA HSM. 124

Chapter 1

Introduction

If you think cryptography is the answer to your problem, then you don't know what your problem is.

Peter G. Neumann

1.1 Cryptography

By the twentieth century, cryptography was considered an art ([Katz and Lindell, 2007]): the construction of efficient codes and the break of the existing ones. All this was conceived through the creativity and skill of each individual. There were no theoretical foundations nor precise definitions of what defines a code to be considered "a good code" or to be considered "a bad code" or what can actually be considered to be a safe code.

The first known use of cryptography was found in artifacts belonging to the era of the Old Kingdom of Egypt, thousands of years before Christ [D'Agapeyeff, 2008]. In those times cryptography referred almost exclusively to encryption, which is the process of converting ordinary information (called plaintext) into unintelligible text (called ciphertext). In its turn, decryption is the reverse of encryption, i.e. the ciphertext is converted to its original and intelligible format. This type of data encryption has been used for thousands of years up until now.

In the twentieth century, however, the encryption concept has undergone a big change. The studies held in cryptography area became more rigorous and it became necessary to formalize some concepts. This gave rise to theoretical foundations in which cryptography should be based. Thus, the study of cryptography started to be seen as a science.

Nowadays cryptography comprises much more than secret communications. Actually, these days, cryptography is used in message authentication, digital signatures, public keys protocols, authentication protocols, electronic auctions and elections as well as digital cash .

Cryptography is defined as the study and practice of techniques capable of providing secure communication even when intersected by a third party [Rivest, 1990]. Thus, cryptography focuses on the construction and analysis of protocols that seek to ensure communication without the risk that an unauthorised outsider can understand the content of the shared information.

These constructions and protocols are related to various aspects of information security such as data confidentiality, data integrity, authentication and non-repudiation.

[Katz and Lindell, 2007] provide a modern cryptography definition as the scientific studies of techniques for securing digital information, transactions, and distributed computations.

A relevant difference between classical and modern cryptography is related to the type of individuals who use each one of them. Classical cryptography had great development and importance in military and in intelligence organizations developments, being these their biggest consumers. Modern cryptography, in its turn, is used worldwide. With computer development and with the advent of the Internet it was necessary to find protection mechanisms to guarantee data security. Since World War I and the advent of the computer, the methods used to carry out cryptology have become increasingly complexes and their application more widespread.

Modern cryptography intersects the disciplines of mathematics, computer science, and electrical engineering. It is heavily based on mathematical theory and computer science practice. Cryptographic algorithms are designed around computational hardness assumptions, making them hard to break, in practice, by any adversary. It is theoretically possible to break such a system, but it is infeasible to do so by any known practical means. These schemes are therefore denominated computationally secure.

Thus, cryptography is the basis of security mechanisms that are an integral part of all computer systems and it is the basis of security provided to each user accessing secure websites by preventing the information from being intercepted by third parties. Modern cryptography became a more and more central topic in computer science.

Modern cryptography is based on some basic principles and paradigms that make the distinction between classical and modern cryptography. The three main principles are:

- The formulation of a rigorous and precise definition of security is the first step to solve any cryptographic problem.

- When the security of a cryptographic construction relies on an unproven assumption, this assumption must be precisely stated. Furthermore, the assumption should be as minimal as possible.

- Cryptographic constructions should carry a rigorous proof of security with respect to a formulated definition according to the first principle, and relative to an assumption stated as in the second principle (if an assumption is needed).

These principles are discussed in greater depth in [Katz and Lindell, 2007].

So, in the beginning, encryption/decryption process was achieved by sending messages personally. With the increase of the amount and distance of communications the method became impossible to use

As stated, the main goal of cryptography is to enable two people, usually referred as Alice and Bob, to communicate over a channel without an outsider, Eve, being able to understand the exchanged messages unless she knows exactly how they were encrypted.This concept is called a cryptosystem and it is formally described using the following mathematical notation:

Definition 1. *A cryptosystem is a five-tuple* (M, C, K, e, d) *where the following conditions are satisfied:*

- M *is a finite set of possible plaintexts;*
- C *is a finite set of possible ciphertexts;*
- K *is the keyspace, i.e., it is a finite set of possible keys;*
- e *is the encryption function* $e : M \times K \to C$ *and* d *is the decryption function* $d : C \times K \to M$*, such that* $d_k(e_k(m)) = m$*, with* $m \in M$ *and* $k \in K$*.*

For communicating, Alice and Bob will choose a random key, $k \in K$. Thus, to send a plaintext m, Alice encrypts it with an *a-priori* defined key, obtaining the ciphertext $c = e_k(m)$. Alice sends c to Bob, who decrypts it by using the same key, obtaining $m = d_k(e_k(m))$. The key that Alice and Bod chose should be only known by the two of them. Otherwise, Eve can decrypt the message just by knowing the decryption function. The encryption function e_k must be an injective function because, otherwise, decryption could not be accomplished in an ambiguous way. For every key k the function d_k is the inverse of the function e_k ([Hoffstein et al., 2008]). It also should be notice that if $M = C$, the encryption function will be nothing but a permutation of the set elements ([Stinson, 2002]).

The fact that the key chosen by Alice and Bob should be known only by themselves defines this kind of cryptosystems as Private Key Cryptosystems ([Stinson, 2002]).

Numerous private key cryptosystems were created increasing their complexity but, eventually they were considered insufficient because this kind of system has as a requirement that the secret key should be known before exchanging the encrypted messages and therefore it is necessary to have a secure channel to exchange the secret key.

Then the challenge arose: would Alice and Bob being able to exchange encrypted messages between them, with Eve knowing all these ciphertexts but without being able to decrypt them? In 1976, Diffie-Hellman algorithm came up as a solution to this problem, predicting a coming revolution in cryptography [Diffie and Hellman, 1976] and proposing a new concept of cryptosystem which would lead to the creation of the RSA algorithm.

1.2 Public Key Cryptography

As said before, the Diffie and Hellman ([Diffie and Hellman, 1976]) publication was an extremely important event. It set forth the basic definitions and goals of a new field of mathematics/computer science. Indeed, their paper begins with: "We stand today on the brink of a revolution in cryptography" [Diffie and Hellman, 1976].

The first important contribution of Diffie and Hellman was the definition of a Public Key Cryptosystem (PKC) and its associated components: one-way functions and trapdoor information. A one-way function is an invertible function easy to compute, but whose inverse is difficult to calculate, meaning that the computation of the inverse function is made in a "reasonable"

amount of time and it will almost certainly fail ([Hoffstein et al., 2008]).For a more detailed discussion about one-way functions see 6.4.

Basically, PKC consists in cryptosystems where there is no need for a secure channel to exchange any prior information. This time, Alice creates a pair of public, e, and private keys, d, revealing her public key and therefore allowing anyone to encrypt messages and send them to her. If Bob wants to receive encrypted messages he should also create a pair of public and private keys and follow Alice's procedure. This way, there is no need for a safe channel to agree on a key since there are no common keys that need to be changed between the several users. Thus, PKC assumes that the knowledge of the public key, e, does not allow computation of the private key, d. It assumes the existence of trapdoor one-way functions, which are functions that are easy to compute but difficult to invert without knowing some extra parameters (in this case, the private key).

Definition 2. [Diffie-Hellman concept of public key cryptosystem]

[Menezes et al., 1996]

Let K be the keyspace, enc be the encryption function and dec be the decryption function.

Consider an encryption scheme consisting of the sets of encryption and decryption transformations $\{enc_e : e \in K\}$ and $\{dec_d : d \in K\}$, respectively.

The encryption method is said to be a public-key encryption scheme if for each associated encryption/decryption pair (e, d), one key, e (the public key), is made publicly available, while the other, d (the private key), is kept secret. By knowing e it should be easy to encrypt the messages and by knowing d it should be easy to decrypt messages. For the scheme to be secure it must be infeasible to compute d from e.

1.3 Diffie-Hellman Algorithm

The Diffie-Hellman algorithm was created considering the protocol created by both (Definition 2) and by taking advantage of the discrete logarithm problem (Definition 3).

Definition 3. [Discrete Logarithm Problem] *[Kiayias, 2009]*

Let $G = \langle g \rangle$ be a multiplicative finite cyclic group of order n. Thus all the element e of G can be written as $e = g^k$, $k \in \mathbb{Z}_n$, and any integer that solves this equality is called a logarithm (in this case a discrete logarithm). G is a cyclic group and integral $\log_g e$ exists for all $e \in G$.

G is finite of order n, so $\log_g e$ is unique only up to congruence modulo n, and the discrete logarithm amounts to a group isomorphism.

$$\log_b : (G; \cdot) \to (\mathbb{Z}_n; +) \tag{1.1}$$

Finding an integer $k \in \mathbb{Z}$ such that $g^k = e$ is called discrete logarithm of base g, and it is a computationally hard problem to solve.

Let consider again Alice and Bob who want to communicate with each other.

Algorithm 1. [Diffie-Hellman] *[Kiayias, 2009]*

Let p be a prime number.

1. *Alice and Bob choose a finite cyclic group, G_p, with p prime and a generator element, g, of G, assuming that g is a public parameter. $G = \mathbb{Z}_p$*
2. *Alice chooses a natural random number, a, and sends $g^a \pmod p$ to Bob.*
3. *Bob also chooses a natural random number, b, and sends $g^b \pmod p$ to Alice.*
4. *Alice computes $(g^b)^a \pmod p$.*
5. *Bob computes $(g^a)^b \pmod p$.*

Since groups are power associative, $(g^b)^a \equiv (g^a)^b \equiv g^{ab} \pmod p$, and in the end, both Alice and Bob have the same value, a value that only the two of them know and that can be used as a secret key. Thus they can use it as an encryption key, known only by them, for sending messages across the same open communication channel.

Diffie-Hellman is a key agreement protocol. It enables the use of any symmetric cipher in order to securely communicate over an open communication channel. For example, suppose m is a message and an element of the group. This message can be encrypted by computing $e = mg^{ab} \pmod p$. Then, it can be decrypted from e doing the computation of $(g^{ab})^{-1}$, using $|G|$, the order of the group. This can be done in the following way: Bob knows G, b, and $g^a \pmod p$. Using the corollary of Lagrange's theorem, the order of a generator of a cyclic group G is $|G|$. If Bob calculates $(g^a)^{|G|-b} = g^{a(|G|-b)} = g^{a|G|-ab} = g^{a|G|}g^{-ab} = (g^{|G|})^a g^{-ab} = 1^a g^{-ab} = g^{-ab} = (g^{ab})^{-1} \pmod p$.

Therefore, Alice sends Bob the encrypted message, $e = mg^{ab}$ and then Bob computes $(g^{ab})^{-1}e = mg^{ab}(g^{-ab}) = m \times 1 = m$, which is usually called ElGamal encryption.

For a greater security it is necessary to choose the numbers a, b and p large enough. If they are large enough, not even the fastest modern computer can find a given only g, p, $g^b \pmod p$ and $g^a \pmod p$. This problem, that needs to be solved, is called the discrete logarithm problem.

Chapter 2

The RSA Public-Key Cryptosystem

All human beings have three lives: public, private, and secret.

Gabriel García Márquez

2.1 RSA Key Generation, Message Encryption and Decryption

The RSA cryptosystem, developed by Rivest, Shamir and Adleman and named after its inventors, is the first ever published public key cryptosystem. The RSA was first presented in their 1978 article [Rivest et al., 1978] and it was based on the Diffie-Hellman proposal. The RSA cryptosystem is the most widely used public key cryptosystem. It may be used to provide both secrecy and digital signatures. Its security is based on the integer factorization problem which is an intractable problem.

Algorithm 2. [RSA key generation]

The keys used on the RSA cryptosystem are generated as follows (with a b bit modulus):

1. *Generate randomly two large prime numbers p and q, each one roughly the same size (b/2) and distinct from one another.*
2. *Compute* $n = pq$ *and* $\Phi(n) = (p-1)(q-1)$*, where* $\Phi(n)$ *is the Euler totient function (number of positive integers less than n which are co-prime with n).*
3. *Select an integer e such that* $1 < e < \Phi(n)$ *and that* $gcd(e, \Phi(n)) = 1$ *(e and* $\Phi(n)$ *are co-prime).*
4. *Use the extended Euclidean algorithm to compute the unique integer d,* $1 < d < \Phi(n)$*, such that* $ed \equiv 1 \pmod{\Phi(n)}$*.*

At the end, the pair (n, e) is the public key and the pair (n, d) is the private key. Nevertheless, p, q, and $\Phi(n)$ must also be kept secret since they can be used to calculate d.

In the first step, the generated numbers can be probabilistic tested for its primality.

Suppose Bob wants to send a message to Alice.

Algorithm 3. [Encryption]

- *Alice broadcasts her public key (n, e) and keeps her private key (n, d) secret.*
- *Bob represents the message as an integer, m in the interval $[0, n-1]$.*
- *Bob computes $c \equiv m^e \pmod{n}$.*
- *The ciphertext c is sent to Alice.*

Alice receives the ciphertext and she can recover the original plaintext using the following algorithm:

Algorithm 4. [Decryption]

- *Alice uses her private key, d, and computes $m = c^d \pmod{n}$.*

2.2 RSA vulnerabilities

2.2.1 The RSA problem

The RSA problem consists in finding the integer m such that $m^e \equiv c \pmod{n}$. Mathematically, the problem presented here is finding the e^{th} roots modulo n.

The conditions imposed for n and e ensure that for each integer $c \in \{0, 1, ..., n-1\}$ there is exactly one and only one $m \in \{0, 1, ..., n-1\}$ such that $m^e \equiv c \pmod{n}$.

Therefore there is a function $f : \mathbb{Z}_n \to \mathbb{Z}_n$ defined as $f(m) = m^e \pmod{n}$ which is nothing rather than a permutation.

It is an open question to know whether there is a polynomial time algorithm that calculates the roots of an equation like $m^e \equiv c \pmod{n}$ ([da Costa Boucinha, 2011]). Nowadays, however

it is assumed that it is hard to calculate such roots when $m \in \mathbb{Z}_n$ is randomly chosen and n is generated by random large primes p and q. In fact, there is no efficient algorithm known for this problem ([Menezes et al., 1996]).

Nevertheless, if the factors of n are known, the RSA problem can be easily solved ([Menezes et al., 1996]), which is explained in detail in the proof of Fact 1.

2.2.2 Large integer numbers factorization

One possible approach which an adversary could employ to solve the RSA problem is factoring n and then compute $\Phi(n)$ and d. This is similar to do the step 4 of the Algorithm 2. Once d is calculated, the adversary can decrypt any ciphertext sent and the RSA problem is solved.

Fact 1. *Let (n, e) be an RSA public key. Given the private key d, one can efficiently factor the modulus $n = pq$. Conversely, given the factorization of n, one can efficiently recover d.*

Proof. ([Boneh, 1999])

Suppose that the private key, d, is known. Given d, $k = de - 1$ can be computed. By definition of d and e, it is known that k is a multiple of $\Phi(n)$ and since $\Phi(n)$ is even, $k = 2^t r$, with r odd and $t \geq 1$. For every $g \in \mathbb{Z}_n^*$, $g^k = 1$ and therefore $g^{k/2}$ is a square root of the unity modulo n. By the Chinese Remainder Theorem, 1 has four square roots modulo $n = pq$. Two of these square roots are ± 1. The other two are equal to x, where x satisfies the conditions $x \equiv 1 \pmod{p}$ and $x \equiv 1 \pmod{q}$. Using one of these last two square roots, not mattering which one, and by computing the $gcd(x-1, n)$, the factorization of n is disclosed. A straightforward argument shows that if g is randomly chosen from $\mathbb{Z}_n^*$, then, with probability at least $1/2$ (over the choice of g), one of the elements in the sequence $g^{k/2}, g^{k/4}, ..., g^{k/2^t} \pmod{n}$ is a square root of the unity, revealing the factorization of n. All elements in the sequence can be efficiently computed in time $\mathcal{O}(n^3)$, where $n = \log_2(n)$.

So, given d, n can be efficiently factored.

Now, let's prove the converse statement.

Suppose the factorization of n is known. Thus, $\Phi(n)$ can be trivially computed. Since e is one part of the public key and $ed \equiv 1 \pmod{\Phi(n)}$, d is easily recovered. Once d is known, all the encrypted messages can be decrypted.

So, exposing the private key and factoring n are equivalent.

□

Although factorization algorithms have been steadily improving, the current state of the art is still far from posing a threat to the security of RSA when RSA is used properly [Boneh, 1999]. In fact, factoring large integers is one of the hardest problems of computational mathematics.

As the RSA modulus n is a large number, factoring it is a difficult task given that its prime factors are randomly and balanced generated. In order to break the system, it will be needed a large amount of time.

Indeed, there are factorization methods for this kind of attack like Fermat's Factorization, Pollard's ρ Algorithm and Elliptic Curve Method. Despite the existence of these methods for computational factorization, they do not run in polynomial time and an appropriate choice of

the size of n makes this factoring attack infeasible. Since this is the most obvious way of trying to break RSA, there are published standards with recommendations for the size of n that should be choose, mostly depending on the amount of time the secret data should be kept secret.

An open question remains and it is to know whether both problems (the RSA and the large integer factorization problems) have the same complexity or not. It is widely believed that the RSA and the integer factorization problems are computationally equivalent, although no proof of this is known [Menezes et al., 1996]. Indeed, if an efficient factoring algorithm exists, then RSA is insecure. The RSA security relies severely on the assumption that both the RSA Problem and the Problem of Factoring Large Integers have no polynomial time algorithm that solves it.

2.2.3 Common modulus

To avoid generating a different modulus $n = pq$ for each different user it could be considered the same n for all users. This n must be provided by a trusted central authority so it could be reliable. This trusted entity defines a modulus n and provides for each user i a pair of public, (n, e_i), and private, d_i, valid RSA keys defined modulus $\Phi(n)$, but not the factorization of n.

Theorem 1. *Let $n = pq$ be a RSA modulus and let (n, e_1), (n, e_2) be two public keys such that $gcd(e_1, e_2) = 1$. Suppose a plaintext m is encrypted with both public keys.*

Knowing $c_1 = m^{e_1} \pmod{n}$, $c_2 = m^{e_2} \pmod{n}$ and the public keys, m can be computed in polynomial time in $\log(n)$.

Proof. e_1 and e_2 are known parameters. Using the Extended Euclidean Algorithm, the integers a_1 and a_2 can be computed, in a way that the equation $a_1 e_1 + a_2 e_2 = 1$ is satisfied. Therefore,

$$c_1^{a_1} c_2^{a_2} \equiv m^{a_1 e_1} m^{a_2 e_2} \equiv m^{a_1 e_1 + a_2 e_2} \equiv m \pmod{n} \tag{2.1}$$

Both the Extended Euclidean Algorithm and the final computation are done in polynomial time in $\log(n)$. Thus, m can be computed in polynomial time in $\log(n)$ ([da Costa Boucinha, 2011]).

□

Therefore, anyone with access to the public keys and the ciphertexts would be able to intercept all the plaintexts which would be encrypted twice to different users (note that the same message has to be encrypted) without factoring the modulus ([Simmons, 1983]).

[Delaurentis, 1984] found out that there is no need for two encryptions of a plaintext to decrypt a message.

Theorem 2. *Let (n, e) be a valid RSA public key with corresponding private key (n, d). Let (n, e_1) be the public key from another user such that $e_1 \neq e$. Then a private key (n, d_1) corresponding to (n, e_1) can be computed by:*

$$d_1 \equiv e_1^{-1} \left(\text{mod } \frac{ed - 1}{gcd(e_1, ed - 1)} \right) \tag{2.2}$$

in polynomial time in $\log(n)$.

Proof. From the RSA algorithm (Algorithm 2), $ed \equiv 1 \pmod{\Phi(n)}$. This equation can be rewritten as $ed - 1 = k\Phi(n)$. e_1 is a public exponent, so is satisfies $gcd(e_1, \Phi(n)) = 1$ and therefore $gcd(e_1, k\Phi(n)) = k'$, for some k' that divides k. Now, let $k'' = \frac{k}{k'}$. The modulus in the equation (2.2) can be written as

$$\frac{ed-1}{gcd(e_1, ed-1)} = \frac{k\Phi(n)}{k'} = k''\Phi(n) \tag{2.3}$$

So, e_1 and d_1 satisfy:

$$d_1 \equiv e_1^{-1} \left(\text{mod } k''\Phi(n)\right) \implies d_1 e_1 \equiv 1 \left(\text{mod } k''\Phi(n)\right) \tag{2.4}$$

Therefore d_1 is a valid private exponent corresponding to e_1. All computations can be done in polynomial time in $\log(n)$.

□

So, when implementing RSA, it is recommended not to use the same modulus for different users.

2.2.4 Blinding

Suppose that (n, d) and (n, e) are Bob's private and public keys, respectively, and that Eve wants Bob's signature on a message $m \in \mathbb{Z}_n^*$ which Bob, obviously, refuses to sign. In order to access the signature Eve picks a random $r \in \mathbb{Z}_n^*$, sets $m' = r^e m \pmod{n}$ and then asks Bob to sign the random message m'. Bob may be agreeable to sign this random message m' and thus providing his signature s'. $s' = (m')^d \pmod{n}$ and Eve just needs to compute $s = \frac{s'}{r} \pmod{n}$ to gather Bob's signature s on the original message m. Indeed,

$$s^e = \frac{(s')^e}{r^e} = \frac{(m')^{ed}}{r^e} \equiv \frac{m'}{r^e} = m \pmod{n}$$

This approach is called *blinding* for the simple reason that Eve can obtain a valid signature on a message of her choice by asking Bob to sign a random "blinded" message that he has no information about what he is actually signing.

This attack is not a serious concern since signature schemes apply a "one-way hash" to the message m prior to signing.

Although this attack is presented as an attack to the RSA, this is actually a good way to see a useful property of this cryptosystem. In cryptography, a blind signature is a type of a digital signature in which the content of a message is blinded before signed and in which the resultant signature can be publicly verified against the original message as any digital signature can. This is a useful property of RSA and it is used, for example, in cryptographic election systems and digital cash schemes.

Chapter 3

The RSA security and cryptanalysis

The magic words are squeamish ossifrage.

Plaintext of the message encoded in RSA-129, given in Martin Gardner's 1977 "Mathematical Games" column about RSA

RSA is used all around the world nowadays, so there have been several attacks and types of attacks on this algorithm. This subsection discusses various security issues related to RSA cryptosystem.

The task faced by a passive adversary is recovering the plaintext m from the corresponding ciphertext c, given the public information (n, e) of the intended receiver.

3.1 Small public exponent

In order to improve the efficiency of encryption, it is desirable to select a small encryption exponent e such as $e = 3$. Various entities may all have the same encryption exponent e, however, each entity must have its own distinct modulus n. In fact, in many practical applications, the encryption process is performed by some limited device, such as a smart card and, in cases like this, raising m to a high power might be quite expensive in terms of battery power, time, ... Fixing the public exponent to be some small number is an attempt to simplify the encryption process. In case $e = 3$, the encryption process simply involves raising a number to the power 3, which can be done using only two multiplications.

This can also be a problem when sending small messages, because if $m < n^{1/e}$, then m can be recovered from the ciphertext $c \equiv m^e \pmod{n}$ simply by computing the e^{th} integer root of c. The method of salting the plaintext messages prevents this kind of problem (this topic is discussed in Chapter 3.1.1).

However, by using a low public exponent, RSA is still considered secure, when used carefully, and for that the current public exponent advised is about $e = 2^{16} + 1$ ([Boneh, 1999]) and the message should be salted with some pseudorandom bits.

3.1.1 Hastad's broadcast attack

The attack presented here is due to [Hastad, 1985] and is known as the Common Plaintext Attack due to the fact that the same plaintext is encrypted more than once, similar to the approach described in Chapter 2.2.3.

For this attack it will be needed k messages, $k > e$, where e is the public exponent used to encode the k messages.

Suppose Bob wishes to send the same message, m, to k beneficiaries. To do that he obtains the beneficiaries public keys (e_i, n_i), for $i = 1, ..., k$, and uses the same public exponent with value equal to 3 to send all the messages. So, $e_i = 3$ for all i. Naively Bob computes $c_i = m^3 \pmod{n_i}$ and sends it to the i^{th} recipient.

A simple argument shows that as soon as $k \geq 3$, the message, m, is no longer secure ([Boneh, 1999]) and that an eavesdrop can recover the message, in this circumstances. Suppose Eve, the attacker, intercepts the encrypted messages c_1, c_2, c_3, where $c_i = m^3 \pmod{n_i}$. Let's assume that $gcd(n_i, n_j) = 1$ for all $i \neq j$, since otherwise it is possible to factorize some of the n_i's. Applying the Chinese Remainder Theorem, Eve can compute $c' \in \mathbb{Z}^*_{n_1 n_2 n_3}$ such that $c' = m^3 \pmod{n_1 n_2 n_3}$. Since $m < n_i$ for all $i = 1, ..., k$, $m^3 < n_1 n_2 n_3$ and thus $c' = m^3$ holds over the integers. This way, Eve recovers m just by computing the cubic root of c' over the integers.

Generally, an attacker can recover any message sent as soon as $k \geq e$. However, this attack is feasible only when a small e is used.

The problem seems to be able to be avoided by never sending the same message, or even the same message with known variations, to more than one entity. To prevent against this kind of attack, a pseudorandomly generated bitstring, independently generated for each encryption, of an appropriate length should be appended to the plaintext message prior to encryption. This process is referred to as *salting* the message.

The following theorem supports the example above.

Theorem 3. *Suppose a plaintext, m, is encrypted k times with the public keys* $(n_1, e), (n_2, e), ..., (n_k, e)$ *where* $k \geq e$ *and* $n_1, n_2, ..., n_k$ *are pairwised co-prime. Let* $n_0 = min\{n_1, n_2, ..., n_k\}$ *and* $n = \prod_{i=1}^{k} n_i$. *If the plaintext satisfies* $m < n_0$ *then, any attacker, knowing* $c_i \equiv m^e \pmod{n_i}$ *and* (n_i, e) *for* $i = 1, ..., k$, *can compute the plaintext m in polynomial-time in* $\log(n)$.

Proof. n_i's are co-prime, so by the Chinese Remainder Theorem it is possible to compute $c \equiv m^e \pmod{n}$. $m < n_0$ so $m^e < n_1 n_2 ... n_k = n$ and thus $c = m^e$. Therefore, all that is needed to do is to compute the e^{th} root of c over the integers to discover m. All computations can be done in polynomial-time in $\log(n)$ ([Boneh, 1999]).

□

[Hastad, 1985] reported a much stronger result that improves the attack described above. Suppose that Bob, knowing about the existence of the previous attack, tries to naively defend his message from it by applying a pad to the message m prior to encrypting it. Therefore, if m is k bits long, Bob can send $m_i = i2^n + m$ and thus the recipients will receive slightly different

messages. Since the attacker obtains encryptions of different messages, he cannot mount the previous attack.

Nonetheless, Hastad proved that this linear padding scheme is not secure. Indeed, he proved that any fixed polynomial applied to the message will result in an insecure scheme, not preventing the attack.

Now, suppose Bob takes another chance to ensure his messages are protected from the attacks described above, once he knows their existence. For each of the recipients Bob has a fixed public polynomial $f_i \in \mathbb{Z}_{n_i}[x]$ and to broadcast a message, m, he sends the encryption of $f_i(m)$ to each one of the beneficiaries. By eavesdropping, Eve learns $c_i = f_i(m)^{e_i} \pmod{n_i}$ for $i = 1, ..., k$. Hastad [Hastad, 1985] showed that if enough parties are involved, Eve can recover the plaintext m from all the ciphertexts.

The following theorem is a stronger version of Hastad's original result.

Theorem 4. [Hastad]

Let $n_1, ..., n_k$ be relatively prime integers and set $n_{\min} = \min_i\{n_i\}$. Let $g_i(x) \in \mathbb{Z}_{n_i}[x]$ be k polynomials of maximum degree q. Suppose there exists a unique $m < n_{\min}$ satisfying

$$g_i(m) = 0 \pmod{n_i} \text{ for all } i \in \{1, ..., k\}.$$

Under the assumption that $k > d$, there exists an efficient algorithm which, given $(n_i, g_i(x))$ for all i, computes m.

Proof. Let $n' = n_1 n_2 ... n_k$ and since n_i's are pairwise relatively prime integers, by the Chinese Remainder Theorem one can compute the coefficients T_i satisfying $T_i \equiv 1 \pmod{n_i}$ and $T_i \equiv 0 \pmod{n_j}$ for all $i \neq j$.

The function $g(x)$ can be set as $g(x) = \sum_i T_i g_i(x)$ and one can see that $g(m) \equiv 0 \pmod{n'}$. T_i are nonzero, so $g(x)$ is not identically zero and if the leading coefficient of $g(x)$ is not one it can be assumed without loss of generality that g is a monic polynomial obtained by multiplying this coefficient by its inverse.

The degree of $g(x)$ is at least q. By the Coppersmith's Theorem (Theorem 14) all integer roots, x_0, that satisfy $g(x_0) \equiv 0 \pmod{n'}$ and $|x_0| < (n')^{\frac{1}{q}}$ can be computed. Knowing that $m < n_{\min} < (n')^{(\frac{1}{k})} < (n')^{\frac{1}{q}}$ it is trivial to conclude that m is such a root.

□

This result can be applied to the problem of broadcasting RSA. Suppose the i^{th} plaintext is padded with a polynomial $f_i(x)$ so that $c_i \equiv (f_i(m))^{e_i} \pmod{n_i}$. The polynomials $g_i(x) = (f_i(m))^{e_i} - c_i \pmod{n_i}$ satisfy the above relation and the message m can be recovered by an attacker from the given ciphertexts whenever $k > \max_i(e_i \cdot deg(f_i(x)))$. Particularly, if Bob sends messages linearly related, Eve, the attacker, can recover the plaintext as soon as $k > e$.

Hastad's original theorem is weaker than the one stated above, requiring $k = \mathcal{O}(q^2)$ messages where $q = \max_i(e_i \cdot deg(f_i(x)))$ ([Boneh, 1999]).

So, a proper way to defend against the broadcast attack above is using a randomized padding in RSA encryption.

Another attack, known as the Related Plaintext Attack, allows for the encrypted messages to be different but related by known polynomials. However, this attack requires a larger number of messages to be encrypted. The result, due to Bleichenbacher, is ([Franklin et al., 1996]):

Theorem 5. *Given the public keys* $(n_1, e_1), (n_2, e_2), ..., (n_k, e_k)$ *where the modulus are pairwise co-prime, and* $f_1(x) \in \mathbb{Z}_{n_1}[x], ..., f_k(x) \in \mathbb{Z}_{n_k}[x]$*, set* $n_0 = \min\{n_1, n_2, ..., n_k\}$ *and* $n = \prod_{i=1}^{k} n_i$*. For a plaintext* $m < n_0$*, if* $k \geq \max_i(e_i \cdot deg(f_i(x)))$ *then given* $c_i = f_i(m) \pmod{ni}$ *and* (n_i, n_i) *for* $i = 1, 2, ..., k$*, the plaintext* m *can be computed in time polynomial in* $\log(n)$ *and* $\max_i(e_i \cdot deg(f_i(x)))$*.*

Proof. Suppose all $f_i(x)$ are monic functions. If they are not, one can turn them into monic functions by multiplying them with the leading coefficients inverse. If the inverse does not exist for $f_j(x)$, one can find a factor of n_j and from c_j we find m. Let us set $\delta = \max_i(e_i \cdot deg(f_i(x)))$ and also $h_i = \delta - deg(f_i(x)^{e_i})$ for $i = 1, 2, ..., k$. The k^{th} monic polynomials of degree δ can be defined as follows:

$$g_i(x) = x^{h_i}(f_i(x)^{e_i} - c_i) \in \mathbb{Z}_{n_i}, \text{ for } i = 1, ..., k$$

Noting that $g_i(m) \equiv 0 \pmod{n_i}$ for $i = 1, ..., k$, and using the Chinese Remainder Theorem with $g_i(x)$ and n_i as inputs, a new degree δ monic polynomial $G(x) \in \mathbb{Z}_n[x]$ can be computed satisfying:

$$G(m) \equiv 0 \pmod{n}$$

where $m < n_0 < n^{\frac{1}{l}} < n^{\frac{1}{D}}$. This makes all the conditions to use Coppersmith's result (Theorem 14). Therefore, m can be computed in time polynomial in $\log(n)$ and δ.

□

May and Ritzenhofen [May and Ritzenhofen, 2008] have improved the bound for the number of ciphertexts required to compute the plaintext. Let $\delta_i = e_i \cdot deg(f_i(x))$. If the inequality

$$\sum_{i=1}^{k} \frac{1}{\delta_i} \geq 1$$

is satisfied then the plaintext m can be recovered from the k ciphertexts.

If transmitting the same (or related) message massively is not possible, one way to prevent this type of broadcast attacks is by ensuring that this last inequality is not satisfied. For this to happen it is only needed to transform the messages with polynomials of high degree or alternatively use high public key exponents.

3.1.2 Franklin-Reiter Related Message Attack

Franklin and Reiter ([Coppersmith et al., 1996]) identified an attack against RSA for a small exponent e if two messages differ only by a known fixed difference between them and if they are RSA encrypted under the same modulus n. Franklin and Reiter stated that in that conditions, it is possible to recover both messages.

Suppose $m_1, m_2 \in \mathbb{Z}_n^*$ are two distinct messages that satisfy $m_1 = f(m_2) \pmod n$ for some $f \in \mathbb{Z}_n[x]$. So, the messages are related under the same modulus. Bob wants to send both messages to Alice and to do that he encrypts the messages and transmits the resulting ciphertexts c_1, c_2. For any small e, Eve can easily recover the original plain texts, m_1, m_2.

For the following lemma, in order to simplify the proof, let us consider $e = 3$.

Lemma 1. [Franklin-Reiter]

Let's set the encryption key $e = 3$ and let (e, n) the RSA public key. Let $m_1 \neq m_2 \in \mathbb{Z}_n^*$ satisfying $m_1 = f(m_2) \pmod n$ for some linear polynomial $f(x) = ax + b \in \mathbb{Z}_n[x]$ with $b \neq 0$ and let c_1, c_2 be the encryption of m_1, m_1, respectively. Then, given (n, e, c_1, c_2, f), an attacker can recover the original plaintexts m_1, m_2 in time quadratic in $\log(n)$.

Proof. There is no need to restrict e in this first part of the proof, so let's consider an arbitrary e and keep this part of the proof general. Since $c_1 = m_1^e \pmod n$, m_2 is a root of the polynomial $g_1(x) = f(x)^e - c_1 \in \mathbb{Z}_n[x]$. Similarly, m_2 is a root of $g_2(x) = x^e - c_2 \in \mathbb{Z}_n[x]$. The linear factor $x - m_2$ divides both polynomials. Therefore, using the Euclidean algorithm the $gcd(g_1, g_2)$ can be computed and if it turns out to be linear, m_2 is found. The great common divisor can be computed in quadratic time in e and $\log(n)$.

When $e = 3$, the great common divisor must be linear. The polynomial $x^3 - c_2$ factors both modulus p and q into a linear factor and an irreducible quadratic factor (recalling that $gcd(e, \Phi(n)) = 1$ and hence $x^3 - c_2$ has only one root in $\mathbb{Z}_n$). Since g_2 cannot divide g_1, the gcd must be linear.

Once the attacker knows (n, e, c_1, c_2, f) he/she can compute

$$\frac{b(c_2 + 2a^3c_1 - b^3)}{a(c_2 - a^3c_1 + 2b^3} \equiv \frac{m_1(3a^3bm_1^2 + 3a^2b^2m_1 + 3ab^3)}{3a^3bm_1^2 + 3a^2b^2m_1 + 3ab^3} \equiv m_1 \pmod n$$

All calculations are done in time polynomial in $\log(n)$.

For $e > 3$ the gcd is almost always linear. However, for some rare m_1, m_2 and for some rare f, it is possible to obtain a nonlinear gcd. In such cases the attack will fail.

□

For $e > 3$ the attack takes time quadratic in e. Consequently, it can be applied only when a small public exponent is used. For larger values of e, the computation of the great common divisor is not feasible.

3.1.3 Coppersmith's Short Pad Attack

Coppersmith showed that a simplistic randomized message padding may put in danger the security of RSA [Coppersmith, 1997]. If a naive random padding algorithm puts in question the

security of RSA and if randomized padding is used with RSA then, a question arises: how many bits of randomness are needed to consider a message properly-padded?

Suppose Bob sends a padded encryption of a message m to Alice using a small random pad before encrypting m. An attacker, Eve, can intercept the transmission of m, preventing it from reaching its destination, prompting Bob to resend the message. Bob, noticing that Alice did not received his message decides to resend m to Alice. In order to do that, Bob sends the message once again, but with a new random pad. Eve now has two ciphertexts corresponding to two encryptions of the same message using two different random pads and even though she does not know the random pads used, she can still recover m if the random pads are too short.

The follow theorem generalises the illustrated situation above:

Theorem 6. *Let (e, n) be a public RSA key where n is n' bits long. Set $k = \left\lfloor \frac{n'}{e^2} \right\rfloor$. Let $m \in \mathbb{Z}_n^*$ be a message of length at most $n' - k$ bits. Define $m_1 = 2^k m + r_1$ and $m_2 = 2^k m + r_2$, where r_1 and r_2 are distinct integers with $0 \leq r_1, r_2 < 2^k$. If an attacker knows (n, e) and the encryptions c_1, c_2 of m_1, m_2 (but is not given r_1 or r_2), he/she can efficiently recover m.*

Proof. Let's assume that the padding is placed in the least significant bits, so that $c_i = (2^k m + r_i)^e \pmod{n}$ for some small k and random $r < 2^k$. Eve now knows

$$c_1 = (2^k m + r_1)^e \pmod{n} \text{ and } c_2 = (2^k m + r_2)^e \pmod{n}$$

for some unknown m, r_1, and r_2. Considering the functions $f(x, y)$ and $g(x, y)$ defining by $f(x, y) = x^e - c_1$ and $g(x, y) = (x + y)^e - c_2$, when $x = 2^k m + r_1$, both of these polynomials have $y = r_2 - r_1$ as a root modulo n. The resultant $h(y) := Res_x(f, g)$ will be of degree at most e^2 and $y = r_2 - r_1$ is also a root of $h(y)$ modulo n. If $|r_i| < \left(\frac{1}{2}\right) n^{\frac{1}{e^2}}$ for $i = 1, 2$ then $|r_2 - r_1| < n^{\frac{1}{e^2}}$. By Coppersmith's Theorem (Theorem 14) all of the roots of $h(y)$ may be computed, which will include $r_2 - r_1$. Once $r_2 - r_1$ is discovered, m can be extracted by using the result of Franklin and Reiter [Coppersmith et al., 1996] (described in 3.1.2).

□

This attack exploits a weakness of RSA with a public exponent $e = 3$. However, when $e = 3$ the attack can be mounted as long as the pad length is less than $\left(\frac{1}{9}\right)^{th}$ the message length. But for the recommended values of e the attack is useless against standard moduli sizes.

3.1.4 Partial Key Exposure Attack

The main goal of this section is to study the following question: how many bits of d does an attacker require in order to reconstruct all of d? Surprisingly, for a short public exponent an adversary only needs a quarter of significant bits of d to efficiently recover all of d. Thus, the RSA cryptosystem, particularly with a small public exponent, is vulnerable to partial key exposure.

Lets consider a computer system which has an RSA private key stored on it. To obtain the private key an adversary may attempt to attack the system in a variety of ways. Some of

these attacks are able to reveal some bits of the key, though they may fail when the entire key is wanted to be revealed. However, considering this results, any attack done in order to reveal the private key only needs to be carried out until a quarter of the least significant bits of d are exposed. Once these bits are revealed the attacker can effortlessly compute the entire private key.

Another scenario where partial key exposure comes up is in the presence of covert channels. Such channels are often slow or have a bounded capacity. Some results show that as long as a fraction of the private exponent bits can be leaked, the remaining bits can be reconstructed. These results can be found in [Durfee, 2002] and [Boneh, 1999].

3.2 Low private exponent

The decryption time or signature-generation time can be reduced by using a small value of d rather than a random d. The use of a small value of d can improve the performance by at least a factor of 10 for a 1024 bit modulus since modular exponentiation takes linear-time in $\log_2(d)$. However, [Wiener, 1990] shows that the choice of a small d will result in an insecure system in which an attacker can recover all secret information.

3.2.1 Wiener's Attack

Theorem 7. [Wiener's theorem]

Let $n = pq$ with $q < p < 2q$. Let $d < \frac{1}{3}n^{\frac{1}{4}}$. Given (n, e) with $ed \equiv 1 \pmod{\Phi(n)}$, the attacker can efficiently recover d.

Proof. The proof is based on approximations using continued fractions (see). Since $ed \equiv 1 \pmod{\Phi(n)}$, there exists a k such that $ed - k\Phi(n) = 1$. Therefore

$$\left|\frac{e}{\Phi(n)} - \frac{k}{d}\right| = \frac{1}{d\Phi(n)}.$$

Hence, $\frac{k}{d}$ is an approximation of $\frac{e}{\Phi(n)}$. Although the attacker does not know $\Phi(n)$, he may use n to approximate it. Since $\frac{\sqrt{n}}{2} < q < p < 2\sqrt{n}$, then $p + q - 1 < 3\sqrt{n}$ and thus $n - \Phi(n) < 3\sqrt{n}$. If $\frac{e}{n}$ is used as an approximation,

$$\begin{aligned}
\left|\frac{e}{n}-\frac{k}{d}\right| &= \left|\frac{ed-kn}{nd}\right| \\
&= \left|\frac{ed-k\Phi(n)-kn+k\Phi(n)}{nd}\right| \\
&= \left|\frac{1-kn-\Phi(n))}{nd}\right| \\
&\leq \left|\frac{3k\sqrt{n}}{nd}\right| \\
&= \frac{3k\sqrt{n}}{\sqrt{n}\sqrt{n}d} \\
&= \frac{3k}{d\sqrt{n}}
\end{aligned}$$

Now, $k\Phi(n) = ed - 1 < ed$, so $k\Phi(n) < ed$. Since $e < \Phi(n)$, $k\Phi(n) < ed < \Phi(n)d$. Then, $k\Phi(n) < \Phi(n)d \implies k < d$.

Since $k < d$ and by hypothesis $d < \frac{1}{3}n^{\frac{1}{4}}$. Thus,

$$\left|\frac{e}{n}-\frac{k}{d}\right| \leq \frac{1}{dn^{\frac{1}{4}}}$$

Since $d < \frac{1}{3}n^{\frac{1}{4}}$ and $2d < 3d$, then $2d < 3d < n^{\frac{1}{4}}$ and $2d < n^{\frac{1}{4}} \implies \frac{1}{2d} > \frac{1}{n^{\frac{1}{4}}}$.

Concluding,

$$\left|\frac{e}{n}-\frac{k}{d}\right| \leq \frac{3k}{d\sqrt{n}} < \frac{1}{d \cdot 2d} = \frac{1}{2d^2}$$

This is a classic approximation relation. The number of fractions $\frac{k}{d}$ with $d < N$ approximating $\frac{e}{n}$ so closely is bounded by $\log_2(n)$ ([Boneh, 1999]). In fact, such fractions are obtained as convergents of the continued fraction expansion (Appendix B) of $\frac{e}{n}$ and all that is needed to do is compute the $\log(n)$ convergents of the continued fraction for $\frac{e}{n}$. One of the computations will be equal to $\frac{k}{d}$ and since $ed - k\Phi(n) = 1$, which means that $gcd(k, d) = 1$, $\frac{k}{d}$ is a reduced fraction. This is a linear-time algorithm for recovering the secret key d.

□

So, Wiener has proved that the attacker may efficiently find d when $d < \frac{1}{3}n^{\frac{1}{4}}$ [Boneh, 1999].

Algorithm 5. Wiener's Attack Algorithm

---

1. *Given the public key* (e, n) *compute the continued fraction expansion of* $\frac{e}{n}$*, finding the possible values for the convergences* $\frac{k}{d}$.
2. *Verify the convergent calculated before produce a factorization of* n*:*

- *Compute* $\Phi(n) = \dfrac{e.d-1}{k}$.
- *Solve the equation* $x^2 - ((n - \Phi(n)) + 1)\,x + n = 0$. *Then the roots of this equation are the factors* p *and* q *of* n.

Notice that, depending on n, Wiener's Theorem will work only if $d < \dfrac{n^{\frac{1}{4}}}{3}$.

Although Wiener found a way to attack the RSA algorithm he also presented some countermeasures against his attack that allow a fast decryption without the need of a small private exponent. Two techniques are described:

- **Choosing a large public key**: Instead of reducing e modulo $\Phi(n)$ replace e by e' , where $e' = e + t\Phi(n)$ for some large value of t. When e' is large enough, i.e., when $e' > n^{\frac{3}{2}}$, Wiener's attack cannot be applied regardless of how small d is since the value of k in the proof above is no longer small.
- **Using the Chinese Remainder Theorem**: Suppose one chooses d such that both $d_p \equiv d \pmod{(p-1)}$ and $d_q \equiv d \pmod{(q-1)}$ are small values but d, itself, is not. Then, a fast decryption of the encrypted message can be done as follows:
  1. Compute $m_p \equiv c^{d_p} \pmod{p}$ and $m_q \equiv c^{d_q} \pmod{q}$.
  2. Use the Chinese Remainder Theorem to compute the unique value of $m \in \mathbb{Z}_n$ that satisfies $m \equiv m_p \pmod{p}$ and $m \equiv m_q \pmod{q}$. The result of m satisfies $m \equiv c^d \pmod{n}$. Although the values of d_p and d_q are small, the value of $d \pmod{\Phi(n)}$ can be a large one and Wiener's attack does cannot be applied to those situations.

These purposed methods are not proved to be secure, but it is proved that Wiener's attack is ineffective against them.

3.3 Bleichenbacher's Attack on PKCS#1

This attack focuses the achievement of its goal by attacking the implementation of RSA rather than attacking the underlying structure of the RSA cryptosystem.

Let n be a n' bit RSA modulus and m be a m' bit message with $m' < n'$. Before applying the RSA encryption, a common practice is to pad the message m to n' bits by appending random bits to it. The old version of PKCS#1 (Public Key Cryptography Standard #1) uses this padding approach.

After padding the message m, the message will look as follows:

00	02	padding string	00	m

The resulting message is n' bits long and it is directly encrypted using the RSA cryptosystem. The random pad is recognised due to the presence of "02" on the initial block of the message that will be encrypted. This initial block is 16 bits long.

Suppose that Bob's machine receives a PKCS#1 message. An application will decrypt the message and then it will check the initial block for stripping of the random pad. However, some applications send *"invalid ciphertext"* as an error message when the message do not contain the "02" initial block. Bleichenbacher [Bleichenbacher, 1998] showed that this error message can lead to catastrophic repercussions: an attacker can decrypt any ciphertext by using the error message.

Suppose there is a message c intended for Bob that is intercepted by Eve, who wants to decrypt it. To mount the attack, Eve chooses a random $r \in \mathbb{Z}_n^*$, computes $c' = r \cdot c \pmod{n}$ and sends c' to Bob's machine. The application that intends to decrypt all the messages received attempts to decrypt c'. If c' is properly formatted the machine will give no response, but if c' is not properly formatted it will respond with an error message. Hence, Eve gets to know whether the 16 most significant bits of the decryption of c' are equal to 02. In effect, Eve has an *oracle* that tests for her the 16 most significant bits of the decryption of $r \cdot c \pmod{n}$, for any r of her choice. Bleichenbacher showed that such an oracle is sufficient for decrypting c ([Bleichenbacher, 1998]).

Chapter 4

Factorization

It's all very simple. But maybe because it's so simple, it's also hard.

Natsuki Takaya

The integers factoring problem comes down to compute a non-trivial factor of a given integer. In current days various rigorous and fast methods are known to solve the integers factoring problem. However, these methods prove to be inefficient as the size of the number to be factor increases.

Many encryption methods used to convey confidential information securely via an open channel based their security on the hard problem of factoring large integers, including the RSA cryptographic method, as seen previously in Section 2.2.2. In fact, the RSA cryptosystem has become widely used for the transmission of confidential information due to its security relies in the integer factorization problem, since there are no fast and efficient methods for the factorization of n, the product of two large primes.

Thus, if it is possible to find an efficient method for factoring large integers, the RSA security is compromised. Moreover, the study of factorization methods proves to be extremely important since the more one knows about the existing factorization methods, the more securely the required parameters can be chosen in order to hinder the factorization of n and thus improving the safety of the method.

4.1 Trial Factorization

Let n be the number to be factored. $\sqrt{n}$ is calculated and all primes p smaller $\sqrt{n}$ are tested until the condition $gcd(p, n) \neq 1$ is satisfied. Then, p is one factor of n. In the worst scenario, the number of tests ran is equal to the number of primes, p, smaller than $\sqrt{n}$.

$\pi(x)$ is a function that counts the number of primes smaller than x. It's known that for large values of x, $\pi(x) \sim \frac{x}{\ln x}$ ([Menezes et al., 1996]). Thus, in the worst scenario, the algorithm is executed in $\mathcal{O}\left(\frac{2^{|n|/2}}{|n|}\right)$ steps.

4.2 Fermat Factorization

Almost all of the classic factorization algorithms are based on the Fermat factorization.

The Fermat factorization assumes the possibility of, given a number n to be factored, n can be written as the difference between two squares.

In the Fermat factorization algorithm, the main goal is to find u and v, integer, such that $u \neq \pm v \pmod{n}$ and $u^2 = v^2 \pmod{n}$. Once u and v are known, one has $(u-v)(u+v) = 0 \pmod{n}$. Since $(u \pm v) \neq 0 \pmod{n}$, at least one of the values of $gcd(u-v, n)$ and $gcd(u+v, n)$ is not equal to 1 and this is going to be a factor of n.

Algorithm 6. [Fermat's Factorization Algorithm]

Given a composite integer n,

1. *Set* $w = \sqrt{n}$ *and* $x = w$
2. *Set* $y = \left\lceil \sqrt{x^2 - n} \right\rceil$
3. *If* $n = x^2 - y^2$, *output* $x - y$ *and* $x + y$
4. *If* $x < n$, *replace* x *by* $x + 1$ *and go to step 2.*

Checking the last step, one can conclude that this algorithm will try all integers from $\sqrt{n}$ to n, which makes this algorithm very slow, and in the worst case scenario, it will take $\mathcal{O}(\sqrt{n})$ steps to finish, meaning it is even worst than trial division.

4.3 Continued Fraction Method

In 1931 one of the first modern approaches to the problem of factoring large integers appeared. That approach gave rise to the continued fractions method, also known as the CFRAC method.

Due to its computational difficulty, this method, described by D. Lehmer e R. Powers ([Lehmer and Powers, 1931]), only gained popularity and applicability with the advent of the first computers, being developed as a computer algorithm by Michael A. Morrison and John Brillhart in 1975 ([Morrison and Brillhart, 1975]), who managed to factorize the Fermat number $F_7 = 2^{2^7} + 1$.

Suppose one wants to factor an odd and compound integer n.

The result presented on Theorem 15 implies that

$$p_k^2 = (-1)^{k+1} Q_{k+1} \pmod{n}$$

The above equation has the conditions to apply the Fermat Theorem if one squares both sides of the equation.

The idea of continued fractions is to generate pairs (p_k, Q_{k+1}) and take suitable combinations to produce a square on the right and to possibly factor n. An integer is a perfect square if and only if the exponents in the prime factorization are all even. Hence, in order to find the products of Q_k that yield perfect square, their prime factorization needs to be calculated and combine their factors so that the exponents become even. In that way, the factorization of Q_k is obtained by trial division.

A set of primes over which Q_k factors is selected and then a factor base (Definition 12) is found.

Algorithm 7. [The CFRAC Method]

---

1. *Expand $\sqrt{n}$ (or $\sqrt{cn}$) into a simple continued fraction expansion to some point m. Thus, it follows that $\sqrt{n} = [a_0, a_1, a_2, \cdots, a_m]$, $m \in \mathbb{N}$ (defined in Appendix B).*
2. *Generate $p_k - Q_k$ pairs.*
3. *Among the set of $p_k - Q_k$ pairs previously generated find certain subsets (the S-sets) that have as property the product $\prod_i (-1)^i Q_i$ of its Q_i's to be a square. If no such set is found go to Step 1*
4. *For each S-set found calculate the congruence $X^2 \equiv \prod_i p_i \equiv \prod_i (-1)^i Q_i = Y^2 \pmod{n}$, where $1 \leq X < n$*
5. *Compute Y and the $gcd(X - Y, n) = d$ for each one of the S-sets. If $1 < d < n$ for some S-set, the method succeeds and d is a non-trivial factor of n. Otherwise, return to the first step*

Although the CFRAC method has been a great accomplishment in the 30's, nowadays, due to its limitations, this method has fallen into disuse. Yet their study is of great importance in the context of the integer factorization problem, since many of the CFRAC method ideas are the inspiration for the subsequent algorithms, often used nowadays.

4.4 Pollard's $p-1$ Factorization Algorithm

The Pollard's $p-1$ factorization method is not efficient for all numbers. However it is very efficient for certain kinds of numbers.

Let $n = pq$ be the number to be factored. The factorization result will be the factors p and q. Suppose that an integer L was found with the properties

$$p-1 \text{ divides } L \quad \text{and} \quad q-1 \text{ does not divide } L$$

which implies that $L = i(p-1)$ and $L = j(q-1)+k$, with $i, j, k \neq 0$ integers.

Applying Fermat's Little Theorem to a randomly chosen a to compute a^L,

$$a^L = a^{i(p-1)} = (a^{p-1})^i \equiv 1 \,(\mathrm{mod}\, p)$$
$$a^L = a^{j(q-1)+k} = a^k (a^{q-1})^j \equiv a^k \,(\mathrm{mod}\, q)$$

If a given number w is congruent to 1 modulo a factor of n, then the $gcd(w-1, n)$ will be divisible by that factor.

Without loss of generality, let's consider only the case of the exponent $p-1$. The idea of the Pollard's $p-1$ factorization algorithm is to turn de exponent of a into a large multiple of $p-1$. Thus, this will be a number with many prime factors. Generally, in order to achieve the increase of the product of all prime powers less than some limit, r is taken. Hence, it starts with a random a and it is repeatedly replaced by $a^x \,(\mathrm{mod}\ n)$ as x runs through those prime powers. At each stage $gcd(a-1, n)$ is checked to be or not to be equal to 1.

Algorithm 8. [Pollard's $p-1$ Factorization Algorithm]

---

1. *Set a with some convenient value. Let's say $a = 2$*
2. *For some $r \in \mathbb{N}$, for $j = 1, 2, 3, \cdots, r$*
   - *Set $a = a^j \,(\mathrm{mod}\, n)$*
   - *Compute $d = gcd(a-1, n)$*
   - *If $1 < d < n$ the algorithm was succeed. Return d*

   increment j

The running time of this algorithm is $\mathcal{O}(r \log r \log_2 n)$. Thus, although the algorithm is more likely to produce a factor of n with larger values of r, the algorithm will run slower with those values.

Pollard's $p-1$ factorization algorithm may not be the most efficient algorithm to factor large numbers. However, this approach is important in relation to the construction of cryptographic algorithms that base their security in the hard problem that is to factor large integers, like RSA does.

To avoid the dangers of Pollard's $p-1$ method when creating RSA keys one just need to check if the chosen secret primes p and q have the property that neither $p-1$ nor $q-1$ factors entirely into small primes.

4.5 Elliptic Curve Method

In 1985, Hendrik Lenstra had the thought of using Elliptic Curves to factor integers. There are assumptions that the method developed by Lenstra is based on Pollard's $p-1$ method (Chapter 4.4), which is a variant of Pollard's p method. This method is a fast, sub-exponential running time algorithm for integer factorization that operates with elliptic curves.

The Lenstra's Elliptic Curve Algorithm is describe below:

Algorithm 9. [Lentra's Elliptic Curve Algorithm]

---

1. *Choose integers* a, x_1, y_1 *such that* $1 < a, x_1, y_1 < n$
2. *Let* E *be the elliptic curve* $E : y^2 = x^3 + ax + b$ *where* $b = y_1^2 - x_1^3 - ax_1 \pmod{n}$ *and set* $P = (x_1, y_1) \in E$
3. *For some* $r \in \mathbb{N}$ *compute, for* $j = 1, 2, 3, \cdots, r$

$$jP \pmod{n} = \sum_{j=1}^{r} P \pmod{n} = r!P$$

and set $P = Q$

At each stage of the previous addiction, there are three things that may happen:

- *The calculation of the sum can be successfully computed. Go to step 1 and choose another curve and another point*
- *Along the computation of the sum there may be necessary to find the reciprocal of a number,* d*, that is a multiple of* n*, which would not be helpful (this case is very unlikely to occur). Go to step 1 and choose another curve and another point*
- *Along the computation of the sum there may be necessary to find the reciprocal of a number,* d*, that satisfies the condition* $1 < gcd(d, n) < n$*. In this case the computation of* $r!P$ *fails but the value of* $gcd(d, n)$ *is a non-trivial factor of* n *and the algorithm was succeed. A non-trivial factor of* n *is calculated. Return the non-trivial factor*

The time complexity revolve around the size of the prime, p, rather than revolving around n itself. The running time is $\mathcal{O}(e^{\sqrt{2}(\log p)^{1/2}(\log\log p)^{1/2}})$ ([Hoffstein et al., 2008]), where n is the integer to be factored and p is smallest prime factor. This dependence makes the Elliptic Curve Method ideal for finding moderately large factors of large integers. In real cases, this method has been used to find 43-digits factors, but nothing larger.

4.6 Quadratic Sieve

The Quadratic Sieve (QS), was invented by Carl Pomerance in 1981 and was the fastest known factoring algorithm until the Number Field Sieve was discovered in 1993 ([Pomerance, 1982]). Still, the QS is faster than the Number Field Sieve for numbers up to 110 digits long.

Let n be the number to be factor. The goal of the QS method is to find two numbers x and y such that $x \not\equiv \pm y \pmod{n}$ and $x^2 \equiv y^2 \pmod{n}$, which implies that $(x-y)(x+y) \equiv 0 \pmod{n}$. The Euclidean algorithm is then used to check if $gcd(x-y, n)$ is a non-trivial factor of n. To do this, one has to define

$$Q(x) = (x + \lfloor\sqrt{n}\rfloor)^2 - n = \tilde{x}^2 - n$$

and then compute $Q(x_1), Q(x_2), \cdots, Q(x_k)$ and, from all of these, one of them is chosen such that $Q(x_{i_1})Q(x_{i_2})\cdots Q(x_{i_r})$ is a square, y^2.

To check if the product above is a square, the exponents of the prime factors of the product need to be all even. Thus, $Q(x_i)$ needs to be a factor and, in order to make this task easier $Q(x_i)$ needs to be small as well as the prime numbers which factor it need to be small. Those prime numbers compose a factor base (Definition 12), $\mathcal{B}$. To make $Q(x)$ small, the selection of x needs to be close to zero. Thus, a bound, M, needs to be set and only consider the values of x over the sieving interval $[-M, M]$. If $x \in [-M, M]$ and if some prime p divides $Q(x)$,

$$(x + \lfloor\sqrt{n}\rfloor)^2 \equiv n \pmod{p}$$

and thus n is a quadratic residue (mod p) and the primes in $\mathcal{B}$ must be primes such that the Legendre symbol,

$$\left(\frac{n}{p}\right) = 1$$

Some values of $x \in [-M, M]$ are chosen, $Q(x)$ is calculated and the result is checked to see if it totally factors over $\mathcal{B}$. If the result is positive, it is said to have smoothness, but if the result is not positive, the element is excluded and the next element of the sieving interval is chosen.

The most efficient way to do those computations is to work in parallel with the entire sieving interval, since checking out all the primes of a large factor base is an inefficient task to do separately.

Since $Q(x) \equiv \tilde{x}^2 \pmod{n}$ for all x,

$$Q(x_{i_1})Q(x_{i_2})\cdots Q(x_{i_r}) \equiv (x_{i_1}x_{i_2}\cdots x_{i_r})^2 \pmod{n}$$

If all the conditions hold, the factors of n are found.

4.7 General Number Field Sieve

The General Number Field Sieve (GNFS) algorithm can be seen as an improvement to the Quadratic Sieve (Chapter 4.6) algorithm.

The GNFS is an incredibly complex algorithm which uses results from several fields of mathematics. Only a brief explanation is going to be described in this chapter and for an extensive explanation the reading of [Pomerance, 1996] is recommended.

To factor a number n, two polynomials with integer coefficients $f(x)$ and $g(x)$ of degrees d and e, respectively (d and e should be small integers), are chosen. The chosen polynomials should be irreducible over the rationals and the equations $f(x) = 0 \pmod{n}$ and $g(x) = 0 \pmod{n}$ must have one common solution, m.

Let r_1 and r_2 be the roots of f and g and $\mathbb{Z}[r_1]$ and $\mathbb{Z}[r_2]$ the field rings over r_1 and r_2. The functions f and g have degree d and e, respectively, and all its coefficients are integers. Thus, for a, b integers, $r = b^d f(a/b)$ and $s = b^e g(a/b)$ are also integers. The aim of the GNFS algorithm is to find the values of a and b that simultaneously make r and s smooth (Definition 13) relative to the chosen factor basis (Definition 12).

Finding enough pairs of a and b in the above conditions, by the application of the Gaussian elimination, the products of r and of the corresponding s can be obtained so that they are squares at the same time.

m is a root of f and $g \pmod{n}$, hence there are homomorphisms from the rings $\mathbb{Z}[r_1]$ and $\mathbb{Z}[r_2]$ to the ring $\mathbb{Z}/\mathbb{Z}_n$, which map r_1 and r_2 to m. These homomorphisms will map each square root previously found into its representative integer. The product of the factors $a - mb \pmod{n}$ can be obtained as a square for each homomorphism. Thus, two numbers can be found, x and y, with $x^2 - y^2$ divisible by n. The factor of n can be found by calculate the greatest common divisor of n and $x - y$.

The GNFS is the fastest known general purpose method for factoring large integer numbers. The GNFS complexity is $\mathcal{O}(e^{1.93(\log n^{1/3} \cdot (\log \log n)^{2/3})})$ ([Boneh, 1999]).

4.8 RSA Modulus Factorization

The security of the RSA cryptosystem relies on the difficulty of factoring large integers. A 512-bit RSA moduli would be feasible to factor. In fact, in 1999 [Cavallar, 2000] factored a 512-bit RSA modulus using the Number Field Sieve. A 1024-bit modulus is suggested for corporate usage. If the aim is a permanent use, a 2048-bit is suggested. These suggestions have already considered the possible advances in factoring techniques and the speed processor increasing. [Riesel, 1994] showed that an algorithm that factor integers and runs practically in polynomial time is possible to create.

A revolutionary proposal came from [Shor, 1997] who discover an algorithm capable to factor integers in polynomial time if a quantum computer is ever built with a sufficient number of qubits. If ever this accomplishment is achieved, RSA is expected to fall into disuse in benefit of other encryption schemes, as the moduli required to RSA to be secure would be much larger than what would be convenient.

Chapter 5

Recent Attacks on RSA Keys

He attacked everything in life with a mix of extraordinary genius and naive incompetence, and it was often difficult to tell which was which.

Douglas Adams

An efficient attack to RSA keys was successful performed on certified smart cards of Taiwan, in 2013. This attack was inspired by the results of [Heninger et al., 2012] and [Lenstra et al., 2012].

In 2003, Taiwan introduced an e-government initiative to provide a national public-key infrastructure for all citizens. This national certificate service allows citizens to use a "smart" ID card to digitally authenticate themselves to government services. The RSA keys are generated by the cards, digitally signed by a government authority, and placed into an online repository of "Citizen Digital Certificates". On some of the smart cards the random number generators used for key generation are fatally damaged, and have generated real certificates containing keys that provide no security whatsoever.

The attack consisted on finding the factorization of RSA generated modules and it was performed in distinct 1024-bit RSA keys downloaded from Taiwan's national "Citizen Digital Certificate" database. Those keys were generated by government-issued smart cards that have built-in hardware a random number generators that are advertised as having passed FIPS 140-2 Level 2 certification ([FIP, 2002]). The attack that led to the [Bernstein et al., 2013] paper efficiently factored 184 distinct RSA keys out of more than two million 1024-bit RSA keys in the conditions described before. Among the 184 keys factored, 103 of them share prime factors and are efficiently factored by a batch-GCD (see Chapter 5.1.1) computation, which is the same type of computation that was used in 2012 by the two independent teams [Heninger et al., 2012] and [Lenstra et al., 2012] to factor tens of thousands of cryptographic keys on the Internet. If 103 RSA keys shared prime factors then, the left 81 keys of the 184 factored keys do not share primes and, in order to factor them, a deeper advantage of randomness-generation failures is required: first the shared prime numbers are used as a springboard to characterize the failures and then the Coppersmith-type partial-key-recovery attacks are used. This attack had a great impact due to be the first successful public application of Coppersmith-type attacks to RSA keys

found in the wild.

The efficiency of this attack has inspired the attempt to apply the same attack on the RSA keys provided by a Luna SA hardware security module (HSM) that has also certifications of FIPS 140-2, Level 2 among with Level 3, approved DRBG (SP 800-90 CTR mode), Common Criteria EAL4+ and BAC & EAC ePassport Support, which is mainly used for cryptographic purposes. Thus, a similar approach was carried out in the HSM Luna SA generated RSA keys.

In the first part of this chapter the whole process of the attack performed in [Bernstein et al., 2013] attack will be explained. Then the results on the application of this attack on RSA keys generated by HSM Luna SA will be discussed.

5.1 How the attack works

The bottom line of [Bernstein et al., 2013] work was the use of the basic attack used in [Heninger et al., 2012] and in [Lenstra et al., 2012]. In those works, the two teams, working individually, exploited bad randomness to break tens of thousands of keys of SSL certificates on the Internet, a similar number of SSH host keys and a few PGP keys. So, [Bernstein et al., 2013] started by scanning the pairs of distinct keys that share a common divisor using the batch-GCD attack (see Chapter 5.1.1). The calculated shared primes provide enough data to build a model of the prime generation procedure, since there are visible patterns of non-randomness in the primes generated by the smart cards. The next step was to do an extrapolation from those primes: [Bernstein et al., 2013] hypothesised a particular model of randomness-generation consistent with some of the common divisors. In this approach they were able to generate 164 different primes, and to test all of those batch trial division. With the obtained results they successfully factored further keys.

There are also several prime factors that are similar to the 164 patterns but that contain sporadic errors: some bits are flipped here and there, or there exists short sequences of altered bits. Thus, [Bernstein et al., 2013] mounted several Coppersmith-style lattice-based partial-key-recovery attacks to efficiently find prime divisors close to the patterns (Chapter 5.1.2). Those attacks produced various additional factorizations, raising the total factored keys to 184. At the end, nearly half of the factored keys did not share any common divisors with the other keys. Most of them were factored by the Coppersmith-style attacks.

From this attack, the idea of centrally testing RSA moduli for common divisors as a mechanism to detect some types of randomness-generation failure was endorsed. Since finding repeated prime numbers is more than an indication that those RSA keys are vulnerable. In fact, that shows that the underlying randomness-generation system is malfunctioning. However, an absence of common divisors is not an indication of security. There are many potential vulnerabilities resulting from bad randomness. It is important to thoroughly test every component of a random number generator, not merely to look for certain types of extreme failures.

5.1.1 Batch-GCD

The approach used in [Bernstein et al., 2013] was the same approach used in [Heninger et al., 2012] and [Lenstra et al., 2012] for detecting common factors in a collection of the Citizen Digital Certificates RSA keys.

Let $N_1 = pq_1$ and $N_2 = pq_2$ be two distinct RSA moduli that share exactly one prime factor p. The greatest common divisor (GCD) of N_1 and N_2 will be p.

Computing the GCD is fast, and dividing it out of N_1 and N_2 produces the other factors q_1 and q_2. This type of vulnerability should never arise in properly generated RSA keys, but [Heninger et al., 2012] and [Lenstra et al., 2012] observed weak random number generators producing keys with repeated factors in the wild, as well as [Bernstein et al., 2013] observed repeated factors among the Citizen Digital Certificates.

Instead of the naive quadratic-time method of doing the GCD computation (checking each N_1 against each N_2), [Bernstein et al., 2013] used a faster batch-GCD algorithm that use product and remainder trees described in [Heninger et al., 2012] and [Bernstein, 2004].

Briefly, the batch-GCD can be described as follows ([bat, 2012]).

Given a n length sequence X of positive integers, the batch-GCD algorithm computes the follow sequence:

$$
\begin{aligned}
&gcd\{X[0], X[1]X[2]X[3]\dots X[n-1]\}\\
&gcd\{X[1], X[0]X[2]X[3]\dots X[n-1]\}\\
&gcd\{X[2], X[0]X[1]X[3]\dots X[n-1]\}\\
&\vdots\\
&gcd\{X[n-1], X[0]X[1]X[2]\dots X[n-2]\}
\end{aligned}
$$

The computation of these greatest common divisors allows checking which integers share primes with other integers in the sequence X. If $gcd\{X[0], X[1]X[2]X[3]\dots\} = 1$, then $X[0]$ has no primes shared with any of $X[1]$, $X[2]$, $\dots$, $X[n-1]$. If $gcd\{X[0], X[1]X[2]X[3]\dots\}$ is larger than 1, then $X[0]$ must have at least one shared prime. This gcd will often simply equal the shared primes. If $X[0]$ shares one prime with $X[1]$ and another prime with $X[2]$ then more calculations are required to separate those primes and this can become a speed problem if such sharing is very common. However, this problem can be handled at tolerable speed by more complicated algorithms for "factoring into coprimes".

The batch-gcd algorithm is much faster than separately computing each gcd, or the gcd of each pairs.

Algorithm 10. [Batch-GCD Algorithm]

The batch-gcd algorithm has three steps:

1. *A product tree is used to efficiently compute the product*

$$Z = X[0]X[1]X[2]X[3]\dots X[n-1]$$

2. *A remainder tree is used to efficiently compute*

$$Z \left(\text{mod } X[0]^2\right), Z \left(\text{mod } X[1]^2\right), \dots, Z \left(\text{mod } X[n-1]^2\right)$$

3. *The i^{th} remainder is divided by $X[i]$, and the gcd of the result is computed with $X[i]$.*

Experimental Results

In this work the implementation of the batch-GCD algorithm was done in Java 8 using the Eclipse STS interface. The implemented algorithm ran in a TOSHIBA computer with Windows 8.1 Pro 64 bits operating system, 8192Mb RAM and a Intel(R) Core(TM) i5-2520M CPU 2.50GHz (4 CPUs). The implementation ran over ten batches of 5000 distinct 1024-bit RSA moduli and no common factors were found. The same approach was taken for 512-bit RSA moduli and 2048-bit RSA moduli and no common factors were found neither. The computation for each batch of 5000 512-bit RSA moduli it took about 1 hour to finish; for the 5000 1024-bit RSA moduli took about 5 hours and for the 5000 2048-bit RSA moduli took about 12 hours to finish.

5.1.2 Coppersmith-style attacks

Coppersmith's method ([Coppersmith, 1996] and [Coppersmith, 1997]) is a method based in the use of lattice basis reduction ([Menezes et al., 1996]) that factors RSA moduli if at least half of the top bits of the primes are known. Coppersmith's method runs in polynomial-time.

[Bernstein et al., 2013] uses the Univariate Coppersmith attack and the Bivariate Coppersmith attack. Although none of these attacks has been used in the modulus produced by the HSM Luna SA, a brief description of each one of these attacks, based on the description present in [Bernstein et al., 2013] is done. The attacks had not being used in the modulus produced by the Luna SA, since no common factor were previously found, contrary to what happened in [Bernstein et al., 2013].

Univariate Coppersmith

Let $p = a + r$ be the prime factor of n, where a is a known 512-bit integer and r is a small integer error to account for a sequence of bit errors among the least significant bits of p.

In the Coppersmith method, and following the [Howgrave-Graham, 2001] approach, the polynomial $f(x) = a+x$ can be written. It will be calculated a root, r of f modulo a large divisor of n. Let X be the bound on the size of the root. This divisor is going to be approximately $n^{1/2} \approx p$. A new polynomial, $g(x)$, where $g(x) = 0$, is going to be constructed over the integers using the lattice basis reduction. Thus, by factoring g, one can discover r.

Let L be the lattice generated by the rows of the basis matrix

$$\begin{bmatrix} X^2 & Xa & 0 \\ 0 & X & a \\ 0 & 0 & n \end{bmatrix}$$

corresponding to the coefficients of the polynomials $Xxf(Xx), f(Xx), n$. Any vector in L can be written as an integer combination of basis vectors. If one divides those basis vectors by the appropriate power of X, one will have the coefficients of a polynomial, $g(x)$. Thus, $g(x)$ is an integer combination of f and n and thus it is divisible by p by construction. A prime p is found by this method only if the function g found satisfies the condition $g(r_i) \equiv 0 \,(\mathrm{mod}\ p)$ not only modulo p but all over the integers. The latter is ensured if the coefficients of g are sufficiently small. Finding the sufficiently small coefficients of g to ensure the previous condition is the same as finding a short vector in L.

The application of the LLL basis reduction algorithm, that can be found in [Lenstra et al., 1982], enables to find the short vector.

In the last step of the algorithm, the shortest vector in the reduced basis is regarded as the coefficients of a polynomial $g(Xx)$ and the roots r_i of $g(x)$ are computed and the condition $a + r_i$ divides n is checked. If so, n has been factored.

The length of the shortest vector, v is

$$|v| \leq 2^{(dim(L)-1)/4}(det(L))^{1/dim(L)}$$

which will be smaller than p for the attack to succeed.

Bivariate Coppersmith

This attack is an extension of the previous attack. It intends to factor keys with unpredictable bits amongst the middle or from the most significant bits of one of the factors, without addressing to brute-force of the bottom bits.

Let $p = a + 2^t s + r$ be a prime that factors n, where a is a 512-bit integer with a predictable bit pattern, t is a bit offset where a sequence of bit errors s deviating from the predictable pattern in a occurred during the key generation, and r is an error at the least significant bits.

Let's consider the equation $f(x, y) = a + 2^t x + y$. The lattice basis reduction is going to be used to find new polynomials $Q_i(x, y)$. If $f(s, r)$ vanishes modulo a large unknown divisor p of n and s and r are substantially small, then $Q_i(s, r) = 0$ over the integers. $Q_i(x, y)$ must hold this property. In order to do that, the appropriate zeros of Q_i should be found. The most common method to achieve that is to take multiple distinct polynomials Q_i and expect that their common solution set is not too large.

Almost all the applications of multivariate Coppersmith methods demand a heuristic assumption that the attacker can obtain two or various algebraically independent polynomial equations determined by the short vectors in a LLL-reduced lattice. This requirement gives authorization for the attacker to compute a finite set of common solutions.

Many cryptanalytic applications use these kinds of bivariate Coppersmith attack. One of those applications is the Boneh and Durfee's attack against RSA private key $d < n^{0.29}$ in

[Boneh and Durfee, 1999]. The approach used in [Bernstein et al., 2013] is identical to the approach described by Herrmann and May for factoring RSA modulus with some known bits, in [Herrmann and May, 2008].

Chapter 6

Randomness

It may be taken for granted that any attempt at defining disorder in a formal way will lead to a contradiction. This does not mean that the notion of disorder is contradictory. It is so, however, as soon as I try to formalize it.

Hans Freudenthal

Randomness is often defined as the outcome of an experiment, where, no matter how many times the experiment is done, the next outcome is unpredictable. Nonetheless this definition, the real question is: how can something be defined as random?

For example, in the area of artificial intelligence (AI), there is a proposed test to decide if a computer program possesses artificial intelligence or not. This test consists on the following set up, there are two rooms, one has a single person in front of a computer terminal that allows he/her to ask questions to another room. On this other room there are 2 responders, a computer running AI-software and a human being. The task is for the person on the first room to decide, by asking questions and making conversation, which one of the two is responding to the conversation. If judgements are correct as often as incorrect, the AI-software is said to pass the test.

Is there some similar test that decides if a source is random or not? One way to do it is to measure the randomness in the studying source and compare the results with true randomness. This measurement and comparison can be done by computer testing programs. If the results are indistinguishable, the source is said to pass the test and it is considered to be a random source. Otherwise, the source is not considered to be random. The only problem is that it seems very hard to do such testing, using "all" possible testing software. However, it has been shown that the testing process can be restrict to a single test (or class of tests) that covers all aspects.

Generating random numbers is critical to the security of cryptographic systems. Nevertheless, it is also very difficult to accomplish. Non-deterministic behaviour is considered to be a fault in almost every component of a computer but it is a vital component of a random numbers generator. Several national and international standards for random numbers generation specify the correct behaviour one wants to achieve for this kind of systems.

6.1 Random and Pseudorandom Numbers

The random and pseudorandom numbers are fundamental to many cryptographic applications. In fact, in almost all cryptosystems the keys used must be generated in a specific random way and, for example, many cryptographic protocols like authentication and digital signatures protocols require random or pseudorandom inputs at various points.

A random bit sequence is the result of a random choice between the numbers 0 or 1 for each bit. The probability of choosing the value 0 or the value 1 must be exactly 1/2. Also, the choices of the value must be independent: the choice of a certain bit must not affect the choice of the following one. This unbiased choice of bits is considered to be the perfect random stream generator, since the possible values for each bit are randomly distributed, following a uniform distribution.

Pseudorandomness refers to a distribution on strings. When a string of length l is said to follow a pseudorandom distribution $\mathcal{D}$ this means that $\mathcal{D}$ is indistinguishable from the uniform distribution over strings of length l, which means that it is infeasible for any polynomial-time algorithm to tell whether it is a string sampled according to $\mathcal{D}$ or an l-bit string chosen uniformly at random. Pseudorandomness is a computational relaxation of true randomness.

The distributions $\mathcal{D}$ are defined by choosing a random seed $s \leftarrow \{0,1\}^n$ uniformly at random and then outputting $G(s) \in \{0,1\}^l$. The distribution $\mathcal{D}$ define the string $y \in \{0,1\}^l$ as output with the exact probability

$$\frac{|\{s \in \{0,1\}^n\} \quad | \quad G(s) = y\}|}{2^n}$$

which will, in general, not be the uniform distribution ([Katz and Lindell, 2007]).

There are two different strategies for generating random bits: the bits can be produced non-deterministically or they can be computed deterministically by using a certain algorithm. The generators that use the first strategy are known as random bits generators (RBGs) which originate the Random Numbers Generators (RNGs), and the generators that use the second strategy are known as deterministic random bits generators (DRBGs) which, in their turn, originate the Pseudorandom Numbers Generators (PRNGs). The DRNG produce a sequence of bits by instantiate the used algorithm with an initial value that is determined by the seed set from the input entropy. Those bits are said to be pseudorandom bits instead of random bits because of the deterministic nature of their generation process. The seed used to instantiate the DRBG must contain sufficient entropy to provide a randomness assurance and it must be kept secret. If these properties are ensured and if the algorithm is well designed, the bits outputted by the DRBG will be unpredictable.

Both RNGs and PRNGs produce a stream of zeros and ones that may be divided into substreams or blocks of random numbers.

Another important feature of a bitstream generation is its unpredictability. This means all elements of the sequence must be generated independently, i.e., for all position i it should not be possible to predict the value of the position $i + 1$, regardless of how many of the elements have already been produced.

The unpredictability of the random numbers can be divided into two notions of unpredictability: the forward unpredictability and the backward unpredictability.

The forward unpredictability is the property that, if the seed is unknown, guarantees the unpredictability of he next output number, once known all the previous outputted numbers.

The backward unpredictability is the property that guarantees the incapacity of knowing the seed used to produce a certain random number from the knowledge of the already generated values.

The generation of a truly random sequence is impracticable for cryptographic purposes. However, the truly random number generator outputs serve as a benchmark for the evaluation of random and pseudorandom number generators.

6.2 Random Number Generators (RNGs)

In order to produce randomness, a RNG uses an entropy source as a non-deterministic source, along with a processing function, also known as a distillation process that is needed to overcome any weakness in the entropy source that may result in non-random numbers.

The outputs of a RNG can be used in two different ways: they can be used directly as random numbers or they can be introduced in a PRNG for latter processing. For the random numbers generated by a RNG to be used directly as random numbers, they need to satisfy strict randomness criteria measured by statistical tests in order to determine whether the physical sources of the RNG inputs appear to be random. For cryptographic purposes, the output of a RNG needs to be unpredictable but some physical sources are quite predictable, which can be a problem. One possible solution to this problem is to use distinct entropy sources combined to generate the inputs of a RNG. However, this is not the the ideal solution since the resulting outputs from the RNG may still be deficient when evaluated by statistical tests.

A disadvantage of using a RNG is the time consuming. Thus, the production of large quantities of high-quality random numbers is preferable to be done with a PRNG.

6.3 Pseudorandom Number Generators (PRNGs)

A PRNG is a deterministic algorithm that uses one or more truly random inputs, called seeds (which have short length), and generates multiple pseudoramdom numbers by stretching the seed into a long string. Concluding, a pseudorandom generator uses a small amount of true randomness in order to generate a large amount of pseudorandomness.

Definition 4. *[Katz and Lindell, 2007] Let $l(\bullet)$ be a polynomial and let G be a deterministic polynomial-time algorithm such that upon any input $s \in \{0,1\}^n$, G outputs a string of length $l(n)$. G is a pseudorandom generator if the following two conditions hold:*

1. *Expansion: For every n, $l(n) > n$.*
2. *Pseudorandomness: For all probabilistic polynomial-time distinguishers, D, there exists a negligible function, $negl$, such that*

$$Pr[D(r) = 1] - Pr[D(G(s)) = 1] \leq negl(n)$$

where r is chosen uniformly at random from $\{0,1\}^{l(n)}$, the seed s is chosen uniformly at random from $\{0,1\}^n$, and the probabilities are taken over the random coins used by D and the choice of r and s.

The function $l(\bullet)$ is called the expansion factor of G.

If G is a PRNG it is guaranteed that there do not exist any polynomial-time procedures that succeed in distinguishing random and pseudorandom strings, which means that pseudorandom strings are just as good as truly random ones, as long as the seed is kept secret and only polynomial-time observers are considered.

There must not be any correlation between the seed, s, and the generated numbers. In most of the cases, the pseudorandom generating algorithms are publicly available. Thus, the choice of the seed must be carefully handled and it must be kept secret because once the seed is known, all the values produced by the PRNG are known. Thereby it is not possible to ensure the forward unpredictability property. The PRNG must ensure either the forward as the backward unpredictability and, in addition, the seed itself must be unpredictable. In addition, s must be long enough in order to be impossible to efficiently do a brute-force attack.

By default, a PRNG should obtain its seeds from the outputs of an RNG for the seeds to be random and unpredictable. Thus, a PRNG requires a RNG.

In fact, all the randomness relies on the seed generation, since the outputs of a PRNG are deterministic functions with the seed as an input. The term *pseudorandom* come out due to the deterministic nature of the PRNG. Each element of a pseudorandom sequence is reproducible from its seed. So in the requirement of reproduction or validation of the pseudorandom sequence the seed needs to be known, which leads to the conclusion that the seed has to be saved in those cases.

As said before, a pseudorandom string is a string that looks like a uniformly distributed string, as long as the algorithm that is testing it runs in polynomial time. In a properly constructed pseudorandom sequence each value in the sequence is produced from the previous value via transformations that appear to introduce additional randomness. These transformations provide better statistical properties for the outputs of a PRNG and in addition, the outputs of a PRNG can be produced faster than the outputs of a RNG.

When a seed has sufficient entropy and when it is possibly de-skewed and mixed (see Sections 6.6.2 and 6.6.3), it is possible to algorithmically extend that seed to produce a large number of cryptographically-strong random quantities. Such algorithms are platform independent and can operate in the same way in any computer. For the algorithms to be secure, their input and internal work must be protected from the observation of an attacker.

Usually it is better to use a pseudorandom string rather than a truly random string since a long pseudorandom string can be generated from a relatively short random seed (or key) while the generation of a random string is too time consuming.

So, is there any entity that satisfies Definition 4? There is no way to unequivocally prove the existence of pseudorandom generators. However the belief that pseudorandom generators

exist is based on the fact that they can be constructed under the assumption that one-way functions exist ([Katz and Lindell, 2007]). In fact there are long-studied problems that do not have any known efficient algorithm to solve them and that are widely assumed to be unsolvable in polynomial-time, like the discrete logarithm problem (Definition 3) and the problem of integer factorization (Chapter 4). So, one-way functions, and hence pseudorandom generators, can be constructed under the assumption that these problems are truly "hard" to solve.

6.3.1 Failed Algorithms

Pseudorandom number generation algorithms must be designed so that they are not susceptible to failures to occur. There have been many ideas that might seem reasonable to ensure the security of pseudorandom numbers generation but led precisely to the opposite situation.

The Fallacy of Complex Manipulation

One approach that may give a misleading appearance of unpredictability is to take a very complex algorithm to calculate a cryptographic key using as input a seed provided by some limited data such as the value of the computer system clock.. Adversaries who roughly knew when the generator has started would have a relatively small number of seed values to test, as they would know likely values of the system clock.

A very strong or complex manipulation of data may not be a solution if the adversary can learn the manipulation that is been done and if there is not enough entropy in the starting seed value. The attackers can usually use the limited number of results generated from a limited number of seed values to defeat security.

A very complex pseudorandom number generation algorithm does not necessarily produce strong random numbers and it can be a serious strategic error when there is no theory behind or algorithm analysis.

If there exists only a limited range of seeds, a complex manipulation do not help and, in fact, a blindly-chosen complex manipulation can destroy the entropy in a good seed.

The Fallacy of Selection from a Large Database

Another approach that can give a misleading appearance of unpredictability is to randomly select a bits sequence from a database and to assume that its strength is related to the total number of bits in the database.

For example, this argument is valid when sequences are selected from the data on a publicly available CD/DVD recording or from any other large public database. If the adversary has access to the same database he/she is able to break the system. However, if the adversary has no access to the database, this type of selection may be of help.

The Fallacy of Traditional Pseudorandom Sequences

Traditional sources of "pseudorandom" numbers typically start with a seed quantity and use simple numeric or logical operations to produce a sequence of values.

A typical pseudorandom number generation technique is the linear congruence pseudorandom number generator that uses modular arithmetic, where the value numbered as $N + 1$, V_{N+1}, is calculated from the value numbered as N, V_N, by

$$V_{N+1} = (V_N \times a + b) \,(\text{mod } c)$$

with a, b, c integers.

This technique has a strong relationship to linear shift register pseudorandom number generators, which are well understood cryptographically. In such generators, bits are introduced at one end of a shift register as the Exclusive Or of bits from selected fixed taps into the register.

The quality of traditional pseudorandom number generator algorithms is measured by statistical tests on such sequences. Carefully-chosen a, b, c values and initial V or carefully-chosen placement of the shift register tap can produce excellent statistics.

Such sequences are not recommended to use in security applications since they are fully predictable if the initial state is known. In fact, given V_N it is possible to determine V_{N+1}. It has been shown that with these techniques, even if only one bit of the pseudo-random values are released, the seed can be determined from short sequences.

6.3.2 Cryptographically Strong Sequences

In cases where a series of random quantities must be generated, an adversary may learn some values in the sequence but he should not be able to predict other values from the ones that he knows.

The correct technique is to start with a strong random seed, take cryptographically strong steps from that seed ([Ferguson and Schneier, 2003], [Schneier, 1996]), and not reveal the complete state of the generator in the sequence elements.

Output Feedback and Counter Mode Encryption Sequences

One way to produce a strong sequence is to take a seed value and hash the produced quantities produced by concatenating the seed, for example, with successive integers and then mask the obtained values in order to limit the amount of generator state available to the adversary.

In counter (CTR) mode encryption it is used an encryption algorithm with a random key and seed value to encrypt successive integers. Alternatively, one can feedback all of the output value from encryption into the value to be encrypted for the next iteration. This is a particular example of output feedback mode (OFB) ([mod, 1980b], [mod, 1980a]).

To predict values of a sequence from other values when the sequence was generated by these techniques is equivalent to break the cryptosystem or to invert the "non-invertible" hashing with only partial information available. The less information revealed in each iteration, the harder it will be for an adversary to predict the sequence. Thus, it is best to use only one bit from each value. It has been shown that in some cases this makes a system impossible to break even when the cryptographic system is invertible and it could be broken if all of each generated value were revealed.

The Blum Blum Shub Sequence Generator

The Blum Blum Shub generator (BBS) (discussed with more detail in Section 6.7) is a simple generator based on quadratic residues and it is considered to be a good generator ([Katz and Lindell, 2007]).

It has only one disadvantage: it is computationally intensive compared to the traditional techniques seen before. However it is not a major drawback once it can be used for moderately, infrequent purposes, such as generating key sessions.

Entropy Pool Techniques

Many modern pseudorandom number sources use as technique maintaining a "pool" of bits and providing operations for strongly mixing the input with some randomness into the pool and extracting pseudorandom bits from that pool.

The bits destined to be fed into the pool can come from any of the various hardware, environmental, or user input sources discussed above. It is also common to save the state of the pool on the system shutdown and to restore it on the re-starting, when stable storage is available.

Care must be taken: enough entropy may be added to the pool to support particular output uses desired.

6.4 Theoretical Constructions of Pseudorandom Objects

Although the constructions of pseudoramdom generators are not likely to be used in practice ([Katz and Lindell, 2007]),the existence of a strong theoretical component allows a more precise analysis of the schemes that are actually used in real situations. In fact, the theoretical study of this matter allows for a better understanding of how one could get security and what are the minimal assumptions necessary to do so.

Some assumptions are necessary because an unconditional proof of the existence of pseudorandom generators or pseudorandom functions would involve breakthroughs in complexity theory ([Katz and Lindell, 2007]). Thus, with the current understanding of complexity, a pseudorandom function cannot be constructed from scratch and being prove mathematically that it is indeed pseudorandom: the goal is to try to base the pseudorandom constructions on the "minimal assumption" possible. Formally, a minimal assumption is one that is both necessary and sufficient for achieving constructions of pseudorandom generators and pseudorandom functions. The minimal assumption required is the existence of one-way functions. As a matter of fact, one-way functions are a minimal assumption for cryptography in general, and not just for obtaining pseudorandom generators and functions.

Definition 5. [One-way Function]*[Katz and Lindell, 2007]*

A function $f : \{0,1\}^* \to \{0,1\}^*$ *is called one-way if it holds the two conditions:*

- *Easy to compute: there exists a polynomial-time algorithm* M_f *such that for any input* $x \in \{0,1\}^*$ *outputs* $f(x)$.
- *Hard to invert: for every probabilistic polynomial-time inverting algorithm A, there exists a negligible function negl such that*

$$Pr[A(f(x)) \in f^{-1}(f(x))] \leq negl(n)$$

where the probability is taken over the uniform choice of $x \in \{0,1\}^n$ and the random choice of A.

It is only guaranteed that a one-way function is hard to invert when the input is uniformly distributed and further, if x holds a long enough value.

Thus, f hides information about x, since the value of x is unknown to any polynomial-time inverting algorithm, even when $y = f(x)$ is given. However, any one-way function can be inverted if enough time is given. Specifically, given a value y, it is always possible to simply try all values of x by increasing its length until one of those values satisfy the condition $f(x) = y$. This approach always succeeds but it runs in exponential time, which makes the existence of one-way functions inherently an assumption about computational complexity and computational hardness. Thus, a one-way function is a problem that can be solved but not in a feasible amount of time.

It is important to clarify that a function that is not a one-way function is not necessarily easy to invert. The converse of Definition 5 is that there exists a probabilistic polynomial-time algorithm A and a non-negligible function ϵ such that A inverts $f(x)$ for $x \in \{0,1\}^n$ with probability at least $\epsilon(n)$.

Definition 6. [One-way Permutation]*[Katz and Lindell, 2007]*

Let f be a function with domain $\{0,1\}^*$ and f_n a function that restricts f to the domain $\{0,1\}^n$, i.e., for every $x \in \{0,1\}^n$, $f_n(x) = f(x)$. Then, a one-way function f is called a one-way permutation if for every n, the function f_n is a bijection onto $\{0,1\}^n$.

Due to the fact that a one-way permutation is a bijection, for any value y there exists only one preimage x and even though y fully determines x, it is still hard to find x in polynomial-time.

Definition 7. *A tuple $\prod = (Gen, Samp, f)$ of probabilistic, polynomial time algorithms is a family of functions if:*

- *The generation algorithm Gen, on input 1^n, outputs parameters I with $|I| \geq n$. Each value of I defines the sets D_I and R_I that constitute the domain and range, respectively, of the function f.*
- *The sampling algorithm $Samp$, on input I, outputs a uniformly distributed element of D_I (except possibly with probability negligible in $|I| \geq n$).*
- *The deterministic evaluation algorithm f, on input I and $x \in D_I$, outputs an element $y \in R_I$. We write this as $y := f_I(x)$.*

$\prod$ is a family of permutations if for each value of I outputed by $Gen(1^n)$, it holds that $D_I = R_I$ and the function $f_I : D_I \to D_I$ is a bijection.

A family of functions can be created, but in order to achieve the properties of an one-way function or permutation, one has to define one-wayness for the family created. A way of doing this is to run an inversion test, $Invert_{A,\prod}(n)$, consisting of the following:

- $Gen(1^n)$ is run to obtain I, and then $Samp(I)$ is run to obtain a random $x \leftarrow D_I$. Finally, $y = f_I(x)$ is computed.
- I and y are given as an input for A, and outputs x'.
- The output of the experiment is defined to be 1 if $f_I(x') = y$, and 0 otherwise.

A function is one-way if success occurs in the above experiment with at most negligible probability. This idea can be formalized:

Definition 8. *A family of functions/permutations $prod = (Gen, Samp, f)$ is called one-way if for all probabilistic polynomial-time algorithms A there exists a negligible function negl such that*

$$Pr[Invert_{A,\prod}(n) = 1] \leq negl(n).$$

As said before, there is no way, at least nowadays, to prove one-way functions exist. So, their existence is a conjecture based on some very natural computational problems that have received many attention, and do not have yet an inversion algorithm that solves the problem in polynomial-time. There are many candidates to be a problem with a one-way function. Perhaps the most famous of these problems is the integer factorization (with the restriction on the length of the number that one tries to factor: the numbers that will be multiplied must be large enough) (Chapter 4) but there exists another ones like the discrete logarithm problem, the subset-sum problem (one can read more about them in [Menezes et al., 1996], Chapter 3).

Hereupon, one may have the impression that x is completely unknown, even given $f(x)$. However, this is not the case. Indeed, a one-way function f may yield a lot of information about its input, and yet still be hard to invert. The information truly hidden by f is called a hard-core predicate.

Definition 9. [Hard-Core Predicate]

A polynomial-time computable predicate $hc : \{0,1\}^* \rightarrow \{0,1\}$ is called a hard-core (hc) of a function f if for every probabilistic polynomial-time algorithm A, there exists a negligible function negl such that

$$\Pr_{x \leftarrow \{0,1\}^n}[A(f(x)) = hc(x)] \leq \frac{1}{2} + negl(n)$$

where the probability is taken over the uniform choice of $x \in \{0,1\}^n$ and the random choice of A.

So, a hard-core predicate hc of a function f is a function that outputs a single bit with the following property: if f is one-way, then upon input $f(x)$ it is infeasible to correctly guess $hc(x)$ with any non-negligible advantage above $1/2$ (it is always possible to compute correctly $hc(x)$ with probability $1/2$ by just randomly guessing it).

A hard-core predicate candidate has to be chosen carefully, because hc can be part of the function output. Thus, the predicate $hc(x)$ is not always a hard-core predicate and it can be shown ([Katz and Lindell, 2007]) that for every given predicate hc, there exists a one-way function for which hc is not a hard-core predicate of the function f.

Intuitively, a $1-1$ function f has a hard-core predicate only if it is one-way because, given the value $f(x)$, a bijective function fully determines x. And if the function f is bijective and is not one-way, it can be inverted, revealing the unique preimage x and thus, $hc(x)$ can be computed from $f(x)$ with some non-negligible advantage. Hence, if a $1-1$ function is not one-way it cannot have a hard-core predicate. The difficulty of guessing the hard-core of a bijective function is due to its computational difficulty in calculating it. The hard-core predicates are used in order to construct pseudorandom generators.

Following there will be described the steps needed in the construction of pseudorandom generators and pseudorandom functions from one-way functions.

The function $g(x,r) = (f(x), r)$ is going to be considered for each one-way function f. The first step is to show that for every one-way function f, there exists a hard-core predicate for the function $g(x,r)$, where $|x| = |r|$. To do that, there exists the following theorem:

Theorem 8. *Let f be a one-way function and define $g(x,r) = (f(x), r)$. Then, the function $gl(x,r) = \oplus_{i=1}^{n} x_i \times r_i$, where $x = x_1, \ldots, x_n$ and $r = r_1, \ldots, r_n$, is a hard-core predicate of g.*

This theorem was proved by Goldreich and Levin and for more details of the demonstration one can see [Goldreich and Levin, 1989]. If f is a bijection or a permutation, g is also a bijection or a permutation, respectively, and if f is one-way, then g is also one-way. Due to the fact that r is uniformly distributed and it can be seen as the factor that selects a subset for x, the function $gl(x,r)$ outputs a bit that consists of the exclusive-or of a random subset of the bits $x_1, \ldots, x_n$ and it states whether $f(x)$ hides the exclusive-or of a random subset of the bits of x or not.

For notation simplification, everytime $f(x)$ is written, it is considered to be a one-way function of the form $g(x,r) = (f(x), r)$.

The next step in the construction is to show how hard-core predicates can be used to obtain pseudorandom generators.

Theorem 9. *Let f be a one-way permutation and let hc be a hardcore predicate of f. Then, $G(s) = (f(s), hc(s))$ constitutes a pseudorandom generator with expansion factor $l(n) = n+1$.*

If f is a permutation, for a uniformly distributed x it holds that $f(x)$ is also uniformly distributed and if hc is a hard-core of f, then the bit $hc(x)$ looks random, even given $f(x)$ ([Katz and Lindell, 2007]), because a polynomial-time algorithm can guess the value $hc(x)$ given $f(x)$ with probability only negligibly greater than $1/2$ and this is equivalent to say that it looks random, or more formally, that it is pseudorandom.

It is very difficult to construct a pseudorandom generator that stretches the seed even by just a single bit and the goal is to obtain many pseudorandom bits and therefore have a pseudorandom generator with large expansion factors. Pseudorandom generators that stretch the seed by one bit can be used to construct pseudorandom generators with any polynomial expansion factor ([Katz and Lindell, 2007]).

Theorem 10. *Assume that there exist pseudorandom generators with expansion factor $l(n) = n+1$. Then, for every polynomial $p(\bullet)$, there exists a pseudorandom generator with expansion factor $l(n) = p(n)$.*

Thus, pseudorandom generators can be constructed from any one-way permutation.

The last step is to construct pseudorandom functions from pseudorandom generators:

Theorem 11. *Assume that there exist pseudorandom generators with expansion factor $l(n) = 2n$. Then, there exist pseudorandom functions and pseudorandom permutations.*

The following corollary results from the combination of the theorems 8 to 11.

Corollary 1. *Assuming the existence of one-way permutations, there exist pseudorandom generators with any polynomial expansion factor, pseudorandom functions and pseudorandom permutations.*

This results are possible to obtain from any one-way function.

So, given $f(s)$ for a random s, it is hard to guess the value of $hc(s)$ with probability non-negligibly higher than $1/2$, which causes $hc(s)$ to be a pseudorandom bit. Moreover, since f is a permutation, $f(s)$ is uniformly distributed. This facts make the string $(f(s), hc(s))$ to be pseudorandom and one can conclude that the algorithm $G(s) = (f(s), hc(s))$ constitutes a pseudorandom generator.

Theorem 12. *Let f be a one-way permutation, and let hc be a hardcore predicate of f. Then, the algorithm $G(s) = (f(s), hc(s))$ is a pseudorandom generator with $l(n) = n + 1$.*

Proof. The theorem is proved by reduction: it is shown that if there exists a distinguisher D that can distinguish $G(s)$ from a truly random string, this distinguisher can be used to construct an adversary A that guesses $hc(s)$ from $f(s)$ with probability that is nonnegligibly greater than $1/2$.

By contradiction, let's assume that there exists a probabilistic polynomial-time distinguisher D and a non-negligible function $\epsilon(n)$ such that

$$\left| \Pr_{s\in\{0,1\}^n}[D(f(s), hc(s)) = 1] - \Pr_{r\in\{0,1\}^{n+1}}[D(r) = 1] \right| \geq \epsilon(n)$$

ϵ is called *distinguishing gap*. D is said to distinguish $(f(s), hc(s))$ from a random r with probability $\epsilon(n)$. As a first step to construct an algorithm A that guesses $hc(s)$ from $f(s)$, the fact that D can distinguish $(f(s), hc(s))$ from $(f(s), hc(s))$, where $hc(s) = 1 - hc(s)$ must be demonstrated. In order to do this, one have to first note that

$$\Pr_{s\in\{0,1\}^n,\beta\in\{0,1\}}[D(f(s),\beta) = 1] = \frac{1}{2} \cdot Pr[D(f(s), hc(s)) = 1] + \frac{1}{2} \cdot Pr[D(f(s), \overline{hc}(s)) = 1]$$

because with probability $1/2$ the random bit β equals $\overline{hc}(s)$, and with probability $1/2$ it equals $hc(s)$. Thus,

$$\begin{aligned}
&|Pr[D(f(s), hc(s)) = 1] - Pr[D(f(s),\beta) = 1]| = \\
&= |Pr[D(f(s), hc(s)) = 1] - \frac{1}{2} \cdot Pr[D(f(s), hc(s)) = 1] - \frac{1}{2} \cdot Pr[D(f(s), \overline{hc}(s)) = 1]| \\
&= \frac{1}{2} \cdot |Pr[D(f(s), hc(s)) = 1] - Pr[D(f(s), \overline{hc}(s)) = 1]|
\end{aligned}$$

where, in all of the probabilities above, $s \leftarrow \{0,1\}^n$ and $\beta \leftarrow \{0,1\}$ are chosen uniformly at random. By the contradicting assumption, and noticing that $(f(s), \beta)$ is just a uniformly distributed string of length $n+1$,

$$\begin{aligned}
&\left| \Pr_{s\in\{0,1\}^n}[D(f(s), hc(s)) = 1] - \Pr_{s\in\{0,1\}^n}[D(f(s), \overline{hc}(s)) = 1] \right| = \\
&= 2 \cdot \left| \Pr_{s\in\{0,1\}^n}[D(f(s), hc(s)) = 1] - \Pr_{r\in\{0,1\}^{n+1}}[D(r) = 1] \right| \\
&\geq 2\epsilon(n)
\end{aligned}$$

Without loss of generality let's assume that $Pr[D(f(s), hc(s)) = 1] > Pr[D(f(s), \overline{hc}(s)) = 1]$. Let's use D to construct an algorithm A that, given $f(s)$, guesses $hc(s)$. Upon the input $y = f(s)$, for a random s, the algorithm A works as follows:

1. $\sigma \leftarrow \{0,1\}$ is uniformly chosen.
2. D is invoked upon (y, σ).
3. If D returns 1, then A outputs σ. Otherwise, A outputs $\overline{\sigma}$.

A should succeed because D outputs 1 when $\sigma = hc(s)$ with a $2\epsilon(n)$ probability more than it outputs 1 when $\sigma = \overline{hc}(s)$. Thus, if $\sigma = hc(s)$, A invokes D on the input $(f(s), hc(s))$ and outputs $hc(s)$ if and only if D outputs 1 upon the input $(f(s), hc(s))$. Likewise, if $\sigma \neq hc(s)$, A invokes D on the input $(f(s), \overline{hc}(s))$ and outputs $hc(s)$ if and only if D outputs 0 upon the input $(f(s), \overline{hc}(s))$. Continuing the analysis,

$$\begin{aligned}
&Pr[A(f(s)) = hc(s)] = \\
&= \frac{1}{2} \cdot Pr[D(f(s), hc(s)) = 1] + \frac{1}{2} \cdot (1 - Pr[D(f(s), \overline{hc}(s)) = 1]) \\
&= \frac{1}{2} + \frac{1}{2} \cdot Pr[D(f(s), hc(s)) = 1] - \frac{1}{2} \cdot Pr[D(f(s), \overline{hc}(s)) = 1] \\
&= \frac{1}{2} + \frac{1}{2} \cdot (Pr[D(f(s), hc(s)) = 1] - Pr[D(f(s), \overline{hc}(s)) = 1]) \\
&\geq \frac{1}{2} + \frac{1}{2} \cdot 2\epsilon(n) \\
&= \frac{1}{2} + \epsilon(n)
\end{aligned}$$

and thus A guesses $hc(s)$ with probability $1/2 + \epsilon(n)$. Since $\epsilon(n)$ is a nonnegligible function, this contradicts the assumption that hc is a hard-core predicate of f.

□

A crucial point in creating a pseudorandom generator is in the property that states that its construction allows expansion factor to be increase. It is shown that, in fact, any pseudorandom generator can have an arbitrary polynomial expansion factor.

Theorem 13. *If there exists a pseudorandom generator G_1 with expansion factor $l_1(n) = n+1$, then for any polynomial $p(n) > n$, there exists a pseudorandom generator G with expansion factor $l(n) = p(n)$.*

Proof. Given an initial seed s of length n, the generator G_1 can be used to obtain $n+1$ pseudorandom bits. One of the $n+1$ bits may be outputted, and the remaining n bits can be used once again as a seed for G_1. The reason for those n bits to be used as a seed is their pseudorandomness, and therefore they are essentially as good as a truly random seed. This procedure can be iteratively applied to output as many bits as desired.

The construction will be done as follows:

1. Let $s \in \{0,1\}^n$ be the seed, and denote $s_0 = s$.
2. For every $i = 1, \ldots, p(n)$, $(s_i, \sigma_i) = G_1(s_{i-1})$ is computed, where $\sigma_i \in \{0,1\}$ and $s_i \in \{0,1\}^n$.
3. $\sigma_1, \ldots, \sigma_{p(n)}$ is outputted.

Let's proceed proving that $G(s)$ is a pseudorandom string of length $p(n)$. The proof is done for the special and simple case of $p(n) = 2$. That is, the output of $G(s) = (\sigma_1, \sigma_2)$. This is not a pseudorandom generator, because the output length is shorter than the input length but, nevertheless, it is helpful for understanding the basis of the proof. For a full proof one can check [Katz and Lindell, 2007], Chapter 6.

The output (σ_1, σ_2) of $G(s)$ is obtained in two stages: firstly by computing $(s_1, \sigma_1) = G_1(s)$ and then by computing $(s_2, \sigma_2) = G_1(s_1)$. Let's consider an experiment with an algorithm G' that receives $n+1$ random bits as input, denoted by $\tilde{s} \in \{0,1\}^{n+1}$. Let $\tilde{s}^{[n]}$ be the first n bits of $\tilde{s}$ and let $\tilde{s}^{n+1}$ be the last bit of $\tilde{s}$. Then, G' works by computing $(s_2, \sigma_2) = G_1(\tilde{s}^{[n]})$ and setting $\sigma_1 = \tilde{s}^{n+1}$ (thus σ_1 is uniformly distributed). As with G, the algorithm G' outputs (σ_1, σ_2). For every probabilistic polynomial-time distinguisher D there exists a negligible function $negl$ such that

$$\left| \Pr_{s \in \{0,1\}^n}[D(G(s)) = 1] - \Pr_{\tilde{s} \in \{0,1\}^{n+1}}[D(G'(\tilde{s})) = 1] \right| \leq negl(n) \tag{6.1}$$

This must be the case because otherwise D can be used by an algorithm D' to distinguish $G_1(s)$ from random in the following way: given a string w of length $n+1$, the algorithm D' can compute $(s_2, \sigma_2) = G_1(w^{[n]})$ and define $\sigma_1 = w^{n+1}$ (exactly like G'). Then, D' invokes D on (σ_1, σ_2) and outputs whatever D outputs. There are some important observations:

1. If $w = r$ is truly random, then the pair (σ_1, σ_2) prepared by D' is identically distributed to $G'(\tilde{s})$. Thus,

$$\Pr_{r \in \{0,1\}^{n+1}}[D'(r) = 1] = \Pr_{\tilde{s} \in \{0,1\}^{n+1}}[D(G'(\tilde{s})) = 1] \tag{6.2}$$

2. If w is the output of $G_1(s)$ for a random $s \leftarrow \{0,1\}^n$, then the pair (σ_1, σ_2) prepared by D' is identically distributed to $G(s)$. In order to see this, note that in this case, σ_1 is the $(n+1)^{th}$ bit of $G_1(s)$ and σ_2 is the $(n+1)^{th}$ bit of $G_1(s_1)$, where $s_1 = G_1(s)^{[n]}$, exactly as in the construction of G. Therefore,

$$\Pr_{s\in\{0,1\}^n}[D'(G_1(s)) = 1] = \Pr_{s\in\{0,1\}^n}[D(G(s)) = 1] \tag{6.3}$$

From equations (6.2) and (6.3),

$$\left|\Pr_{s\in\{0,1\}^n}[D'(G_1(s)) = 1] - \Pr_{r\in\{0,1\}^{n+1}}[D'(r) = 1]\right| =$$
$$\left|\Pr_{s\in\{0,1\}^n}[D(G(s)) = 1] - \Pr_{\tilde{s}\in\{0,1\}^{n+1}}[D(G'(\tilde{s})) = 1]\right|$$

If equation (6.1) does not hold, this implies that D' distinguishes the generator $G_1(s)$ from random with non-negligible probability. This contradicts the assumption of G_1 pseudorandomness.

It is intended that for every probabilistic polynomial-time, D, there exists a negligible function $negl$ such that

$$\left|\Pr_{\tilde{s}\in\{0,1\}^{n+1}}[D(G'(\tilde{s})) = 1] - \Pr_{r\in\{0,1\}^2}[D(r) = 1]\right| \leq negl(n) \tag{6.4}$$

This proof is analogous to the proof done before. Specifically, if the equation (6.4) does not hold, D can be used by an algorithm D' to distinguish $G_1(s)$ from random as follows: given a string w of length $n+1$, the distinguisher D' sets σ_1 to be truly random and σ_2 to be the $(n+1)^{th}$ bit of w. As above, there are some important observations:

1. If $w = r$ is truly random, then the pair (σ_1, σ_2) prepared by D' is truly random. Thus,

$$\Pr_{w\in\{0,1\}^{n+1}}[D'(w) = 1] = \Pr_{r\in\{0,1\}^2}[D(r) = 1] \tag{6.5}$$

2. If w is the output of $G_1(s)$ for a random $s \leftarrow \{0,1\}^n$, then the pair (σ_1, σ_2) prepared by D' is identically distributed to $G'(\tilde{s})$. This follows because G' sets σ_2 to be the $(n+1)^{th}$ bit of the output of $G_1(\tilde{s})$ and σ_1 to be truly random. Therefore,

$$\Pr_{s\in\{0,1\}^n}[D'(G_1(s)) = 1] = \Pr_{\tilde{s}\in\{0,1\}^n}[D(G'(\tilde{s})) = 1] \tag{6.6}$$

$$\left| \Pr_{s\in\{0,1\}^n}[D'(G_1(s))=1] - \Pr_{w\in\{0,1\}^{n+1}}[D'(w)=1] \right| =$$
$$\left| \Pr_{\tilde{s}\in\{0,1\}^{n+1}}[D(G'(\tilde{s}))=1] - \Pr_{r\in\{0,1\}^2}[D(r))=1] \right|$$

From equations (6.5) and (6.6),

If equation (6.4) does not hold, this implies that D' distinguishes the generator $G_1(s)$ from random with non-negligible probability. This contradicts the assumption of the pseudorandomness of G_1.

Lastly, combining equations (6.1) and (6.4), one can conclude that for every probabilistic polynomial-time distinguisher there exists a negligible function *negl* such that

$$\left| \Pr_{s\in\{0,1\}^n}[D(G(s))=1] - \Pr_{r\in\{0,1\}^2}[D(r)=1] \right| \leq negl(n)$$

and so it is proven that G is pseudorandom. However, as mentioned, it is not a truly generator because it only outputs 2 bits.

□

By combining the Theorem 12 construction, that states that $G(s) = (f(s), hc(s))$ is a pseudorandom generator, together with the proof of Theorem 13 (maybe with some efficiency improvement), it is obtained that for every polynomial p,

$$G_l(s) = (f^{p(n)}(s), hc(s), hc(f(s)), \ldots, hc(f^{p(n)-n}(s)))$$

is a pseudorandom generator with expansion factor $p(n)$, assuming that f is a one-way permutation and hc is its hard-core predicate.

6.5 Testing Randomness

How can we access the level of randomness in the generated numbers?

Randomness is a probabilistic property. Thus, various statistical tests can be applied to a sequence to attempt to compare and evaluate that sequence to a truly random sequence since the likely outcome of statistical tests, when applied to a truly random sequence, is *à priori* known and can be described in probabilistic terms. All statistical tests check for the presence or absence of a pattern in the bits sequences evaluated. If a pattern is detected that would indicate that the sequence is nonrandom. There are an infinite number of statistical tests but no finite set of tests is deemed complete and the results of those tests must be interpreted with some care and caution to avoid incorrect conclusions about a specific generator.

In this project, the statistical tests that will be reviewed are the statistical tests recommended by the National Institute of Standards and Technology (NIST) to check if a given number passes or fails the hypothesis of being random. The tests that will be used have as reference the package of 15 tests provided by NIST named "A Statistical Test Suit for Random and Pseudorandom Number Generators for Cryptography Applications" ([Rukhin et al., 2010]). Since NIST is

responsible for specifying safety standards, and since there is a package of tests created by NIST that aim to test the patterns of randomness (standards set by NIST itself) of a given sequence of bits, the choice of using this statistical test suit seemed the most obvious choice. All tests in the test suit attempt to test the existence of any type of non-randomness in the tested sequences. Most of the existing tests in the NIST test suit follow a standard normal distribution and a chi-square distribution (χ^2). The standard normal distribution is used to compare the value obtained by the statistical test applied to the data with the expected value in case of the existence of really randomness. The standardization of a normally distributed random variable is done in order to always have a standard normal distribution, which enables an easily determination of the associated probability with a range of values for that variable by using a standardized distribution table. From now on, z represents the standardized random variable and, thus, a normally distributed random variable can be standardized using the formula $z = \frac{x - \mu}{\sigma}$, where μ and σ^2 are the mean and the variance values, respectively, of the statistic test and x is the the value that is being standardized. The χ^2 distribution is used to compare the quality of the frequencies observed by the data adjustment with the frequencies corresponding to the hypothesized distribution. The test statistic is of the form $\chi^2 = \sum \frac{(o_i - e_i)^2}{e_i}$, where o_i and e_i are the observed and expected frequencies of occurrence of the measure, respectively.

For many of the NIST statistical tests presented in NIST statistical test suit, the assumption that the size of the sequence length, n, is large (of the order 10^3 to 10^7) has been made.

6.5.1 How a Statistical Test Works

A statistical test is formulated to test a specific null hypothesis (H_0) that is always associated with an alternative hypothesis (H_a). In order to test randomness, the null hypothesis is that the sequence being tested is random and the alternative hypothesis is that the sequence is not random. The result of the applied statistical test is the acceptance or rejection of the null hypothesis. If the conclusion of the applied test is to accept the null hypothesis it means that the generator is producing random values, based on the tested data. Otherwise, if the conclusion is to reject the null hypothesis, this means that the generator is not producing random values.

For each test, a relevant randomness statistic must be chosen and used to determine the acceptance or rejection of the null hypothesis. Under an assumption of randomness, the statistic chosen has a distribution of possible values and from this distribution, a critical value is determined (typically 99%). During a test, a test statistic value is computed on the data and then, this value is compared to the critical value. If it is greater than the critical value, the null hypothesis for randomness is rejected. Otherwise, the null hypothesis is not rejected, i.e., the null hypothesis is accepted. If the randomness assumption is, in fact, true for the data evaluated by the statistical test, the value of the calculated result will have a very low probability of exceeding the critical value, i.e. the null hypothesis has a low probability of being rejected. From a statistical hypothesis testing point of view, the low probability event should not naturally occur and, therefore, if the calculated statistic test value exceeds the critical value, i.e., if the low probability event occurs, the conclusion is that the original assumption of randomness is suspect or faulty.

Concluding, statistical hypothesis testing may give two possible outcomes: accept H_0 (concluding the data is random) or reject H_0 (concluding the data is non-random).

However, a statistical test can give an outcome that does not conclude the real information about the data.

	H_0 **is true**	H_0 **is false**
Accept H_0	Right conclusion	Type II error
Reject H_0	Type I error	Right conclusion

Table 6.1: True status of the data available for analysis and the conclusion arrived by the usage of the testing procedure. An important observation is that the status of the data available for analysis is unknown in almost all the cases, in fact, that is why the statistical tests are made.

Table 6.1 illustrates the possible errors to occur. There are basically two types of possible errors: when H_0 is true but the statistical test outcomes its rejection (Type I error) and when H_a is true but the statistical test outcomes the acceptance of H_0 instead of its rejection (Type II error).

In this study case, where the null hypothesis is the data to be random, the correct conclusions are: accept H_0 when the tested data is really random, and reject H_0 when the tested data is non-random.

The probability of a Type I error to occur is called the test significance level, denoted by α, and it can be set prior to a test. Thus, α indicates the probability for the test to indicate that the sequence is not random when it really is random. In cryptography, the common value of α is chosen in $[0.001, 0.01]$ range.

The probability of a Type II error to occur is denoted as β. Therefore, β denotes the probability that the test will indicate that the sequence is random when it is not. This can occur, for example, when a "bad" generator produces a sequence that appears to have random properties. Unlike α, β is not a fixed value. In fact, β can take many different values because there are an infinite number of ways that a data stream can be non-random, and each one of those ways yields a different probability for the sequence to appear to have random properties. The calculation of β is more difficult than the calculation of α because of the many possible types of non-randomness. However, the probabilities α and β are related to each other and to the size, n, of the tested sequence in such a way that if two of them are specified, the third value is automatically determined (usually the selected parameters to be specified are n and α, and then a critical point for a given statistic is selected and the smallest β will be produced).

One of the primary goals of the statistical tests is to minimize the probability of a Type II error to occur.

Each test is based on a calculation of a test statistic value. If the test statistic value is S and the critical value is t, then:

- Type I error probability:

$$P(S > t \mid\mid H_0 \text{ is true}) = P(\text{reject } H_0 \mid H_0 \text{ is true})$$

- Type II error probability:

$$P(S \leq t \mid\mid H_0 \text{ is false}) = P(\text{accept } H_0 \mid H_0 \text{ is false})$$

The test statistic is used to calculate a *P-value*. A *P-value* is the probability of a perfect random number generator to produce a sequence less random than the sequence that was tested. Thus, the *P-value* summarizes the strength of the evidence against the null hypothesis. If a *P-value* for a test is determined to be equal to 1, the sequence appears to have perfect randomness and, thus, if a *P-value* is equal to 0, the sequence appears to be completely non-random. If *P-value* $\geq \alpha$, the null hypothesis is accepted. If *P-value* $< \alpha$, the null hypothesis is rejected.

An $\alpha = 0.01$ indicates that one would expect one sequence in 100 sequences to be rejected. For a *P-value* ≥ 0.01, a sequence would be considered to be random with a confidence level of 99.9%. For a *P-value* < 0.01, a sequence would be considered to be non-random with a confidence level of 99.9%.

6.5.2 Frequency Test

This test analyzes the proportion of zeros and ones existing in the data to be tested. In a truly random sequence, the number of zeros and ones should be approximately the same and thus the fraction of zeros and ones in the sequence should be approximately 1/2. The tests that will be described below depend on if the sequence passes this test or not. The reference distribution for the Frequency Test is the half-normal distribution (a one-sided variant of the normal distribution).

Let $\epsilon = \epsilon_1, \epsilon_2, \ldots, \epsilon_n$ be a sequence of bits, of size n, that is going to be tested. The bit sequence is produced by a RNG or by a PRNG, where each ϵ_i takes the value of the bit in position i.

Let $S_n = X_1 + X_2 + \cdots + X_n$, where $X_i = 2\epsilon_i - 1$, i.e., all zeros and ones of ϵ will be converted in -1 and 1 and then they will be added together. In case of a real random sequence, the number of 1s will be approximately equal to the number of -1s and thus the result of S_n will be near zero. If the number of ones and zeros presented in the sequence varies significantly, the result of S_n will be very far from zero. A positive large value of S_n is an indicative that the sequence has a lot of ones, while a large negative value of S_n indicates that the sequence contains many occurrences of zeros.

After compute $s_{obs} = \frac{|S_n|}{\sqrt{n}}$, the *P-value* $= erfc\left(\frac{s_{obs}}{\sqrt{2}}\right)$ is computed, where $erfc$ is the complementary error function, defined as

$$erfc(z) = \frac{2}{\sqrt{\pi}} \int_z^{\infty} e^{-u^2} du \tag{6.7}$$

If *P-value* < 0.01 (at 1% significance level), it can be concluded that the sequence is non-random. Otherwise it can be concluded that the sequence is random.

In fact, a small *P-value* means that $|S_n|$ or $|S_{obs}|$ have large values and, as seen before, this means that too many ones or too many zeros in ϵ.

In the NIST Statistic Test Suit ([Rukhin et al., 2010]) the recommendation is for each sequence to be tested to have a minimum of 100 bits.

6.5.3 Frequency Test Within a Block

This test analyzes the proportion of zeros and ones in a block of size M of the tested data. In a truly random M-bits block, the number of existing ones in the sequence is approximately $M/2$. It should be notice that if $M = 1$, this test is the Frequency Test (Section 6.5.2). The Frequency Test Within a Block has as reference the χ^2 distribution.

Let's split the input sequence in $N = \left\lfloor \frac{n}{M} \right\rfloor$ non-overlapping blocks. If some bits are leftover, they will be discarded, not having influence in the statistical test.

The proportion of each M-bit block, i is determined by $\pi_i = \frac{\sum_{j=1}^{M} \epsilon_{(i-1)M+j}}{M}$.

The $\chi^2 = 4M \sum_{i=1}^{N} (\pi_i - 1/2)^2$ statistic is computed, along with

$$P\text{-}value = igame\left(\frac{N}{2}, \frac{\chi^2(obs)}{2}\right)$$

where $igame$ is the incomplete gamma function defined as

$$\Gamma(z) = \int_0^{\infty} t^{z-1} e^{-t} dt \tag{6.8}$$

If $P\text{-}value < 0.01$ (at 1% significance level), it can be concluded that the sequence is non-random. Small values of the $P\text{-}value$ indicate a large imbalance between the number of zeros and ones in at least one of the M-blocks.

It is recommended by NIST Statistic Test Suit ([Rukhin et al., 2010]) to use sequences with a minimum of 100 bits and it should be notice that $n \geq MN$.

6.5.4 Runs Test

Let's define a run as an uninterrupted sequence of zeros or ones. Thus, a run of length k has k equal bits bounded at the beginning and at the end by a bit of the opposite value.

This test determines whether the total number of runs of various lengths existing in the sequence to be evaluated resembles the expected number of runs existing in a truly random sequence or not. Particularly, this test evaluates if the oscillation between runs is too fast or too slow. By oscillation one means changing from zero to one or one to zero. The Runs Test has as reference the χ^2 distribution and carries out a Frequency Test as a prerequisite.

The first step is compute the proportion of ones in the sequence ϵ as $\pi = \frac{\sum_j \epsilon_j}{n}$.

The prerequisite needs to be calculated, i.e., the Frequency Test must be compute. If $|\pi - 1/2| \geq \tau$, where $\tau = \frac{2}{\sqrt{n}}$, the Runs Test does not need to be performed and the $P\text{-}value$ is set to 0.0000.

If the Runs Test is applicable, the test statistic $v_n(obs) = \sum_{k=1}^{n-1} r(k) + 1$ is computed, where $r(k) = 0$ if $\epsilon_k = \epsilon_{k+1}$ and $r(k) = 1$ otherwise. The *P-value* is calculated as $P\text{-}value = erfc\left(\frac{|v_n(obs) - 2n\pi(1-\pi)|}{2\sqrt{2n}\pi(1-\pi)}\right)$, where $erfc$ is defined as the equation 6.7.

If $P\text{-}value < 0.01$ (at 1% significance level), it can be concluded that the sequence is non-random. Otherwise it can be concluded that the sequence is random. A large value of $v_n(obs)$ indicates a fast oscillation in the sequence, which means that there are lots of changes between the values in the sequence. On the other hand, a small value of $v_n(obs)$ indicates a slow oscillation, which means that the sequence has fewer runs than what would be expected.

In the NIST Statistic Test Suit ([Rukhin et al., 2010]) the recommendation is for each sequence to be tested to have a minimum of 100 bits.

6.5.5 Test for the Longest Run of Ones in a Block

This test analyzes the longest run of ones within M-bit blocks, i.e., it evaluates if the size of the longest runs of ones in the tested sequence is consistent with the expected size in a trully random sequence. It should be noted that an irregularity in the expected length of the longest run of ones implies that there is also an irregularity in the expected length of the longest run of zeros. Therefore, one only needs to test the longest run of ones.

The Test for the Longest Run of Ones in a Block has as reference the χ^2 distribution.

The first step is to divide the sequence into M-bit blocks an then tabulate the frequencies, v_i, of the longest runs of ones in each block into categories, where each cell contains the number of runs of ones of a given length.

The next step is to compute $\chi^2(obs) = \sum_{i=0}^{k} \frac{(v_i - N\pi_i)^2}{N\pi_i}$, where, in the NIST Statistic Test Suit ([Rukhin et al., 2010]), the values of the theoretical probabilities π_i are provided by ([Revesz, 1990]). In the NIST Statistic Test Suit ([Rukhin et al., 2010]) the values of K and N are determined by the value of M (the values presented in the M column are the values supported by their test code) in accordance with Table 6.2:

M	K	N
8	3	16
128	5	49
10^4	6	75

Table 6.2: Values of N and K according to the values of M.

The $P\text{-}value = igame\left(\frac{K}{2}, \frac{\chi^2(obs)}{2}\right)$, where $igame$ is given by the equation 6.8, is computed and, if $P\text{-}value < 0.01$ (at 1% significance level), it can be concluded that the sequence is non-random. Otherwise it can be concluded that the sequence is random. Large values of $\chi^2(obs)$ indicate that the tested sequence has clusters of ones.

In the NIST Statistic Test Suit ([Rukhin et al., 2010]) the recommendation is for each sequence to be tested to have a minimum number of bits, in accordance with the values of the sequence length, n, as specified in Table 6.3:

Minimum n	M
128	8
6272	128
750000	10^4

Table 6.3: Acceptable values for M according to the sequence to be tested minimun length, n.

It should be notice that the number of bits, i.e., the values of M are the NIST test code values acceptable for M.

6.5.6 Binary Matrix Rank Test

The purpose of this test is to check for a linear dependence among fixed length substrings of the original sequence. In other words, this test evaluates the rank of disjoint sub-matrices of the entire sequence. This statistical test also appears in the DIEHARD battery of tests ([Marsaglia, 1995]). The Binary Matrix Rank Test has as reference the χ^2 distribution.

Let M be the number of rows and Q be the number of columns in each matrix. For the NIST Statistic Test Suite, M and Q have been set to 32. If other values of M and/or Q are used, new approximations need to be computed.

The first step is to sequentially divide the sequence into $M \cdot Q$-bit disjoint blocks wherein the discarded bits will be reported as not being used in the computation within each block and then, the $M \cdot Q$-bit segments will be collected into a M by Q matrices. In the end there will be $N = \left\lfloor \frac{n}{MQ} \right\rfloor$ disjoint blocks.

Then, the binary rank, R_l, where $l = 1, \ldots, N$, of each matrix will be determined. For more information about the method used by NIST to determined the rank, one can check [Rukhin et al., 2010], Appendix A.

Let F_M be the number of matrices with $R_l = M$ (full rank), F_{M-1} be the number of matrices with $R_l = M - 1$ (full rank - 1) and $N - F_M - F_{M-1}$ be the number of the remaining matrices. The computation of $\chi^2(obs) = \frac{(F_M - 0.2888N)^2}{0.2888N} + \frac{(F_{M-1} - 0.5776N)^2}{0.5776N} + \frac{(N - F_M - F_{M-1} - 0.1336N)^2}{0.1336N}$ is done and then, the $P\text{-}value = e^{-\chi^2(obs)/2}$ is calculated.

If $P\text{-}value < 0.01$ (at 1% significance level), it can be concluded that the sequence is non-random. Otherwise it can be concluded that the sequence is random. Small values of the $P\text{-}value$ indicate a deviation of the rank distribution from what it is expected in a true random sequence.

In the NIST Statistic Test Suit ([Rukhin et al., 2010]) the probabilities for $M = Q = 32$ have been calculated and inserted into the test code. Other choices of M and Q may be selected, but the probabilities would need to be recalculated. The minimum number of bits to be tested, n, must be such that $n \geq 38MQ$, i.e., at least 38 matrices must be created. For $M = Q = 32$,

each sequence to be tested should have a minimum of 38,912 bits.

6.5.7 Discrete Fourier Transform (Spectral) Test

The focus of this test is the peak heights in the Discrete Fourier Transform of the sequence. The purpose of this test is to detect periodic features, i.e., repetitive patterns that are near to each other in the tested sequence. The patterns would indicate a deviation from the assumption of randomness. To detect this periodic features, one have to test the peak heights in the Discrete Fourier Transform of the sequence and check whether the number of peaks exceeding the 95% threshold is significantly different than 5%.

The Discrete Fourier Transform (Spectral) Test has as reference the standard normal distribution.

The first step is to create the sequence $X = x_1, x_2, \ldots, x_n$, where $x_i = 2\epsilon_i - 1$. This sequence will be the transformation of the zeros and ones of the input sequence, ϵ, into -1 and 1. Then, the Discrete Fourier Transform (DFT) will be applied on X, producing $S = DFT(X)$, that will be a sequence of complex variables. This sequence represents the periodic components of ϵ at different frequencies. Then $M = modulus(S') \equiv |S|$ is computed. S' is the first of the first $n/2$ elements substring in S, and the $modulus$ function produces a sequence of peak heights.

Then, $T = \sqrt{\left(\log \frac{1}{0.05}\right) n}$ is calculated and represents 95% of the peak threshold value, which, under the assumption of randomness, 95% of the values obtained from the test should not exceed T.

Let $N_0 = \frac{0.95n}{2}$ be the expected theoretical number of peaks that are less than T (95% under the assumption of randomness) and let N_1 be the actual observed number of peaks in M less than T.

d is the normalized difference between the observed and the expected number of frequency components that are above the 95% threshold and it is computed as $d = \frac{N_1 - N_0}{\sqrt{\frac{n \times 0.95 \times 0.05}{4}}}$.

The last step is to calculate the $P\text{-}value = erfc\left(\frac{|d|}{\sqrt{2}}\right)$, where $erfc$ is given by the equation 6.7.

If $P\text{-}value < 0.01$ (at 1% significance level), it can be concluded that the sequence is non-random. Otherwise it can be concluded that the sequence is random. A small value of d means that exits a few number of peaks (less then 95%) below T, and a large number of peaks (more than 5%) above T.

In the NIST Statistic Test Suit ([Rukhin et al., 2010]) the recommendation is for each sequence to be tested to have a minimum length of 1000 bits.

6.5.8 Non-overlapping Template Matching Test

This test is designed to detect generators that produce too many occurrences of a given non-periodic pattern, previously specified by counting the number of times the pattern occurs in the tested sequence. For this test an m-bit window is used to search for a specific m-bit

pattern. If the pattern is not found, the window slides one bit position. If the pattern is found, the window is reset to the bit after the found pattern, and the search continues.

The Non-overlapping Template Matching Test has as reference the χ^2 distribution.

The sequence must be split into N independent blocks of length M.

Let W_j, $j = 1, \ldots, N$, be the number of times that the pre-defined pattern B occurs within the block j. As said before, the method to continue the search is done by creating an m-bit window on the sequence and matching the bits within that window against the template. If there is no match, the window slides over one bit but if there is a match, the window slides over m bits.

$\mu = \frac{M-m+1}{2^m}$ and $\sigma^2 = M\left(\frac{1}{2^m} - \frac{2m-1}{2^{2m}}\right)$ are computed under the assumption of randomness and, then, $\chi^2(obs) = \sum_{j=1}^{N} \frac{(W_j - \mu)^2}{\sigma^2}$ is computed.

The *P-value* will be calculated for each template of the test. Thus, there will be multiple values of the *P-value*, calculated as $P\text{-}value = igame\left(\frac{N}{2}, \frac{\chi^2(obs)}{2}\right)$, where $igame$ is given by equation 6.8.

If $P\text{-}value < 0.01$ (at 1% significance level), it can be concluded that the sequence is non-random. Otherwise it can be concluded that the sequence is random. Small values of the *P-value* indicate the existence of a number of occurrences of a pattern in the sequence that does not correspond to the estimated number of occurrences of that pattern in a true random sequence.

In the NIST Statistic Test Suit ([Rukhin et al., 2010]), the test code for the Non-overlapping Template Matching Statistic Test has been written to provide templates for $m = 2, 3, \ldots, 10$. Thus, it is recommended for m to be equal to 9 or 10 in order to collect meaningful results. Although N has been set to 8, the assigned value can be modified in the source code. However, it is recommended for N to be less or equal to 100, in order to ensure to be assured that the calculated *P-values* are valid. Additionally, one may ensure that $M > 0.01 \cdot n$ and $N = \lfloor n/M \rfloor$.

6.5.9 Overlapping Template Matching Test

The Overlapping Template Matching Test is design to detect the number of occurrences of a given pattern in the tested sequence, defined *à posteriori*. This test also uses a m-bit window to search for a specific m-bit pattern. If the pattern is not found, the window slides one bit position. The difference between this test and the test described in Section 6.5.8 lies in the cases where the pattern is found. In this test, if the pattern is found, the window slides only one bit before resuming the search.

The Overlapping Template Matching Test has as reference the χ^2 distribution.

Analogously to the Non-Overlapping Template Machine Test, the sequence to be tested must be split into N independent blocks of length M.

The number of occurrences of the template B, in each one of the N blocks is calculated in a similar way to the Non-Overlapping Template Machine Test: the search for correspondences proceeds by creating an m-bit window on the sequence, comparing the bits within that window against the template and incrementing a counter when a correspondence is found. The difference

between this calculation and the calculation seen in Section 6.5.8 is that in this test, the window slides over one bit after each examination. In the NIST Statistic Test Suit ([Rukhin et al., 2010]), the number of occurrences of B in each block is recorded by incrementing the values, v_i (where $i = 0, \ldots, 5$), that are stored in an array, such that v_0 is incremented when there are no matches of B in a substring of the tested sequence, v_1, v_2, v_3, v_4 are incremented for one, two, three or four occurrences of B in the tested sequence, respectively, and v_5 is incremented for five or more occurrences of B.

$\lambda = \frac{M - m + 1}{2^m}$ and $\eta = \frac{\lambda}{2}$ are calculated and are going to be used in the computation of the theoretical probabilities, π_i.

Then, $\chi^2(obs) = \sum_{i=0}^{5} \frac{(v_i - N\pi_i)^2}{N\pi_i}$ is computed, where $\pi_0 = 0.364091$, $\pi_1 = 0.185659$, $\pi_2 = 0.139381$, $\pi_3 = 0.100571$, $\pi_4 = 0.070432$ and $\pi_5 = 0.139865$ ([Hamano and Kaneko, 2007]).

The last step consists in calculate the *P-value*: $P\text{-}value = igame\left(\frac{5}{2}, \frac{\chi^2(obs)}{2}\right)$, where $igame$ is given by the equation 6.8.

If $P\text{-}value < 0.01$ (at 1% significance level), it can be concluded that the sequence is non-random. Otherwise it can be concluded that the sequence is random.

In the NIST Statistic Test Suit ([Rukhin et al., 2010]), the values of K, M and N have been chosen such that each sequence to be tested has a minimum of 10^6 bits. Various values of m may be selected, but NIST recommends m to be equal to 9 or 10. If other values are desired, one has to be careful to chose the values according to the following specifications:

- $n \geq MN$
- N should be chosen so that $N \cdot (min(\pi_i)) > 5$
- $\lambda = \frac{M - m + 1}{2^m} \approx 2$
- $m \approx \log_2 M$
- $K \approx 2\lambda$ (for $K \neq 5$, the π_i values need to be recalculated)

6.5.10 Maurer's "Universal Statistical" Test

This test aims to determine whether a sequence can be compressed without loss of information or not. Thus, the focus of this test is the number of bits between the matching patterns, and this is a measure related to the compression length of a sequence. If a sequence is able to be compressed without loss of information, this sequence is considered non-random.

The reference distribution for the Maurer's "Universal Statistical" Test is the half-normal distribution (a one-sided variant of the normal distribution).

The sequence to be tested, ϵ, composed of n-bits is partitioned into two segments: an initialization segment consisting of Q L-bit non-overlapping blocks, and a test segment consisting of K L-bit non-overlapping blocks. The remaining bits at the end of the sequence that do not

form a complete L-bit block are discarded. The first Q blocks are used to initialize the test and the remaining K blocks are the test blocks. K, n, L, and Q follow the relation: $K = \lfloor n/L \rfloor - Q$.

Using the initialization segment, a table, T_j is created for each possible L-bit value. The block number of the last occurrence of each L-bit block is noted in the table, i.e., for $i = 1, \ldots, Q-1$, $T_j = i$, where j is the decimal representation of the contents of the i^{th} L-bit block.

$sum = sum + \log_2 i - T_j$ is calculated, i.e., for each K block in the sequence the number of blocks since the last occurrence of the same L-bit block is determined. The obtained value is replaced in the table with the current block. Then, the calculated distance between re-occurrences of the same L-bit block and the accumulating $\log_2 sum$ of all the differences detected in the K blocks are added.

The test statistic $f_n = \frac{1}{K} \sum_{i=Q+1}^{Q+K} \log_2 (i - T_j)$, where T_j is the entry table corresponding to the decimal representation of the contents of the i^{th} L-bit block, is computed. Then, the *P-Value* is computed as $P\text{-}value = erfc\left(\left|\frac{f_n - expectedValue(L)}{\sqrt{2}\sigma}\right|\right)$, where $erfc$ is given by the equation 6.7 and the theoretical standard deviation is given by $\sigma = c\sqrt{\frac{variance(L)}{K}}$, where $c = 0.7 - \frac{0.8}{L} + \left(4 + \frac{32}{L}\right)\frac{K^{-3/L}}{15}$.

If $P\text{-}value < 0.01$ (at 1% significance level), it can be concluded that the sequence is non-random. Otherwise it can be concluded that the sequence is random.

If f_n differs significantly from $expectedValue(L)$, then the sequence is significantly compressible.

This test requires a long sequence of bits. In fact, n must be greater or equal to $(Q+K)L$ since the sequence to be tested will be divided into two segments of L-bit blocks. In their Statistic Test Suit ([Rukhin et al., 2010]), NIST recommends to choose L, Q and K so that $6 \leq L \leq 16$, $Q = 10 \cdot 2^L$ and $K = \lceil n/L \rceil - Q \approx 1000 \cdot 2^L$. NIST Statistic Test Suit ([Rukhin et al., 2010]) offers a table (Table 6.4) that represents how the values of L, Q and n should be chosen. The only input is n, wherein L and Q are assigned depending on the value of n.

6.5.11 Linear Complexity Test

This test determines whether a sequence is complex enough to be considered random. The random sequences are characterized by having a long linear feedback shift register (LFSR). If a bit sequence has a very short LFSRs, it can be conclude that the sequence is non-random. Thus, this test measures the length of the LFSR.

The Linear Complexity Test has as reference the χ^2 distribution.

The sequence to be tested, ϵ, composed of n-bits is partitioned into N independent blocks of M bits, where $n = MN$.

The linear complexity, L_i of each one of the N ($i = 1, \ldots, N$) blocks of the sequence is calculated using the Berlekamp-Massey algorithm, described in Chapter 6 of [Menezes et al., 1996]. The linear complexity, L_i, is the shortest linear feedback shift register sequence length that generates all bits in the correspondent block i. Within any L_i-bit sequence, some combi-

n	L	$Q = 10 \cdot 2^L$
≥ 387840	6	640
≥ 904960	7	1280
≥ 2068480	8	2460
≥ 4654080	9	5120
≥ 10342400	10	10240
≥ 22753280	11	20480
≥ 49643520	12	40960
≥ 107560960	13	81920
≥ 231669760	14	163840
≥ 496435200	15	327680
≥ 1059061760	16	655360

Table 6.4: Representative table of how the values of L, Q and n should be chosen.

nation of the bits, when added together modulo 2, produces the next bit in the sequence, the bit $L_i + 1$.

The theoretical mean, μ is calculated as $\mu = \frac{M}{2} + \frac{9 + (-1)^{M+1}}{36} - \frac{M/3 + 2/9}{2^M}$, under the assumption of randomness and for each substring, the value of T_i e also calculated, where $T_i = (-1)^M \cdot (L_i - \mu) + \frac{2}{9}$. The values of T_i are recorded in $v_0, v_1, \ldots, v_6$ described in Table 6.5.

If:	$T_i \leq -2.5$	, increment v_0 by one
	$-2.5 < T_i \leq -1.5$	, increment v_1 by one
	$-1.5 < T_i \leq -0.5$	, increment v_2 by one
	$-0.5 < T_i \leq 0.5$	, increment v_3 by one
	$0.5 < T_i \leq 1.5$	, increment v_4 by one
	$1.5 < T_i \leq 2.5$	, increment v_5 by one
	$T_i > 2.5$	, increment v_6 by one

Table 6.5: Method of recording the values of T_i in $v_0, v_1, \ldots, v_6$.

$\chi^2(obs) = \sum_{i=0}^{K} \frac{(v_i - N\pi_i)^2}{N\pi_i}$ is computed, where $\pi_0 = 0.010417$, $\pi_1 = 0.03125$, $\pi_2 = 0.125$, $\pi_3 = 0.5$, $\pi_4 = 0.25$, $\pi_5 = 0.0625$, $\pi_6 = 0.020833$ are the probabilities computed by the equations presented in Chapter 3 of [Rukhin et al., 2010].

The final step is the computation of the *P-value* as $P\text{-}value = igame\left(\frac{K}{2}, \frac{\chi^2(obs)}{2}\right)$, where $igame$ is given by the equation 6.8.

If $P\text{-}value < 0.01$ (at 1% significance level), the observed frequency counts of T_i stored in the v_l bins varied from the expected values. Thus, if $P\text{-}value < 0.01$, it can be concluded that

the sequence is non-random. Otherwise it can be concluded that the sequence is random.

In the NIST Statistic Test Suit ([Rukhin et al., 2010]) the recommendation is for each sequence to be tested to have a minimum of 10^6 bits. The value of M must be in the range $500 \leq M \leq 5000$, and $N \geq 200$ for the χ^2 result to be valid.

6.5.12 Serial Test

This test determines whether the number of occurrences of the 2^m m-bit overlapping patterns is approximately the same as would be expected for a truly random sequence or not. On random sequences, every m-bit pattern has the same possibility of show up in the sequence as every other m-bit pattern. This is due to the existence of uniformity in random sequences. Thus, this test measures the frequency of all possible overlapping m-bit patterns across the whole sequence. For $m = 1$, the Serial test is equivalent to the Frequency test described in Section 6.5.2.

The Serial Test has as reference the χ^2 distribution.

The sequence ϵ is going to be extended by appending the first $m-1$ bits sequence to the end of the sequence for distinct values of n that are given as input. Thus, a new and augmented sequence, ϵ', is created out of the sequence ϵ.

The frequency of all possible overlapping m-bit, $(m-1)$-bit and $(m-2)$-bit blocks are calculated. $v_{i_1,\ldots,i_m}$ denote the frequency of the m-bit pattern $i_1, \ldots, i_m$, $v_{i_1,\ldots,i_{m-1}}$ denote the frequency of the $(m-1)$-bit pattern $i_1, \ldots, i_{m-1}$ and $v_{i_1,\ldots,i_{m-2}}$ denote the frequency of the $(m-2)$-bit pattern $i_1, \ldots, i_{m-2}$.

The computation of ψ^2_m, ψ^2_{m-1} and ψ^2_{m-2} is done as:

$$\psi^2_m = \frac{2^m}{n} \sum_{i_1,\ldots,i_m} v^2_{i_1,\ldots,i_m} - n$$
$$\psi^2_{m-1} = \frac{2^{m-1}}{n} \sum_{i_1,\ldots,i_{m-1}} v^2_{i_1,\ldots,i_{m-1}} - n$$
$$\psi^2_{m-2} = \frac{2^{m-2}}{n} \sum_{i_1,\ldots,i_{m-2}} v^2_{i_1,\ldots,i_{m-2}} - n$$

Then, $\bigtriangledown\psi^2_m = \psi^2_m - \psi^2_{m-1}$ and $\bigtriangledown^2\psi^2_m = \psi^2_m - 2\psi^2_{m-1} + \psi^2_{m-2}$ are calculated.

In the last step, two *P-values* are calculated: *P-value1* $= igame(2^{m-2}, \bigtriangledown\psi^2_m)$ and *P-value2* $= igame(2^{m-3}, \bigtriangledown^2\psi^2_m)$, where $igame$ is given by equation 6.8.

If *P-value1* < 0.01 and *P-value2* < 0.01 (at 1% significance level), it can be concluded that the sequence is non-random. Otherwise it can be concluded that the sequence is random. For very large values of $\bigtriangledown^2\psi^2_m$ e $\bigtriangledown\psi^2_m$ the m-bit blocks are not uniform.

In the NIST Statistic Test Suit ([Rukhin et al., 2010]) the recommendation is to choose m and n such that $m < \lfloor \log_2 n \rfloor - 2$.

6.5.13 Approximate Entropy Test

This test measures the frequency of all possible overlapping m-bit patterns across the whole sequence, analogously to the test described in Section 6.5.12. The scope of the Approximate

Entropy Test, is to compare the frequency of overlapping blocks of two consecutive/adjacent lengths against the proposed result for a random sequence.

The Approximate Entropy Test has as reference the χ^2 distribution.

The n-bit sequence ϵ is going to be augmented by appending $m-1$ bits from the beginning of the sequence to the end of the sequence. Thus, a new n overlapping m-bit sequence is created.

A frequency count is made of the n overlapping blocks. Let the count of the possible m-bit ($(m+1)$-bit) values be represented as C_i^m, where i is the m-bit value.

$C_i^m = \frac{\#i}{n}$ is calculated for each values of i and, then $\varphi^{(m)} = \sum_{i=0}^{2^m-1} \pi_i \log \pi_i$ is computed, where $\pi_i = C_j^3$ and $j = \log_2 i$.

All this procedure is repeated, replacing m by $m+1$.

Then, the statistic $\chi^2 = 2n(\log 2 - ApEn(m))$, where $ApEn(m) = \varphi^{(m)} - \varphi^{(m+1)}$ is computed and $P\text{-}value = igame\left(2^{m-1}, \frac{\chi^2}{2}\right)$ is calculated, where $igame$ is given by the equation 6.8.

If $P\text{-}value < 0.01$ (at 1% significance level), the conclusion is that the sequence is non-random. Otherwise it can be concluded that the sequence is random. Small values of $ApEn(m)$ imply a strong regularity in the tested sequence, while large values of $ApEn(m)$ imply a substantial fluctuation or irregularity.

In the NIST Statistic Test Suit ([Rukhin et al., 2010]) the recommendation is to choose m and n such that $m < \lfloor \log_2 n \rfloor - 5$.

6.5.14 Cumulative Sums Test

This test determines if the cumulative sum of the partial sequences occurring in the tested sequence is too large or if it is too small when comparing it to the expected behaviour in a truly random sequence. One can look at the cumulative sum as a random walk. This test determines the maximal excursion (from zero) of the random walk defined by the cumulative sum of adjusted (-1, +1) digits in the sequence. On a random sequence, the excursions of the random walk should be near zero, while in a non-random sequence, the excursions of the random walk should be greater than zero.

The Cumulative Sums Test has as reference the standard normal distribution.

The zeros and ones of the inputted sequence are converted to -1 and $+1$ values, using $X_i = 2\epsilon_i - 1$, creating the sequence X_i that will be a normalized sequence.

The partial sums S_i of successively larger subsequences are calculated. There are two modes for compute S_i: mode = 0, where the sum starts with X_1 and mode = 1, where the sum starts with X_n.

The test statistic, z, is going to be $max_{1 \leq k \leq n}(|S_k|)$, the largest of the partial sums, S_k, absolute values.

The last step is the calculation of the $P\text{-}value$ as:

Mode = 0 (forward)	Mode = 1 (backward)
$S_1 = X_1$	$S_1 = X_n$
$S_2 = X_1 + X_2$	$S_2 = X_n + X_{n-1}$
$\vdots$	$\vdots$
$S_k = X_1 + X_2 + \cdots + X_k$	$S_k = X_n + X_{n-1} + \cdots + X_{n-k+1}$
$\vdots$	$\vdots$
$S_n = X_1 + X_2 + \cdots + X_k + \cdots + X_n$	$S_n = X_n + X_{n-1} + \cdots + X_{n-k-1} + \cdots + X_1$

$$\begin{aligned} \textit{P-value} \;=\; & 1 - \sum_{k=\frac{-n/z+1}{4}}^{\frac{n/z-1}{4}} \left[\phi\left(\frac{(4k+1)z}{\sqrt{n}} \right) - \phi\left(\frac{(4k-1)z}{\sqrt{n}} \right) \right] \\ & + \sum_{k=\frac{-n/z-3}{4}}^{\frac{n/z-1}{4}} \left[\phi\left(\frac{(4k+3)z}{\sqrt{n}} \right) - \phi\left(\frac{(4k+1)z}{\sqrt{n}} \right) \right] \end{aligned}$$

If *P-value* < 0.01 (at 1% significance level), one can conclude that the sequence is non-random. Otherwise it can be concluded that the sequence is random.

When mode = 0, if the statistic result is a large value, two scenarios are possible to occur: either exists "too many ones" or "too many zeros" at early stages of the sequence. When mode = 1, if the statistic result is a large value, the two possible scenarios are: either exists "too many ones" or "too many zeros" at late stages of the sequence. Either in mode = 0 or in mode = 1, if the statistic result is a small value, the indication is that ones and zeros are intermixed too evenly in the sequence.

In the NIST Statistic Test Suit ([Rukhin et al., 2010]) the recommendation is for each sequence to be tested to have a minimum of 100 bits.

6.5.15 Random Excursions Test

This test is also based on cumulative sums, analogously to the test described in Section 6.5.14. As said before, a cumulative sum can be seen as a random walk. The cumulative sum random walk is derived from partial sums after the $(0, 1)$ sequence is transferred to the appropriate $(-1, +1)$ sequence. This test determines the number of cycles that have exactly K visits in a cumulative sum random walk. A cycle of a random walk consists of a sequence with unit length steps taken at random that begin at the origin return to the origin. Thus, this test determines if the number of visits to a particular state within a cycle deviates from the number of visits expected in a random sequence.

The Random Excursions Test has as reference the χ^2 distribution.

The zeros and ones of the input sequence are converted into -1 and $+1$, respectively, using $X_i = 2\epsilon_i - 1$, creating X, which will be a normalized sequence.

The partial sums, S_i, $i = 1, \cdots, n$, of successively larger subsequences, each one starting with X_1 are computed, forming the set $S = \{S_i\}$, with S_i defined as:

$$\begin{aligned} S_1 &= X_1 \\ S_2 &= X_1 + X_2 \\ &\vdots \\ S_k &= X_1 + X_2 + \cdots + X_k \\ &\vdots \\ S_n &= X_1 + X_2 + \cdots + X_k + \cdots + X_n \end{aligned}$$

Then, a new sequence S' is formed by attaching zeros before and after the S set. Thus, $S' = 0, S_1, S_2, \ldots, S_n, 0$, and this can be seen as a random sequence.

Let J be the total number of zero crossings in S', where a zero crossing is a value of zero in S' that occurs after the starting zero. J can also be seen as the number of cycles in S'. A cycle of S' must understand as the subsequences of S' consisting of an occurrence of zero, followed by non-zero values, and ending with another zero. In the end if $J < 500$, the test must be discontinued ([Rukhin et al., 2010], Chapter 3).

For each cycle and for each non-zero state value x having values satisfying the conditions $-4 \leq x \leq -1$ and $1 \leq x \leq 4$, the frequency of each x within each cycle is computed. And, for each one of the 8 states of x, $v_k(x)$ ($k = 0, 1, \cdots, 5$) is calculated, where $v_k(x)$ is the total number of cycles in which state x occurs exactly k times among all cycles, where all frequencies greater or equal than five are stored in $v_5(x)$.

The statistic $\chi^2(obs) = \sum_{k=0}^{5} \frac{(v_k(x) - J\pi_k(x))^2}{J\pi_k(x)}$ is calculated for each one of the 8 states of x. The values for $\pi_k(x)$ and their method of calculation can be checked in chapter 3 from [Rukhin et al., 2010], Chapter 3.

The last step is the calculation of the 8 (there are eight values of χ^2) values of *P-value* as: $\textit{P-value} = igame\left(\frac{5}{2}, \frac{\chi^2(obs)}{2}\right)$, where $igame$ is given by the equation 6.8.

If *P-value* < 0.01 (at 1% significance level), it can be concluded that the sequence is non-random. Otherwise it can be concluded that the sequence is random.

If $\chi^2(obs)$ is a too large value, it means that the sequence displays a deviation from the theoretical distribution for a given state across all cycles.

In the NIST Statistic Test Suit ([Rukhin et al., 2010]) the recommendation is for each sequence to be tested to have a minimum of 1000000 bits.

6.5.16 Random Excursions Variant Test

This test detects possible existante deviations from the expected number of visits to various states in the random walk. In order to detect that, this test counts the total number of times that a particular state occurs in a cumulative sum random walk.

The reference distribution for the Random Excursions Variant Test for large values of n is the half-normal distribution (a one-sided variant of the normal distribution).

The zeros and ones of the input sequence are converted into -1 and $+1$, respectively, using $X_i = 2\epsilon_i - 1$ and forming the sequence X_i that will be a normalized sequence.

The partial sums, S_i $(i = 1, \cdots, n)$ of successively larger subsequences, each one starting with X_1 are computed, forming the set $S = \{S_i\}$, with S_i defined as:

$$\begin{aligned} S_1 &= X_1 \\ S_2 &= X_1 + X_2 \\ &\vdots \\ S_k &= X_1 + X_2 + \cdots + X_k \\ &\vdots \\ S_n &= X_1 + X_2 + \cdots + X_k + \cdots + X_n \end{aligned}$$

Then, a new sequence S' is formed by attaching zeros before and after the S set. Thus, $S' = 0, S_1, S_2, \ldots, S_n, 0$, and this can be seen as a random sequence.

For each one of the non-zero states of x (in the NIST tests the number of states is 18), the total number of times that state x occurred across all J cycles, $\xi(x)$, is computed.

For each value of $\xi(x)$, $P\text{-}value = erfc\left(\frac{|\xi - J|}{\sqrt{2J(4|x| - 2)}}\right)$, described before in Section 6.5.15, is computed, where $erfc$ is given by the equation 6.7.

If $P\text{-}value < 0.01$ (at 1% significance level), it can be concluded that the sequence is non-random. Otherwise it can be concluded that the sequence is random.

In the NIST Statistic Test Suit ([Rukhin et al., 2010]) the recommendation is for each sequence to be tested to have a minimum of 1000000 bits.

6.6 Recommendations for Random Numbers Generation. Randomness Requirements for Security.

The described recommendations are mainly based on [Eastlake 3rd et al., 2005], [FIP, 2002] and [Rukhin et al., 2010]. These recomendations are the ones proposed by the National Institute of Standards and Technology (NIST), the responsible entity for safety standards specification.

Security systems are built on strong cryptographic algorithms that defend themselves from pattern analysis attempts and the central focus of all cryptographic systems is the generation of secret, unguessable, random numbers. These systems provide substantial protection against snooping and spoofing. The use of pseudorandom processes to generate secret quantities can result in pseudo-security, if no care is taken into consideration when implementing them. A sophisticated attacker may find easier to reproduce the environment of the secret quantities production and to search the resulting small set of possibilities than to search all the possibilities for the generated number ([Eastlake 3rd et al., 2005]).

The cryptographic systems are designed to provide a massive protection against snooping and spoofing. However, these kind of protection mostly relies on the generation of secret and unguessable random numbers. The random numbers generation may represent a potential flaw by the way they are generated. In fact, the generally lack of available facilities for generating such random numbers presents a glitch in the cryptographic software design.

Cryptographic techniques can be used to provide a variety of services such as confidentiality and authentication. This type of services require an unknown and unguessable, namely random quantities, called "keys".

There are two different types of random quantities that may be wanted. In the case of human-usable passwords, the only main requirement is that they have to be unguessable. The other case, the case of fixed length keys, the requirements are more demanding: the values should be random, i. e., the bits of these values must pass statistical randomness tests.

The frequency and requirement for random quantities vary from cryptographic system to cryptographic system. For example, in pure RSA, random quantities are required only when a new key pair is generated. Thereafter, any number of messages can be signed without a further need for randomness. The public key Digital Signature Algorithm proposed by NIST requires good random numbers for each signature ([dss, 2000]).

An attacker can always try the "brute-force" method to try to find the secret key and this kind of attack is possible as long as the key is smaller enough. The probability of an adversary to be succeed must be significantly low. The success of the attack depends on the amount of key information available and on the number of the secret possible values. The probability of each value is given by:

$$bits \quad of \quad information = \sum -p_i \log_2(p_i) \tag{6.9}$$

where i begins with value 1 until reaching the number of possible secret values, and p_i is the probability of the value numbered i.

The probability of the attacker to be succeed also depends on the amount of key information available because, for example, considering a cryptosystem that uses 128 bit keys, if the attacker knows that the pseudorandom generator is fixed and seeded by an 8 bits seed, then he/she only needs to search trough the keys to find the right one by running the pseudorandom generator with every possible seed. In this example there are only 8 bits of information in the 128 bit keys. For any number, n, of bits of information, there are 2^n different values of equal probability. In this case, an attacker would have to try, on the average, half of the values, or 2^{n-1} before guessing the secret key. However, if the probability of different values is unequal, the attacker will require a fewer guesses to find the secret key. In fact, the attacker can initially ignore the low probability values and search trough the more provable ones first.

6.6.1 Entropy Sources

A way to accomplish the generation of true and strong randomness is the availability of a non-deterministic physical source of unpredictable numbers, an entropy source, to use as a generator input.

The best input entropy sources are the hardware-based random sources that could be easily included as a standard part of a computer system's architecture like the majority of audio/video input devices are ([Denker, 2003]), such as ring oscillators, disk drive timing, thermal noise, or radioactive decay, but there also exists other non-hardware possibilities like system clocks, system or input/output buffers, user/system/hardware/network serial numbers or addresses and timing, user input and user processes like key strokes or mouse movements ([Gifford, 1988]).

The DRBG mechanism uses a seed that is determined from the entropy input and both the entropy input as the seed shall be kept secret. The basis of the RBG security rely in the secrecy of this information.

The entropy input shall provide at least the minimum amount of entropy requested by the DRBG mechanism. Ideally, the entropy input will have full entropy, which means that each bit of a generated bitstring in this conditions is unpredictable, follows a uniform distribution and is independent from every other bit in the bitstring.

The input entropy length does not necessarily need to have a fixed value. Thus, the DRBG mechanisms have been specified to allow for some bias in the entropy input by allowing the entropy input length to be longer than the required entropy amount. When an entropy input is requested it is expected for the returned bitstring to contain at least the requested entropy amount, although a bit more of entropy is desirable, it is not required.

A DRBG mechanism may need other information as input that may or may not be kept secret but that should be checked for validity when possible. Sometimes during DRBG instantiation, a nonce may be required, and when it is used, it is combined with the entropy input to create the initial DRBG seed. NIST strongly advises the insertion of other information as input as, for example, the use of a personalization string during DRBG instantiation. When this approach is used, the personalization string is combined with the entropy input bits and possibly a nonce to create the initial DRBG seed. The personalization string should be unique for all instantiations of the same DRBG mechanism type.

Additional input may also be provided during reseeding and when pseudorandom bits are requested.

Using Existing Sound/Video Input

Many computers are built with inputs that digitize some real-world analog source, such as sound from a microphone or video input from a camera. The input from a sound digitizer with no source plugged in or from a camera with the lens cap on is essentially thermal noise, which is essentially random noise. If the system has enough gain to detect anything, such input can provide reasonably high quality of random bits. This method is extremely dependent on the hardware implementation and, in case of hardware failure, it will have to be de-skewed (Section 6.6.2).

Using Existing Disk Drives

Disk drives have small random fluctuations in their rotational speed due to chaotic air turbulence ([Davis and Fenstermacher, 1984] [Jakobsson and Juels, 1998]). The addition of low-level disk seek-time instrumentation produces a series of measurements that contain randomness.

However, such data is usually highly correlated and, in order to accomplish some reliable randomness, it is needed to do some significant processing to the obtained data. Every increase in the processor speed increases the rate of random bit generation.

This technique is used in random number generators including many operating system libraries.

Ring Oscillator Sources

A free-running ring oscillator can be produced by connecting series of odd number of gates at the time when an integrated circuit is being designed or field-programmed, by sampling a point in the ring at a fixed frequency (for example, one determined by a stable crystal time source oscillator). Thus, some amount of entropy can be extracted due to variations in the free-running oscillator timing.

By XOR'ing sampled values from a few ring oscillators with relatively prime lengths it is possible to increase the rate of the extracted entropy.

For some situations it is recommended to use an odd number of rings. In the possibility of the rings to become synchronously locked to each other, an odd number of rings guarantees that there will still be sampled bit transitions. Another possible source to do the sampling is the output of a noisy diode. All of the sampled bits extracted from such sources will have to be heavily de-skewed (Section 6.6.2).

Clocks and Serial Numbers

Computer clocks and similar operating system or hardware values, provide significantly fewer real unpredictable bits than what might appear from their specifications.

Clocks behaviour on numerous systems can vary widely and in unexpected ways depending on the configuration of the operating system. So, designing portable application code to generate unpredictable numbers based on such system clocks is particularly challenging since the system designer does not always know the properties of the system clock. Sometimes, successive reads of the clock may produce identical values even if enough time has passed and the value should have change, based on the nominal clock resolution. There are also some situations where regular readings of a clock can produce artificial sequential values, because of extra code that checks for the clock being unchanged between two reads and increases it by one.

The use of a hardware serial number may provide quantities that are usually heavily structured, and subfields that may have only a limited range of possible values, or values that may be easily guessable based on the approximate manufacture date or based on another data.

Such problems make the production of the code to generate unpredictable quantities difficult to implement if the code is expected to be ported across a variety of computer platforms and systems.

Timing and Value of External Events

The timing and content of mouse movement, key strokes, and similar user events are possible features to be measured and thus becoming a reasonable source of unguessable data with some qualifications. However, even though the user's interkeystroke timing may have sufficient variation and unpredictability, there might not be an easy way to access that variation and this,

along with the fact that there are no standard methods for sampling timing details, comes up as a problem.

The amount of mouse movement and the actual key strokes are usually easier to access than timings. However they may yield less unpredictability because the user may provide highly repetitive input.

Network packet arrival times and lengths can be used as an external event of an entropy source. Nonetheless these sources must be used with great care since the possibility of such network traffic measurements manipulation by an adversary and the lack of history at system start-up must be carefully considered.

So, almost any external sensor, such as raw radio reception or temperature sensing in appropriately equipped computers can be used if careful consideration is given to how much this data is vulnerable to adversarial manipulation and to how much entropy it can actually provide.

Surely, a source with a considered vulnerable input must not be trusted as a source of entropy.

If the attackers have no access to the quantities being measured, the above techniques have a great potential as entropy sources.

Non-hardware Sources of Randomness

Even though the best sources of input entropy are the hardware-based sources of randomness, there are also non-hardware sources of randomness like mentioned above. Among the non-hardware sources of randomness there are the system clocks, the system or the input/output buffers, the user/system/hardware/network serial numbers or the addresses and timing, and the user input. Unfortunately, all of these sources share the same problem which is producing, under some circumstances, very limited or predictable values.

Some of the sources listed above would be quite strong on multi-user systems but weak on a small single-user or embedded system. In a multi-user systems each user can be seen as a different entropy source, but in a small single-user or embedded system, an adversary may find a pattern on the inputs that he/she can reply.

The use of multiple random inputs with a strong mixing function is recommended and can overcome weakness in any particular input. This strategy may make practical portable code for producing good random numbers for security, even if some of the inputs are very weak on some of the target systems. However, it may still fail against a high-grade attack on small, single-user, or embedded systems, especially if the adversary has ever been able to observe the generation process in the past. A hardware-based random source is still preferable.

6.6.2 De-skewing

The distribution of the entropy quantities gathered to produce the random numbers does not need to be uniform. But it is needed to estimate how non-uniform it is in order to bound performance. To accomplish the recommended distribution it is sometimes necessary to use simple techniques to de-skew a bit stream.

Using Stream Parity to De-Skew

Consider taking a sufficiently long bitstring and mapping the string to 0 or 1. The mapping will not yield a perfectly uniform distribution, but it can be as close to it as desired. Suppose the string is mapped according to its parity. This kind of system has its advantages: it is robust across all skew degrees up to the estimated maximum degree and that it is trivial to implement in hardware.

Suppose that the ratio of 1s to 0s is $0.5 + \epsilon$ to $0.5 - \epsilon$, where ϵ is a value between 0 and 0.5 and it is an "eccentricity" of the distribution measure. Consider the parity function of N bit samples distribution. The respective probabilities that the parity will be 1 or 0 will be the sum of the odd or even terms in the binomial expansion of $(p+q)^N$, where $p = 0.5 + \epsilon$, the probability of a 1, and $q = 0.5 - \epsilon$, is the probability of a 0.

These sums can be easily computed as:

$$\frac{1}{2}\Big((p+q)^N + (p-q)^N\Big) \quad \textit{and} \quad \frac{1}{2}\Big((p+q)^N - (p-q)^N\Big)$$

Since $p + q = 1$ and $p - q = 2\epsilon$, these expressions can be reduced to:

$$\frac{1}{2}\Big(1 + (2\epsilon)^N\Big) \quad \textit{and} \quad \frac{1}{2}\Big(1 - (2\epsilon)^N\Big)$$

None of these expressions will ever be exactly 0.5 unless ϵ is zero. However one can bring them arbitrarily close to 0.5 by defining some δ as small as one wishes. Then,

$$0.5 + 0.5 \times (2\epsilon)^N < 0.5 + \delta$$

Using Transition Mappings to De-Skew

The skew correction algorithm used in the technique presented here is a simple algorithm, originally due to Von Neumann ([Von Neumann, 1963]) and is based on the transition mapping. So, a bitstream is examined as a sequence of non-overlapping pairs. Bits are read two at a time, and if there is a transition between values (if bits are 01 or 10) one of them - say the first - is passed on as random. If there is no transition (if bits are 00 or 11), the bits are discarded and the next two are read.

Thus, any 00 or 11 pairs found are discarded, the pairs 01 are interpreted as 0 and the pairs 10 are interpreted as 1. Assume that the probability of a 1 is $0.5 + \epsilon$ and that the probability of a 0 is $0.5 - \epsilon$, where ϵ is the source eccentricity. Then the probability of each pair is shown in Table 6.6.

pair	probability
00	$(0.5 - \epsilon)^2$
01	$(0.5 - \epsilon) \times (0.5 + \epsilon)$
10	$(0.5 - \epsilon) \times (0.5 + \epsilon)$
11	$(0.5 + \epsilon)^2$

Table 6.6: Bit pairs probability.

This simple algorithm completely eliminates any data bias towards 0 or 1 but it requires an indeterminate number of input bits for any particular desired number of output bits. The probability of any particular pair being discarded is $0.5 + 2\epsilon^2$, so the expected number of input bits to produce x output bits is $\frac{x}{0.25 - \epsilon^2}$.

This technique assumes that the bits are from a stream where each bit has the same probability of being 0 or 1 and that the bits are uncorrelated, since if they are the above analysis breaks down.

If overlapping successive bits pairs were used instead of non-overlapping pairs, the statistical analysis would be the same but the algorithm, instead of providing an unbiased, uncorrelated series of random 1s and 0s, it will produce a totally predictable sequence of exactly alternating 1s and 0s.

Using Fast Fourier Transform to De-Skew

When real-world data consists of strongly correlated bits, it may still contain useful amounts of entropy that can be extracted through various transforms like the Fast Fourier Transform (FFT). It can be shown that this technique will discard strong correlations. If adequate data is processed and if remaining correlations decay, spectral lines that approach statistical independence and normally distributed randomness can be produced ([Brillinger, 1981]).

Using Compression to De-Skew

Reversible compression techniques also provide a crude method of de-skewing a skewed bit stream.

The compression is reversible so, the amount of information present in the shorter output must be the same as the amount of information present in the longer input. By the Shannon information equation (6.9), this is only possible if, on average, the probabilities of the different shorter sequences are more uniformly distributed than the probabilities of the longer sequences. Therefore, the shorter sequences must be de-skewed relative to the input.

However, many compression techniques add a somewhat predictable preface to their output stream and this may insert a similar sequence periodically r subtle patterns in their output. The beginning of the compressed sequence should be skipped and only later bits should be used for applications requiring roughly-random bits.

6.6.3 Mixing

A strong and reliable entropy source can be used as a RNG seed to produce the required amount of cryptographically strong pseudorandomness. However, in the absence of an entropy source there are some strategies that can be used to produce unguessable random numbers. The best strategy is to obtain input from a number of uncorrelated sources and mix them using a strong mixing function that must preserve the given entropy by any of the sources. A strong mixing function is a function that combines inputs and produces an output in which each bit is a different complex non-linear function of all the input bits. A function like this ensures that no particular output bit is guaranteed to change when any particular input bit is changed.

If the used sources are considered to be good (accordingly to the features seen in Section 6.6.1) the use of a strong mixing function creates a strong seed that can be used to produce large quantities of cryptographically strong material.

The use of a strong mixing function is an advisable procedure even with a good hardware source (since hardware is likely to fail). However, it is also a good practice to weigh the use of such a function coupled to a good hardware, since there exists the possibility of an increase in the chance of overall failure due to added software complexity.

An example of a trivial mixing function is the use of the exclusive Or (XOR) (Section 6.6.2) function that is equivalent to addition without carry.

In cases like this, if the inputs are uncorrelated, the outputs will be an even better (less skewed) random bits then the inputs are.

There are a few examples of stronger mixing functions than the ones described above such as the US Government Advanced Encryption Standard ([aes, 2001]) for multiple bit quantities and Data Encryption Standard ([des, 1999], [des, 1999], [des, 1985]).

Another good family of mixing functions is the *message digest* or hashing functions such as the US Government Secure Hash Standards (SHA*) and the MD4, MD5 ([Rivest, 1992a], [Rivest, 1992b]) series.All of these functions take a practically unlimited amount of input and produce a relatively short fixed-length output mixing all the input bits. The MD* series produces 128 bits of output, SHA-1 produces 160 bits, and other SHA functions produce up to 512 bits.

Although the message digest functions are designed for variable amounts of input, AES and other encryption functions can also be used to combine any number of inputs. However, if more than 128 bits of output are needed, a more complex mixing function should be used.

Using S-Boxes for Mixing

Many modern block encryption functions, including the Data Encryption Standard (DES) and the Advanced Encryption Standard AES, incorporate modules known as S-Boxes, meaning substitution boxes. These produce a smaller number of outputs from a larger number of inputs through a complex non-linear mixing function that has the effect of concentrating limited entropy from the inputs into the output.

S-Boxes sometimes incorporate bent boolean functions. These are functions of an even number of bits producing one output bit with maximum non-linearity. Looking at the output for all input pairs differing in any particular bit position, exactly half of the outputs are different. An S-Box in which each output bit is produced by a bent function such that any linear combination of these functions is also a bent function is called a *perfect S-Box*.

S-boxes and various repeated applications or cascades of such boxes can be used for mixing ([Mister and Adams, 1996], [Nyberg, 1991]).

Using Diffie-Hellman as a Mixing Function

Diffie-Hellman (Section 1.3) shared secret is a mixture of initial quantities generated by each of the parties involved in the communication. If these initial quantities are random and uncorrelated, then the shared secret combines their entropy but, of course, it cannot produce more randomness than the size of the shared secret generated.

However, using Diffie-Hellman as a mixing function is not recommended. Diffie-Hellman is a computationally intensive algorithm and moreover if the Diffie-Hellman computation is performed privately, an adversary who can observe either of the public keys and knows the modulus being used only needs to search through the space of the other secret key in order to be able to calculate the shared secret. So, if one wants to use this algorithm it would be best to consider public Diffie-Hellman to produce a quantity whose guessability corresponds to the worse of the two inputs.

Using a Mixing Function to Stretch Random Bits

A mixing function does not have as requirement the production of the same or fewer bits than its inputs. However, mixing bits cannot stretch the amount of the unpredictability present in the inputs, although the output can be expanded to hundreds or thousands of bits. Furthermore, mixing to fewer bits than the inputted ones will tend to strengthen the randomness of the output.

Other Factors in Choosing a Mixing Function

For local use, AES has as advantages being widely tested for flaws, being reasonably efficient in software, and being widely documented and implemented with hardware and software implementations available all over the world including open source code. The SHA* family have a little less study and tend to require more CPU cycles than AES but there is no reason to believe they are flawed. Both SHA* and MD5 were derived from the earlier MD4 ([Rivest, 1992a], [Rivest, 1992b]) algorithm and they all have source code available. Some signs of weakness have been found in MD4 and MD5. In particular, MD4 has only three rounds and there are several independent breaks of the first two or last two rounds. And some collisions have been found in MD5 output.

Where input lengths are unpredictable, hash algorithms are more convenient to use than block encryption algorithms since they are generally designed to accept variable inputs length. Block encryption algorithms generally require an additional padding algorithm to accommodate inputs that are not an even multiple of the block size.

6.7 The Blum Blum Shub Generator Example

The generator described in this section has been dubbed Blum Blum Shub generator (BBS), after its inventors: Lenore Blum, Manuel Blum and Michael Shub in 1986 ([Lenore Blum and Shub, 1986]) but it is also known as the quadratic residue generator since the theory on which this generator is based relies in the theory of the quadratic residues modulo n.

BBS Mechanism

Firstly, p and q are two primes that are found in a way that both are congruent to 3 modulo 4. $n = p * q$ is called a Blum integer. Then, another random integer, x, is chosen in a way that it has to be relatively prime to n. $x_0 = x^2 \pmod{n}$ is computed and it is going to be the seed for the Blum Blum Shub generator.

The next step is to compute the bits. The i^{th} pseudorandom bit is the least significant bit of $x_i = x_{i-1}^2 \pmod{n}$.

This generator has a very particular property, which is the no need for iteration through all the $i-1$ bits in order to get the i^{th} bit, which means that this cryptographically strong pseudorandom bit generator can be used as a stream cryptosystem for a random access file. If p and q are known, the i^{th} bit is computed directly:

$$b_i \text{ is the least significant bit of } x_i, \text{ where } x_i = x_0^{2^i \pmod{(p-1)(q-1)}}$$

BBS Security

The security of the BBS generator relies on the difficulty of factoring n (Chapter 4). Thus, n can be a public parameter and everyone can generate random numbers using the BBS generator, since no one can predict the output, unless n is factored.

More strongly, given a bit sequence generated by the BBS generator, an attacker cannot predict neither the previous bit nor the next bit: the BBS generator is unpredictable to the left and unpredictable to the right.

The BBS generator algorithm is slow, but some speedups can be added. Not only the least significant bit of x_i can be used but more than this bit can be used as a pseudorandom bit. If the length of n is equal to the length of x_i, the least significant $\log_2 n$ bits of x_i can be used ([Vazirani and Vazirani, 1984], [Vazirani and Vazirani, 1985], [W. Alexi and Schnorr, 1984], [W. Alexi and Schnorr, 1988]).

Although BBS generator is a slow generator and although it is not useful for stream ciphers, for high-security applications, like key generation, the Blum Blum Shub generator is considered to be "the best of the lot" ([Schneier, 1996]).

Chapter 7

Quality analysis of generated numbers

Nature almost surely operates by combining chance with necessity, randomness with determinism...

Eric Chaisson

The NIST package "A Statistical Test Suit for Random and Pseudorandom Number Generators for Cryptography Applications" ([Rukhin et al., 2010]) was used to test the randomness of a given generated number. The tested numbers are generated by the Luna SA hardware security module (HSM), a HSM with certifications of FIPS 140-2, Level 2 and Level 3 approved DRBG (SP 800-90 CTR mode), Common Criteria EAL4+ and BAC & EAC ePassport Support, which is mainly used for cryptographic purposes. Since NIST is responsible for specifying safety standards, and since there is a package of tests created by NIST, that aim at testing the patterns of randomness (standards set by NIST itself) of a given sequence of bits, it was decided to evaluate the randomness of the Luna SA generated numbers using this statistical tests package.

The tests were run in a virtual machine with Linux operating system, in a TOSHIBA computer with Windows 8.1 Pro 64 bits operating system, 8192Mb RAM and a Intel(R) Core(TM) i5-2520M CPU 2.50GHz (4 CPUs).

A detailed explanation about the statistical tests used in this chapter can be found in Chapter 6.

7.1 Quality analysis of a Hardware Security Module (HSM) generated number of 10^6 bits

Firstly, a request was made to the Luna SA HSM to generate 1000 $n = 1000000$ bits number. Below is presented an analysis of one of the 1000 numbers, since all of the obtained statistics were similar. In the NIST examples, that can be found in [Rukhin et al., 2010], Appendix B, the number of bits of the sequence to be tested, ϵ, is equal to 1000000. Thus, it was chosen to adopt the conditions used in [Rukhin et al., 2010] examples. It took about 7 seconds to run all the tests for each one of the generated numbers.

7.1.1 Frequency Test

The recommended length of the sequence to be tested consists of a minimum of 100 bits. Since ϵ comprises 1000000 bits, the input size recommendation is fulfilled.

The calculated statistics are represented in Table 7.1.

Computational Information		Result
The n^{th} partial sum (S_n)	74	
S_n/n	0.000074	
P-value	0.941010	Success

Table 7.1: Statistics table of the Frequency Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

P-value $= 0.941010 > 0.01$ and thus, it can be concluded that the tested sequence is considered to be random.

This test aims to evaluate the ratio of the zeros and ones in ϵ. Since the result is the acceptance of the hypothesis that dictates the sequence is random, it can be concluded that the fractions of zeros and ones are thereabout 1/2.

7.1.2 Frequency Test Within a Block

This statistical test evaluates the proportion of the existent zeros and ones in a block of M bits taken from the sequence to be tested. For the block length it was chosen $M = 128$.

The recommended length of the sequence to be tested consists of a minimum of 100 bits. Since ϵ comprises 1000000 bits, the input size recommendation is fulfilled. The condition $n \geq MN$ is satisfied, with $N = \left\lfloor \frac{n}{M} \right\rfloor = 7812$ and $M \times N = 999936$. Since n is not a multiple of 128, there will be 64 bits that will be discarded in the realization of this test.

The calculated statistics are represented Table 7.2.

Computational Information		Result
χ^2	7672.843750	
Number of substrings	7812	
Block length (M)	128	
Bits discarded	64	
P-value	0.867494	Success

Table 7.2: Statistics table of the Frequency Test Within a Block for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

P-value $= 0.867494 > 0.01$ and thus, it can be concluded that the tested sequence is considered to be random.

This result reveals that each one of the M-bits blocks of ϵ have approximately $M/2$ ones and $M/2$ zeros.

7.1.3 Runs Test

The recommended length of the sequence to be tested consists of a minimum of 100 bits. Since ϵ comprises 1000000 bits, the input size recommendation is fulfilled.

The calculated statistics are represented in Table 7.3.

Computational Information		Result
π	0.500037	
$v_n(obs)$ (Total number of runs)	499479	
$\frac{\lvert v_n(obs) - 2n\pi(1-\pi)\rvert}{2\sqrt{2n}\pi(1-\pi)}$	0.736801	
P-value	0.297414	Success

Table 7.3: Statistics table of the Runs Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

P-value $= 0.297414 > 0.01$ and thus, it can be concluded that the tested sequence is considered to be random.

The value of $v_n(obs)$ is a large value, which indicates a fast oscillation in ϵ, i.e., there exists a lot of changes between zero and one in the sequence.

7.1.4 Test for the Longest Run of Ones in a Block

The recommended length of the sequence to be tested consists of a minimum of 128 bits. Since ϵ comprises 1000000 bits, the input size recommendation is fulfilled. Depending on the chosen value for n, the value of M will be assigned according to Table 6.3.

The calculated statistics are represented in Table 7.4.

Computational Information		Result
N (number of substrings)	100	
M (Substring Length)	10000	
χ^2	0.424537	
P-value	0.998639	Success

Table 7.4: Statistics table of the Test for the Longest Run of Ones in a Block for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

P-value $= 0.998639 > 0.01$ and thus, it can be concluded that the tested sequence is considered to be random.

7.1.5 Binary Matrix Rank Test

The recommended length of the sequence to be tested consists of a minimum of 38912 bits. Since ϵ comprises 1000000 bits, the input size recommendation is fulfilled.

Computational Information			Result
Probability	P_{32}	0.288788	
	P_{31}	0.577576	
	P_{30}	0.133636	
Frequency	F_{32}	276	
	F_{31}	570	
	F_{30}	130	
Number of matrices		976	
χ^2		0.193211	
Bits discarded		576	
P-value		0.907914	Success

Table 7.5: Statistics table of the Binary Matrix Rank Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

The calculated statistics are represented in Table 7.5.

P-value $= 0.907914 > 0.01$ and thus, it can be concluded that the tested sequence is considered to be random.

7.1.6 Discrete Fourier Transform (Spectral) Test

The recommended length of the sequence to be tested consists of a minimum of 38912 bits. Since ϵ comprises 1000000 bits, the input size recommendation is fulfilled.

The calculated statistics are represented in Table 7.6.

Computational Information		Result
Percentile	94.976400	
N_l	474882	
N_o	475000	
d	-1.082842	
P-value	0.278878	Success

Table 7.6: Statistics table of the Discrete Fourier Transform (Spectral) Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

P-value $= 0.278878 > 0.01$ and thus, it can be concluded that the tested sequence is considered to be random. The fact that d has a small value reveals the existence of few peaks below T and many peaks above. With the results of this test we conclude there is no detection of periodic features in the sequence.

7.1.7 Non-overlapping Template Matching Test

The Non-overlapping Template Matching Statistic Test source code has been written to provide templates for $m = 2, 3, \ldots, 10$. In order to collect meaningful results it is recommended for m to be equal to 9 or 10. In this evaluation we chose $m = 9$, as recommended. Also, the conditions $M > 0.01 \cdot n = 10000$ and $N = \lfloor n/M \rfloor = 8$ (with $M = 125000$) are met. All the recommendations are fulfilled.

The calculated statistics are represented in Table 7.7. In this test a m-bit window is used to search for a specific m-bit pattern. Thus, as in this test $m = 9$, there will be 2^9 different patterns to be tested (except the all ones and all zeros patterns). For each test, the *P-value* is computed and, thus, there will be $2^9 - 2$ *P-values*. The sequence is split into N independent blocks. W_j represents each one of this blocks, with $j = 1, \ldots, N$, and the frequency of the pattern occurrences in each block is logged in Table 7.7.

All, except one, of the calculated *P-value*, represented in table 7.7 have values less than 0.01. For the pattern 000101101, *P-value* = 0.008724, indicating that, for the tested sequence, the pattern 000101101 occurs a different number of times than what will be expected in a truly random sequence.

By convention, if the sequence fails the test in at least one pattern, then the sequence fails the test. Thus, this sequence fails this test.

Computational Information											
$M = 125000$				$N = 8$				$m = 9$			
Frequency											
Template	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8	χ^2	*P-value*	Result
000000011	265	219	244	233	241	216	244	242	8.456924	0.390155	Success
000000101	223	223	235	266	226	254	259	251	9.104076	0.333593	Success
000000111	262	244	280	251	259	234	247	230	9.258714	0.320944	Success
000001001	256	270	251	238	239	257	264	243	6.285631	0.615271	Success
000001011	230	229	229	255	223	265	254	233	7.959116	0.437474	Success
000001101	269	240	236	229	227	217	236	242	8.600971	0.377067	Success
000001111	249	257	291	250	244	246	263	235	12.135414	0.145264	Success
⋮											
000101011	245	265	259	244	246	264	229	272	8.736545	0.365006	Success
000101101	285	237	230	266	271	215	217	233	20.461531	0.008724	FAILURE
000101111	210	259	237	239	256	239	235	257	7.961234	0.437266	Success
⋮											
111111100	224	258	258	263	277	253	244	217	12.886364	0.115820	Success
111111110	242	264	251	245	266	252	249	249	4.387602	0.820570	Success

Table 7.7: Statistics table of the Non-Overlapping Template Matching Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). The complete table is in C, table C.1

In all of the 1000 generated numbers, there were some patterns failing the test. In fact,

for some of the generated numbers, more than one pattern failed the test. The same test was evaluated, in the same conditions, for a 10^6-bits number generated by the Blum Blum Shub generator and by a random number generator with SHA-1, which are considered a secure pseudorandom generators. Those generated numbers also failed the Non-Overlapping Template Matching Test for some patterns.

7.1.8 Overlapping Template Matching Test

The recommended length of the sequence to be tested consists of at least 10^6 bits and ϵ comprises 1000000 bits, fulfilling this recommendation. As the Non-Overlapping Template Matching Test, in this test a m-bit window is used to search for a specific m-bit pattern. The recommendation for m is to be equal to 9 or to 10, and, in this evaluation, $m = 9$. In this test will be sought occurences of only one pattern, previously determined. The calculated statistics are represented in Table 7.8. The number of times that the pattern occurs in each block will be recorded (the frequency of occurences) and they will be shown in Table 7.8.

Computational Information				Result	
m (block length of ones)			9		
M (length of substring)			1032		
N (number of substrings)			968		
$\lambda[(M-m+1)/2^m]$			2		
η			1		
χ^2			6.574684		
P-value			0.254241	Success	
Frequency					
0	1	2	3	4	≥ 5
339	184	140	94	85	126

Table 7.8: Statistics table of the Overlapping Template Matching Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

P-value $= 0.254241 > 0.01$ and thus, it can be concluded that the tested sequence is considered to be random.

7.1.9 Maurer's "Universal Statistical" Test

The length of the sequence to be tested should be greater than $(Q + K) \cdot L = (1280 + 141577) \cdot 7 = 999999$. Since ϵ comprises 1000000 bits, the input size recommendation is fulfilled. Q and L are chosen according to Table 6.4 values. The sequence ϵ was partitioned into two segments: an initialization segment consisting of Q L-bit non-overlapping blocks, and a test segment consisting of K L-bit non-overlapping blocks. After this partition, one bit remained in the end of the sequence, not belonging to any block. This bit was discarded, since bits remaining at the end of the sequence that do not form a complete L-bit block are discarded. The first Q

blocks are used to initialize the test and the remaining K blocks are the test blocks. K, n, L, and Q follow the relation: $K = \lfloor n/L \rfloor - Q$.

The calculated statistics are represented in Table 7.9.

Computational Information		Result
L	7	
Q	1280	
K	141577	
sum	878177.249285	
σ	0.002768	
$variance$	3.125000	
$expectedValue$	6.196251	
Bits discarded	1	
P-value	0.017574	Success

Table 7.9: Statistics table of the Maurer's "Universal Statistical" Test for a 1000000 bits number. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

P-value $= 0.017574 > 0.01$ and thus, it can be concluded that the tested sequence is considered to be random.

7.1.10 Linear Complexity Test

The recommended length of the sequence to be tested consists of a minimum of 10^6 bits. Since ϵ comprises 1000000 bits, the input size recommendation is fulfilled. M and N should be chosen by preserving the conditions: $500 \leq M \leq 5000$ and $N \geq 200$. In this evaluation, M was chosen to be equal to 500 and N was processed by the test code. The values $T_i = (-1)^M \cdot (L_i - \mu) + \frac{2}{9}$ are recorded in $v_0, v_1, \ldots, v_6$, as described in Table 6.5.

The calculated statistics are represented in Table 7.10.

Computational Information				Result		
M (substring length)			500			
N (number of substrings)			2000			
χ^2			3.289781			
Bits Discarded			0			
P-value			0.771695	Success		
Frequency						
v_0	v_1	v_2	v_3	v_4	v_5	v_6
20	56	230	1027	500	128	39

Table 7.10: Statistics table of the Linear Complexity Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

P-value $= 0.771695 > 0.01$ and thus, it can be concluded that the tested sequence is considered to be random.

7.1.11 Serial Test

For this test it is recommended to choose m and n such that $m < \lfloor \log_2 n \rfloor - 2$. Since ϵ comprises 1000000 bits and m was chosen to be equal to 16 (and $16 < \lfloor \log_2 n \rfloor - 2 = 17$), the input size recommendation is fulfilled.

The calculated statistics are represented in Table 7.11.

Computational Information		Result
Block length (m)	16	
ψ_m	66451.861504	
ψ_{m-1}	33252.438016	
ψ_{m-2}	16755.257344	
P-value1	0.046429	Success
P-value2	0.040018	Success

Table 7.11: Statistics table of the Serial Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

P-value1 $= 0.046429 > 0.01$ and *P-value2* $= 0.040018 > 0.01$. Thus, it can be concluded that the tested sequence is considered to be random.

7.1.12 Approximate Entropy Test

For this test the recommendation is to choose m and n such that $m < \lfloor \log_2 n \rfloor - 5$. Since ϵ comprises 1000000 bits and m was chosen to be equal to 10 (and $10 < \lfloor \log_2 n \rfloor - 5 = 14$), the input size recommendation is fulfilled.

The calculated statistics are represented in Table 7.12.

Computational Information		Result
Block length (m)	10	
χ^2	1069.040430	
ϕ_m	-6.930922	
ϕ_{m+1}	-7.623534	
$ApEn(m) = \varphi^{(m)} - \varphi^{(m+1)}$	0.692613	
P-value	0.159695	Success

Table 7.12: Statistics table of the Approximate Entropy Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

P-value $= 0.159695 > 0.01$. Thus, it can be concluded that the tested sequence is considered to be random. $ApEn(m)$ has a small value and this implies a strong regularity in ϵ.

7.1.13 Cumulative Sums Test

The recommended length of the sequence to be tested consists of a minimum of 100 bits. Since ϵ comprises 1000000 bits, the input size recommendation is fulfilled.

The calculated statistics for the Cumulative Sums Test Mode = 0 are represented in Table 7.13 and the calculated statistics for the Cumulative Sums Test Mode = 1 are represented in Table 7.14.

Computational Information		Result
Maximum partial sum	729	
P-value	0.875052	Success

Table 7.13: Statistics table of the Cumulative Sums (forward) Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

From Table 7.13, *P-value* $= 0.875052 > 0.01$. Thus, it can be concluded the tested sequence is considered to be random.

Computational Information		Result
Maximum partial sum	655	
P-value	0.928211	Success

Table 7.14: Statistics table of the Cumulative Sums (reverse) Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

From Table 7.14, *P-value* $= 0.928211 > 0.01$. Thus, it can be concluded the tested sequence is considered to be random.

7.1.14 Random Excursions Test

The recommended length of the sequence to be tested consists of a minimum of 1000000 bits. Since ϵ comprises 1000000 bits, the input size recommendation is fulfilled.

The calculated statistics are represented in Table 7.15.

The *P-values* shown in Table 7.15 are greater than 0.01. Thus, the tested sequence is considered to be random.

7.1.15 Random Excursions Variant Test

The recommended length of the sequence to be tested consists of a minimum of 1000000 bits. Since ϵ comprises 1000000 bits, the input size recommendation is fulfilled.

The calculated statistics are presented in Table 7.16.

Computational Information			*P-value*	Result
$x = -4$	χ^2	5.522087	0.355530	Success
$x = -3$	χ^2	4.563041	0.471493	Success
$x = -2$	χ^2	4.046021	0.542809	Success
$x = -1$	χ^2	2.469042	0.781150	Success
$x = 1$	χ^2	12.774385	0.025587	Success
$x = 2$	χ^2	14.159468	0.014628	Success
$x = 3$	χ^2	6.182046	0.288906	Success
$x = 4$	χ^2	3.477637	0.626774	Success

Table 7.15: Statistics table of the Random Excursions Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

The *P-values* shown in Table 7.16 are greater than 0.01. Thus, the tested sequence is considered to be random.

7.2 Quality analysis of a Hardware Security Module (HSM) generated number of 10^9 bits

The next step before testing a 10^6 bits number (Section 7.1) was running the statistical tests with a bigger number. Thus, it was requested to the Luna SA HSM a generation of a $n = 10^9$ bits number. The focus of this analysis was to test whether the statistical data previously obtained are consistent. It was not possible to run the Discrete Fourier Transform (Spectral) Test, the Random Excursions Test and the Random Excursions Variant Test with such a large number, because of an insufficient memory error. It took about 12 hours to run all the tests.

7.2.1 Frequency Test

Since ϵ comprises 10^9 bits, the input size recommendation is fulfilled.
The calculated statistics are presented in Table 7.17.

Computational Information			*P-value*	Result
$x = -9$	Total visits	1134	0.822167	Success
$x = -8$	Total visits	1078	0.591242	Success
$x = -7$	Total visits	986	0.270316	Success
$x = -6$	Total visits	1015	0.308536	Success
$x = -5$	Total visits	1042	0.346996	Success
$x = -4$	Total visits	1004	0.173158	Success
$x = -3$	Total visits	1019	0.140604	Success
$x = -2$	Total visits	1041	0.100846	Success
$x = -1$	Total visits	1112	0.167661	Success
$x = 1$	Total visits	1103	0.117560	Success
$x = 2$	Total visits	985	0.021078	Success
$x = 3$	Total visits	996	0.091918	Success
$x = 4$	Total visits	1014	0.199041	Success
$x = 5$	Total visits	944	0.106712	Success
$x = 6$	Total visits	937	0.132939	Success
$x = 7$	Total visits	960	0.210994	Success
$x = 8$	Total visits	954	0.231553	Success
$x = 9$	Total visits	933	0.219192	Success

Table 7.16: Statistics table of the Random Excursions Variant Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

Computational Information		Result
The n^{th} partial sum (S_n)	8364	
S_n/n	0.000008	
P-value	0.791400	Success

Table 7.17: Statistics table of the Frequency Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

P-value $= 0.791400 > 0.01$ and thus, the tested sequence is considered to be random.

7.2.2 Frequency Test Within a Block

The block length of this test is $M = 128$.

ϵ comprises 10^9 bits and thus the input size recommendation is fulfilled. The condition $n \geq MN$ is satisfied, with $N = \left\lfloor \frac{n}{M} \right\rfloor = 7812500$ and $M \times N = 10^9$. Since n is a multiple of 128, there wouldn't be any discarded bits in the realization of this test.

The calculated statistics are presented in Table 7.18.

P-value $= 0.320670 > 0.01$ and thus, the tested sequence is considered to be random. This result reveals that each one of the M-bits block of ϵ have approximately $M/2$ ones and $M/2$

Computational Information		Result
χ^2	7814340.812500	
Number of substrings	7812500	
Block length (M)	128	
Bits discarded	0	
P-value	0.320670	Success

Table 7.18: Statistics table of the Frequency Test Within a Block for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

zeros.

7.2.3 Runs Test

ϵ comprises 10^9 bits and thus the input size recommendation is fulfilled.

The calculated statistics are presented in Table 7.19.

Computational Information		Result
π	0.500004	
$v_n(obs)$ (Total number of runs)	499962850	
$\frac{\lvert v_n(obs) - 2n\pi(1-\pi)\rvert}{2\sqrt{2n}\pi(1-\pi)}$	1.661397	
P-value	0.018795	Success

Table 7.19: Statistics table of the Runs Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

P-value $= 0.018795 > 0.01$ and thus, the tested sequence is considered to be random.

7.2.4 Test for the Longest Run of Ones in a Block

Since ϵ comprises 10^9 bits, the input size recommendation is fulfilled and, depending on the chosen value of n, the value of M will be assigned according to Table 6.3.

The calculated statistics are presented in Table 7.20.

P-value $= 0.480659 > 0.01$ and thus, the tested sequence is considered to be random.

7.2.5 Binary Matrix Rank Test

Since ϵ comprises 10^9 bits, the input size recommendation is fulfilled.

The calculated statistics are presented in Table 7.21.

Computational Information		Result
N (number of substrings)	100000	
M (Substring Length)	10000	
χ^2	5.506601	
P-value	0.480659	Success

Table 7.20: Statistics table of the Test for the Longest Run of Ones in a Block for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

Computational Information			Result
Probability	P_{32}	0.288788	
	P_{31}	0.577576	
	P_{30}	0.133636	
Frequency	F_{32}	281676	
	F_{31}	564029	
	F_{30}	130857	
Number of matrices		976562	
χ^2		1.375720	
Bits discarded		512	
P-value		0.502651	Success

Table 7.21: Statistics table of the Binary Matrix Rank Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

P-value $= 0.907914 > 0.01$ and thus, the tested sequence is considered to be random.

7.2.6 Non-overlapping Template Matching Test

In this evaluation $m = 9$, as recommended. Also, the conditions $M > 0.01 \cdot n = 10^9$ and $N = \lfloor n/M \rfloor = 8$ (with $M = 125000$) are met. All the recommendations are fulfilled.

The calculated statistics are presented in Table 7.22. In this test $m = 9$ and thus there will be 2^9 different patterns (except the all ones and all zeros patterns) to be tested. This means that there will be $2^9 - 2$ *P-values*. The sequence is split into N independent blocks and the frequency of the pattern occurrences in each block is logged in Table 7.22.

There are more than one calculated *P-value*, presented in Table 7.22, that have values less than 0.01. For the patterns 000001001, 011101111 and 101000100 the *P-values* are 0.007990, 0.008124 and 0.001514, respectively. This indicates that, for the tested sequence, the patterns mentioned before occur a different number of times from what will be expected in a truly random sequence.

By convention, if the sequence fails the test in at least one pattern, then the sequence fails the test. Thus, this sequence fails this test and it is not considered to be random.

Computational Information											
$M = 125000000$				$N = 8$				$m = 9$			
Frequency											
Template	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8	χ^2	*P-value*	Result
000000001	244453	243866	244528	243860	243632	244292	244701	244669	5.408738	0.713129	Success
000000011	244246	244208	244181	244114	243820	243561	244738	243642	4.500244	0.809409	Success
000000101	243851	244593	244777	243523	244060	244104	243564	244391	6.261703	0.617943	Success
000000111	243702	243982	244577	243827	244504	244555	244255	244321	3.625408	0.889242	Success
000001001	244032	243022	244004	244622	244548	243187	245637	244381	20.699494	0.007990	FAILURE
⋮											
011010111	244024	244098	243871	243788	244538	244715	244831	243603	6.210717	0.623641	Success
011011111	244361	243475	244224	244784	244656	244316	243641	244265	6.244790	0.619833	Success
011101111	244355	244085	244510	243922	243593	244657	246004	243364	20.654521	0.008124	FAILURE
011111111	243889	243960	244026	244166	244974	244945	245145	244106	10.427673	0.236282	Success
⋮											
101000000	243727	243152	244648	243802	244232	243868	244753	243636	9.459824	0.305001	Success
101000100	244833	243380	243710	244680	245369	243892	245194	242836	25.067049	0.001514	FAILURE
101001000	243399	244164	244159	244215	243963	244509	243992	244634	4.190828	0.839509	Success
⋮											
111111100	243873	244625	244474	244260	244530	244650	243662	244193	4.552591	0.804149	Success
111111110	243889	243960	244026	244166	244974	244945	245145	244106	10.427673	0.236282	Success

Table 7.22: Statistics table of the Non-Overlapping Template Matching Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random). The complete table is in C, table C.2

7.2.7 Overlapping Template Matching Test

ϵ comprises 10^9 bits and thus the input size recommendation is fulfilled. In this evaluation, $m = 9$.

The calculated statistics are presented in Table 7.23.

P-value $= 0.00000 < 0.01$ and thus, the tested sequence is not considered to be random.

7.2.8 Maurer's "Universal Statistical" Test

Since ϵ comprises 10^9 bits, the input size recommendation is fulfilled. Q and L are chosen according to Table 6.4. The sequence ϵ was partitioned into two segments: an initialization segment consisting of Q L-bit non-overlapping blocks, and a test segment consisting of K L-bit non-overlapping blocks. After this partition, 10 bits remained in the end of the sequence, not belonging to any block. Those 10 bits were discard.

The calculated statistics are presented in Table 7.24.

Computational Information				Result	
m (block length of ones)			9		
M (length of substring)			1032		
N (number of substrings)			968992		
$\lambda[(M-m+1)/2^m]$			2		
η			1		
χ^2			116.476980		
{P-value}			0.000000	FAILURE	
Frequency					
0	1	2	3	4	≥ 5
351982	180536	135143	97572	68021	135738

Table 7.23: Statistics table of the Overlapping Template Matching Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

Computational Information		Result
L	15	
Q	327680	
K	66338986	
sum	939865144.454580	
σ	0.000149	
$variance$	3.419000	
$expectedValue$	14.167488	
Bits discarded	10	
P-value	0.398955	Success

Table 7.24: Statistics table of the Maurer's "Universal Statistical" Test for a 10^9 bits number. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

P-value $= 0.398955 > 0.01$ and thus, the tested sequence is considered to be random.

7.2.9 Linear Complexity Test

ϵ comprises 10^9 bits and thus, the input size recommendation is fulfilled. In this evaluation, M was chosen to be equal to 500 and N was processed by the test source code. The values $T_i = (-1)^M \cdot (L_i - \mu) + \frac{2}{9}$ are recorded in $v_0, v_1, \ldots, v_6$, as presented in Table 6.5.

The calculated statistics are presented in Table 7.25.

P-value $= 0.520163 > 0.01$ and thus, the tested sequence is considered to be random.

7.2.10 Serial Test

ϵ comprises 10^9 bits and m was chosen to be equal to 16 (and $16 < \lfloor \log_2 n \rfloor - 2 = 27$) and thus the input size recommendation is fulfilled.

Computational Information				Result		
M (substring length)			500			
N (number of substrings)			2000000			
χ^2			5.186203			
Bits Discarded			0			
P-value			0.520163	Success		
Frequency						
v_0	v_1	v_2	v_3	v_4	v_5	v_6
20888	62344	249283	1001018	499452	125349	41666

Table 7.25: Statistics table of the Linear Complexity Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

The calculated statistics are presented in Table 7.26.

Computational Information		Result
Block length (m)	16	
ψ_m	66755.999433	
ψ_{m-1}	33531.165344	
ψ_{m-2}	16705.072693	
P-value1	0.037629	Success
P-value2	0.466093	Success

Table 7.26: Statistics table of the Serial Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

P-value1 $= 0.037629 > 0.01$ and *P-value2* $= 0.466093 > 0.01$. Thus, the tested sequence is considered to be random.

7.2.11 Approximate Entropy Test

For this test ϵ comprises 10^9 bits and m was chosen to be equal to 10 (and $10 < \lfloor \log_2 n \rfloor - 5 = 24$). So, the input size recommendation is fulfilled.

The calculated statistics are presented in Table 7.27.

Computational Information		Result
Block length (m)	10	
χ^2	1002.410764	
ϕ_m	-6.931471	
ϕ_{m+1}	-7.624618	
$ApEn(m) = \varphi^{(m)} - \varphi^{(m+1)}$	0.693147	
P-value	0.679283	Success

Table 7.27: Statistics table of the Approximate Entropy Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

P-value $= 0.679283 > 0.01$. Thus, it can be concluded that the tested sequence is considered to be random. $ApEn(m)$ has a small value, and this implies a strong regularity in ϵ.

7.2.12 Cumulative Sums Test

ϵ comprises 10^9 bits and thus, the input size recommendation is fulfilled.

The calculated statistics for the Cumulative Sums Test Mode = 0 are presented in Table 7.28 and the calculated statistics for the Cumulative Sums Test Mode = 1 are presented in Table 7.29.

From table 7.28, *P-value* $= 0.842729 > 0.01$. Thus, the tested sequence is considered to be random.

Computational Information		Result
Maximum partial sum	24288	
P-value	0.842729	Success

Table 7.28: Statistics table of the Cumulative Sums (forward) Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

Computational Information		Result
Maximum partial sum	24606	
P-value	0.834051	Success

Table 7.29: Statistics table of the Cumulative Sums (reverse) Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hypothesis (the hypothesis that states the sequence is random).

From table 7.29, $P\text{-}value = 0.834051 > 0.01$. Thus, the tested sequence is considered to be random.

7.3 Quality analysis of a Hardware Security Module (HSM) set of generated numbers

On the individual tests, the acceptance level about the hypothesis "the generated sequence is random" is quite high, since the tested sequences pass most of the tests. However, this does not imply a high pass rate of the test suit. NIST suggests to run the tests multiple times. The number of times the tests should be run is at least the inverse of the significance level, which, in this case, is 0.01. Thus, to check the pass rate, the tests were performed 100 times. Although the sequences failed some of the tests, the failure sample is not significant to deemed this generator to a bad random generator. The tests results can be seen in Table 7.30.

7.4 Conclusions

The HSM Luna SA generated numbers are considered to be random, since they passed almost all of the tests.

The tests results induce to the conclusion that if the number is getting bigger, the probabilities of being "less random" are also bigger. In the Non-overlapping Template Matching Test one can check that some patterns occur a number of times than what would be expected in a truly random sequence. Intuitively, if the tested number gets bigger, the pattern has more probability to occur. Further, in the Overlapping Template Matching Test for a 10^6 bits number, the conclusion is that the number passed the test. However, in the same test for a 10^9 bits number, the conclusion is that the number did not pass the test. The patterns searched in the tested numbers occur a larger number of times than the number of times supposed to occur in a truly random number.

One must notice that, despite the generated numbers from HSM Luna SA had fail in two tests, they had success in the others 13 tests.

Statistical Test	Proportion
Frequency Test	98/100
Frequency Test Within a Block	100/100
Runs Test	100/100
Test for the Longest Runs of Ones in a Block	100/100
Binary Matrix Rank Test	98/100
Discrete Fourier Transform (Spectral) Test	98/100
Non-overlapping Template Matching Test	96/100
Overlapping Template Matching Test	97/100
Maurer's "Universal Statistical" Test	100/100
Linear Complexity Test	97/100
Serial Test (1)	99/100
Serial Test (2)	99/100
Approximate Entropy Test	98/100
Cumulative Sums Test (mode = 0)	99/100
Cumulative Sums Test (mode = 1)	98/100
Random Excursions Test (1)	98/100
Random Excursions Test (2)	99/100
Random Excursions Test (3)	100/100
Random Excursions Test (4)	100/100
Random Excursions Test (5)	99/100
Random Excursions Test (6)	99/100
Random Excursions Test (7)	97/100
Random Excursions Test (8)	99/100
Random Excursions Variant Test (1)	99/100
Random Excursions Variant Test (2)	99/100
Random Excursions Variant Test (3)	99/100
Random Excursions Variant Test (4)	99/100
Random Excursions Variant Test (5)	100/100
Random Excursions Variant Test (6)	100/100
Random Excursions Variant Test (7)	99/100
Random Excursions Variant Test (8)	100/100
Random Excursions Variant Test (9)	99/100
Random Excursions Variant Test (10)	99/100
Random Excursions Variant Test (11)	99/100
Random Excursions Variant Test (12)	99/100
Random Excursions Variant Test (13)	99/100
Random Excursions Variant Test (14)	99/100
Random Excursions Variant Test (15)	100/100
Random Excursions Variant Test (16)	100/100
Random Excursions Variant Test (17)	100/100
Random Excursions Variant Test (18)	100/100

Table 7.30: Pass rates test for each one of the NIST Statistical Test Suit [Rukhin et al., 2010] based on 100 tests for 100 10^6 bits number generated by a Luna SA HSM.

Chapter 8

MQualityTester Application

One of the proposed objectives, in the context of the developed work in MULTICERT, was to develop a Java application whose functionality would be testing the quality of a RSA keys set and the randomness quality of one or several binary sequences generated by an Hardware Security Module (HSM). The Java application, named *MQualityTester*, should be able to analyze the quality of a RSA keys set obtained through a direct request for keys to a HSM, it should be able to analyse a RSA keys set obtained by reading a text file given as input, as well as it should be able to analyze the randomness quality of binary sequences obtained through a direct request to a HSM or binary sequences obtained by reading a text file given as input.

Testing the quality of a RSA keys set, in this application, consists only on a single test that checks for common factors in the various modules of RSA public keys to be tested. This single test was chosen in accordance with the last successfully performed attack on RSA cryptosystem (Section 5). If the tested RSA keys set passes the test, nothing can be concluded about those keys quality. However, if the RSA keys set does not pass this test, it may be said that the tested keys do not have the necessary requirements to be considered as secure keys.

Testing randomness quality of one binary sequence or a set of binary sequences, in the *MQualityTester* application, consists in running a set of 15 statistical tests provided by National Institute of Standards and Technology (NIST) ([Rukhin et al., 2010]). The detailed description of these tests can be found in Chapter 6.

Several images are presented below illustrating the appearance of the *MQualityTester* application interface, followed by a short description about the application's features. The application was developed in Portuguese, our mother language, and this is the reason why the menus and all the descriptions in the images below are written in Portuguese.

The interface has a simple appearance, as illustrated in Figure 8.1, allowing the user to experience its features in an intuitive way. The application has a console area, in which the state of the program will be updated in real-time as the program runs, as well as three tabs:

- *"Teste de qualidade de chaves":* The user should choose this tab if he/she wants to test the quality of a RSA keys set (Figure 8.2 and Figure 8.3). After selecting this tab, the user can choose to generate the keys directly in the HSM, as illustrated in Figure 8.2, or he/she can input a file with the previously generated RSA keys set, as illustrated in Figure

8.3. Since the HSM used along the MULTICERT internship, Luna SA – Network-Attached HSM, only generates 2048-bit RSA keys, this is the default size in the *MQualityTester*. If the user chooses to generate the RSA keys directly on the HSM, he/she must enter the number of keys to be generated and he/she may or may not choose to save the extracted modules of the tested keys for further analysis. The *MQualityTester* is confined to generate keys in the Luna SA – Network-Attached HSM, since the connection to it was developed using the Key Gen Server module already existing in MULTICERT repository. If the user chooses to input a file with the previously generated RSA keys set, he/she should enter the path to it, and he/she may or may not choose to save the extracted modules of the tested keys. The test results are automatically saved in the folder *results* that comes with the *MQualityTester* application. In the end of the test execution, the *MQualityTester* launches a notification about the test result, as illustrated in Figure 8.7.

- *"Teste de aleatoriedade":* The user should choose this tab if he/she wants to test the randomness quality of one or more binary sequences (Figure 8.4 and Figure 8.5). After selecting this tab, the user can choose to generate the binary sequences directly in an HSM, as illustrated in Figure 8.4, or he/she can input a file with the binary sequences previously generated, as illustrated in Figure 8.5. If the user chooses to generate the binary sequences directly on the HSM, he/she must enter the number of sequences to be generated and the number of bits each one should have. The connection is made directly to the HSM, and the HSM in use can be configured in the tab *"Configurações"* (Figure 8.6). The user may or may not choose to save the generated sequences for further analysis. If the user chooses to input a file with the data to be tested, he/she should enter the path to the desired file, the number of sequences to be tested and their number of bits.

  The NIST Statistical Test Suit that is intended to be used is also configurable on tab *"Configurações"* (Figure 8.6) and the results are stored in the NIST application default path intended to save the tests results.

  At the end of the tests, the *MQualityTester* launches a notification about the summarized outcome of the tests, as illustrated in Figure 8.8.

- *"Configurações":* In this tab, illustrated in Figure 8.6, the user can configure the HSM to which the application will connect if he/she chooses to test randomness quality of binary sequences. For this test, the user must also enter the location path of the NIST Statistical Test Suit and the results of these tests will be automatically stored in the target path set by NIST's application.

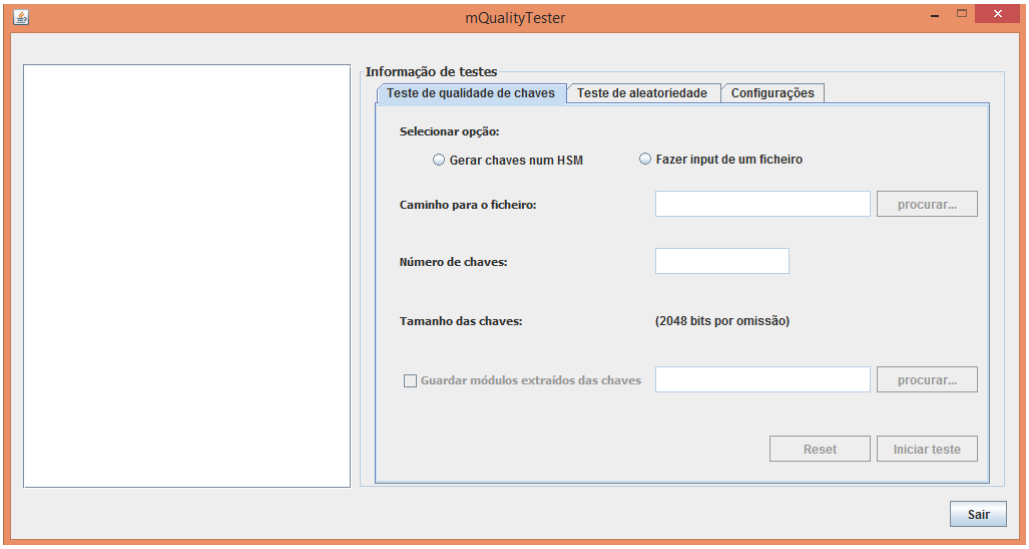


Figure 8.1: *GUI general appearance.*

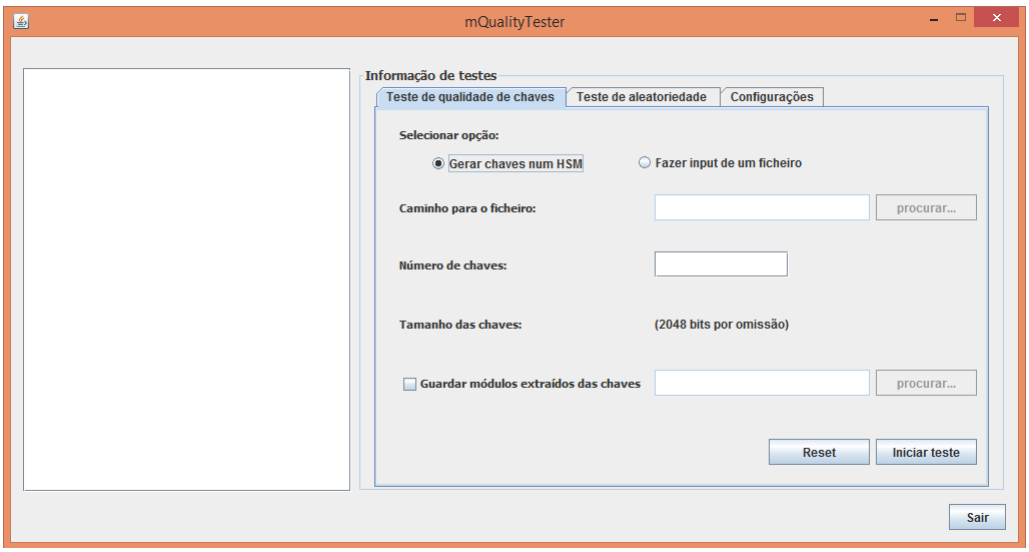


Figure 8.2: *When choosing the tab* "Teste de qualidade de chaves", *the user can choose to generate the RSA keys directly from an HSM.*

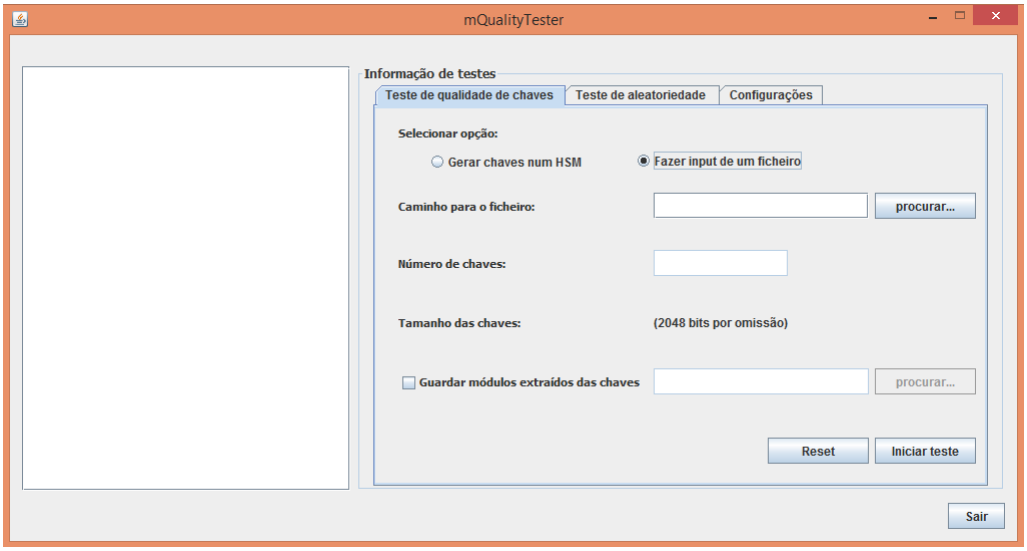


Figure 8.3: *When choosing the tab* "Teste de qualidade de chaves", *the user can choose to give the RSA keys to be tested as an input.*

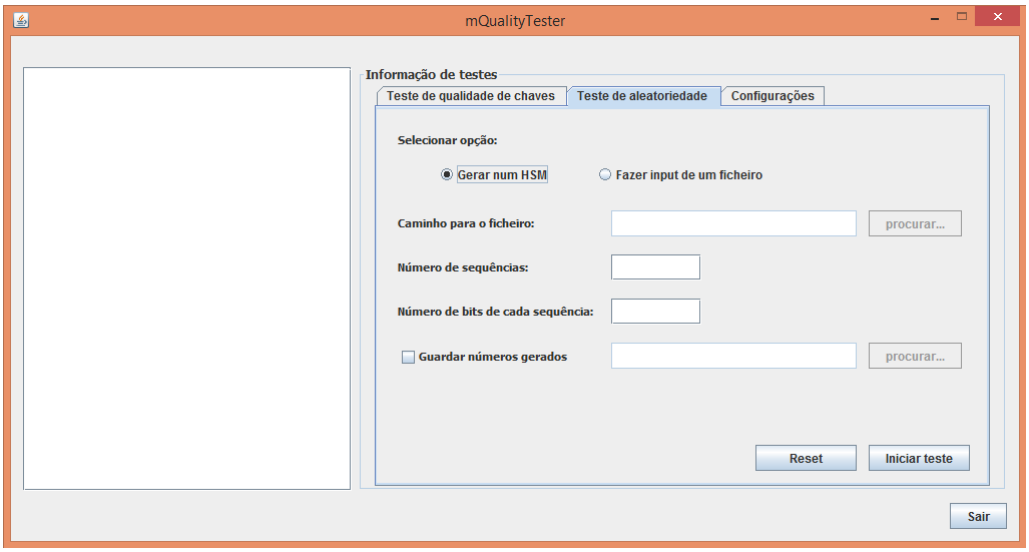


Figure 8.4: *When choosing the tab* "Teste de aleatoriedade", *the user can choose to generate the binary sequences directly from an HSM.*

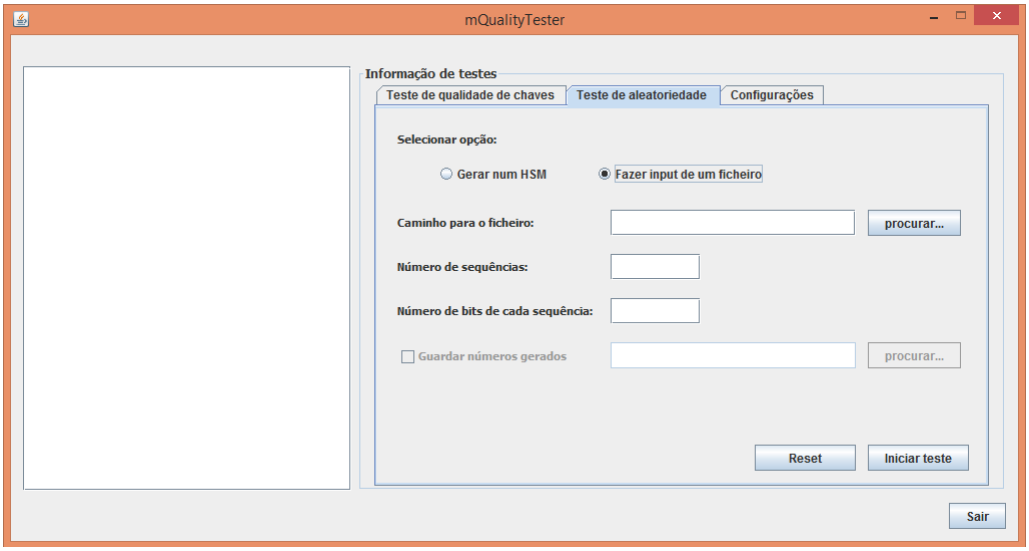


Figure 8.5: *When choosing the tab* "Teste de aleatoriedade", *the user can choose to give the binary sequences to be tested as an input.*

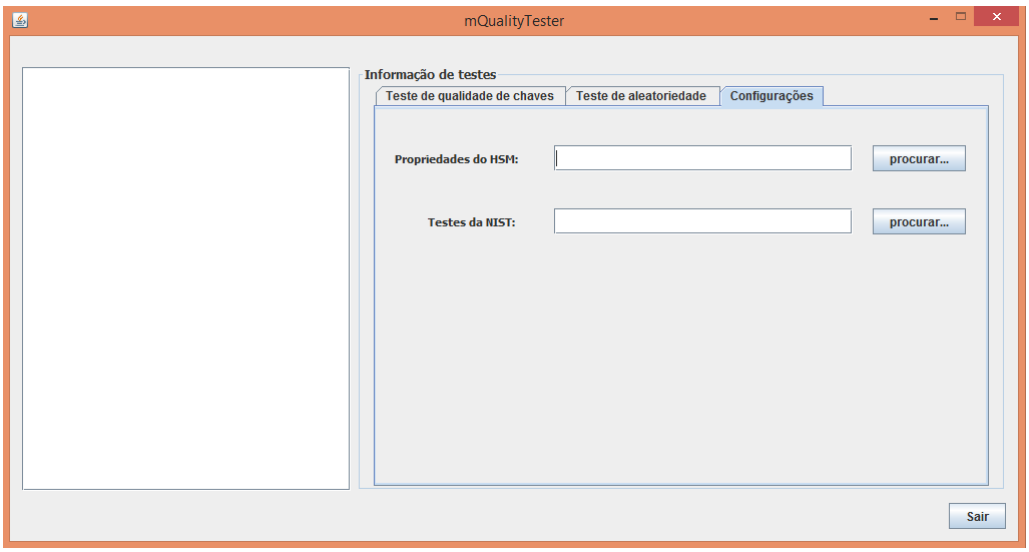


Figure 8.6: *Functional model of an RBG that uses a DRBG mechanism, includes a source of entropy input and, depending on the implementation of the DRBG mechanism, it can include a nonce source.*

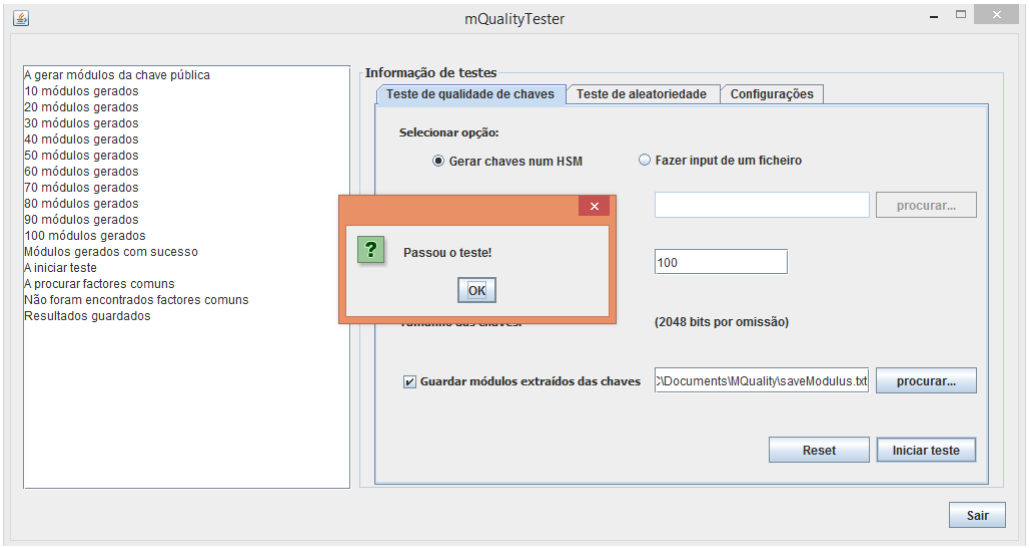


Figure 8.7: *Example of the window shown at the end of the keys quality test with the test results resume. In this example, the user requested Luna SA – Network-Attached HSM 100 2048-bit RSA keys and they passed the test.*

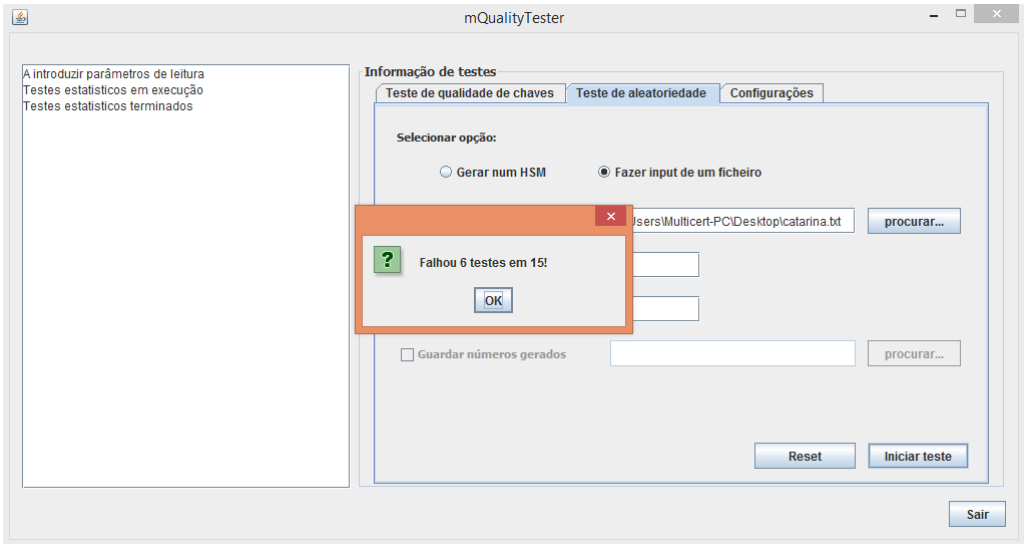


Figure 8.8: *Example of the window shown at the end of the randomness quality test with the test results resume. In this example, the user inputed one file containing the binary sequences to be tested and they passed 6 tests in a total of 15 tests.*

Chapter 9

Conclusions and Future Work

9.1 Conclusions

As said before, cryptography is the basis of security mechanisms of all computer systems and is the basis of the security provided to each user when accessing secure websites, by preventing the information from being intercepted by third parties. In fact, modern cryptography is becoming a more and more central topic in computer science.

The RSA cryptosystem is the most widely used cryptosystem, and, in fact many processes rely their information security on the RSA efficiency. Although RSA has suffered many attacks (either innocuous attacks, which are intended to only test the system safety or malicious attacks intended to access secret information), its vulnerabilities have been well studied and the algorithm was improved, so that this cryptosystem is considered to be an efficient and secure algorithm. The technological progress allows an improvement of the algorithm, such as, for example, large keys sizes being more accessible to use. There are also many security protocols and principles approved by the information security specialized agencies, in order to avoid bad practices of this cryptographic system.

In this work, the lastest successful approach made on some RSA keys vulnerabilities, in order to find some RSA modulus factors, was explored in some detail (Chapter 5). This approach was implemented in Java 8 and it was incorporated into the developed application, named *MQualityTester* (Chapter 8), as a quality test for a set of RSA keys. Several sets of 2048-bits RSA keys generated in Luna SA - network-attached HSM were tested and none of those sets has revealed to have common factors in their public key modules, and thus, all the RSA keys sets passed the test.

Since the RSA cryptosystem is based on the generation of pseudorandom numbers, it was decided to study their generation, and to meet the necessary criteria for a binary sequence considered to be random, or more specifically, pseudorandom. The tests used for the randomness quality analysis were the tests provided by NIST, which include several of the properties that must be met for a binary sequence to be considered pseudorandom. These tests were also incorporated into the *MQualityTester* application. Those statistical tests were performed on various binary sequences generated by the Luna SA - network-attached HSM. A large proportion of the tested sequences did not pass the Non-Overlapping Template Matching Test (Section

7.1.7). However, in the NIST documentation it becomes clear that a sequence does not need to pass all statistical tests to be considered pseudorandom sequences. Indeed, several sets of binary sequences generated by the Blum Blum Shub Generator (well known from the literature to be a good generator) were also tested, and the vast majority of the sequences did not pass this same test.

All tests conducted on elements generated by Luna SA - network-attached HSM showed positive results, not revealing anomalies in the generation of RSA keys or pseudorandom sequences.

The *MQualityTester* application was developed with success, showing reliability in the results, as well as versatility to be adaptable to new projects. Its graphical user interface was developed considering its user-friendly capacity, so that any user would be able to make use of their characteristics in an intuitively way.

9.2 Future Work

There are numerous approaches for the RSA cryptanalysis. In this thesis many of its vulnerabilities were studied but only one of them was implemented. It would be beneficial to explore new attacks and incorporate them into the *MqualityTester* so it can provide more and better information about the quality of the RSA keys generated by an HSM.

Although the user can configure the NIST Statistical Test Suit intended to be used, which is an asset in case of NIST updating the tests, it would be interesting if the statistical tests were built-in the application. To this purpose further study of the statistical tests mechanism would be necessary.

The analysis of the randomness and RSA keys quality generated by a HSM should be done over a time line to see if there are some changes in the numbers generation. And in case there are, the temporal analysis can help uncover the problem source or the decision to reevaluate the system.

Appendix A

Coppersmith's Theorem

The most powerful attacks on low public exponent RSA are based on a theorem due to Coppersmith [Coppersmith, 1997] which has many applications.

Theorem 14. [Coppersmith's Theorem]

Let n be an integer and $f \in \mathbb{Z}[x]$ be a monic polynomial of degree w over the integers. Set $x = n^{\frac{1}{w}-\epsilon}$ for $\frac{1}{w} > \epsilon \geq 0$. Then, given (n, f) an attacker can efficiently find all integers $|x_0| < x$ satisfying $f(x_0) = 0 \pmod{n}$. The running time is dominated by the time it takes to run the LLL algorithm on a lattice of dimension $\mathcal{O}(\omega)$ with $\omega = \min\left(\frac{1}{\epsilon}, \log_2(n)\right)$.

The theorem stated above provides an algorithm for efficiently finding all roots of a function f modulo n that are less than $x = n^{\frac{1}{w}}$. As x gets smaller, the algorithm's running time decreases. The theorem's strength is its ability to find small roots of polynomials modulo a composite n. When working modulo a prime, there is no reason to use Coppersmith's theorem since other, far better, root-finding algorithms exist.

For the proof of Coppersmith's theorem it will be used LLL lattice basis reduction algorithm as well as a simplified approach due to Howgrave-Graham [Howgrave-Graham, 1998]. Let us define $||h||^2 = \sum_i |a_i|^2$ from a given a polynomial $h(x) = \sum a_i x^i \in \mathbb{Z}[x]$.

Lemma 2. *Let $h(x) \in \mathbb{Z}[x]$ be a polynomial of degree w and let x be a positive integer. Suppose $||h(kx)|| < \frac{n}{\sqrt{w}}$. If $||x_0|| < x$ satisfies $h(x_0) = 0 \pmod{n}$, then $h(x_0) = 0$ holds over the integers.*

Proof. From the Schwarz inequality,

$$\begin{aligned} |h(x_0)| &= \left|\sum a_i x_0^i\right| = \left|\sum a_i x^i \left(\frac{x_0}{x}\right)^i\right| \\ &\leq \sum \left|a_i x^i \left(\frac{x_0}{x}\right)^i\right| \\ &\leq \sum |a_i x^i| \\ &\leq \sqrt{w}||h(kx)|| \\ &< n \end{aligned}$$

Thus, $h(x_0) = 0$, since $h(x_0) = 0 \pmod{n}$.

□

This lemma states that if h is a polynomial with a low norm, then all small roots of $h \pmod{n}$ are also roots of h over the integers, suggesting that to find a small root x_0 of $f(x) \pmod{n}$ one should look for another polynomial $h \in \mathbb{Z}[x]$ with small norm having the same roots as f modulo n. Then x_0 will be a root of h over the integers and can be easily found. To do so, one may search for a polynomial $g \in \mathbb{Z}[x]$ such that $h = gf$ has low norm (and for low norm it means having a norm less than n). This amounts to searching for an integer linear combination of the polynomials $f, xf, x^2f, ..., x^rf$ with low norm. Unfortunately, most often there is no nontrivial linear combination with sufficiently small norm.

Coppersmith found a way to solve this problem: if $f(x_0) = 0 \pmod{n}$, then $f(x_0)^k = 0 \pmod{n^k}$, for any k. More generally, define the following polynomials:

$$g_{u,v}(x) = n^{m-v}x^u f(x)^v$$

for some predefined m. Then x_0 is a root of $g_{u,v}(x)$ modulo n^m for any $u \geq 0$ and $0 \leq v \leq m$. To use Lemma 2 one must find an integer linear combination $h(x)$ of the polynomials $g_{u,v}(x)$ such that $h(kx)$ has norm less than n^m (recall that x is an upper bound on x_0 satisfying $x \leq n^{\frac{1}{w}}$). Thanks to the relaxed upper bound on the norm (n^m rather than n), one can show that for sufficiently large m, there always exists a linear combination $h(x)$ satisfying the required bound. Once $h(x)$ is found, Lemma 2 implies that it has x_0 as a root over the integers. Consequently x_0 can be easily found.

To prove Coppersmith's theorem it is used the LLL algorithm. To ensure that the vector produced by LLL satisfies the bound of Lemma 2 the following discriminat must be assured:

$$2^{\frac{j}{4}} det(L)^{\frac{1}{j}} < \frac{n^m}{\sqrt{j}}$$

where $j = w(m+1)$ is the dimension of L. A routine calculation shows that for large enough m the bound is satisfied [Boneh, 1999]. Indeed, when $x = n^{\frac{1}{w}-\epsilon}$, it suffices to take $m = \mathcal{O}(\frac{l}{w})$ with $l = \min\left(\frac{1}{\text{ffl}}, \log(\text{n})\right)$. Consequently, the running time is dominated by running LLL on a lattice of dimension $\mathcal{O}(l)$, as required.

A natural question is whether Coppersmith's theorem can be applied to bivariate and multivariate polynomials. If $f(x, y) \in \mathbb{Z}_n[x, y]$ is given for which there exists a root (x_0, y_0) with $|x_0 y_0|$ suitably bounded, can an attacker efficiently find (x_0, y_0)? Although the same technique appears to work for some bivariate polynomials, it is currently an open problem to prove it. As an increasing number of results depend on a bivariate extension of Coppersmith's theorem, a rigorous algorithm will be very useful.

Appendix B

Continued Fraction Method Fundamental Concepts

B.1 Continued Fractions

Definition 10. Continued Fraction

An expression of the form

$$x = a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 \cdots + \cfrac{b_{n-1}}{a_{n-1} + \cfrac{b_n}{a_n}}}} \tag{B.1}$$

is called a continued fraction. It is called a simple continued fraction if all the b_i's are equal to 1 and all the a_i's are integers such that $a_1, a_2, \cdots \geq 1$. $a_0, a_1, a_2, \cdots, a_n$ are called the continued fraction partial quotients. As a matter of simplicity, the designation of simple continued fractions is going to be referred only as continued fractions.

The notation $[a_0; a_1, a_2, \cdots, a_n]$ can be used to shorten the above expression.

A continued fraction can still be finite or infinite depending on whether it has a finite or an infinite number of terms, respectively.

Every rational number can be represented as a finite continued fraction with the conditions that the first coefficient is an integer and other coefficients are positive integers and this representation is not unique. There are two possible representations, which agree in all their terms except in their last one: in the longer representation of the continued fraction the last term in is set to 1 while in the shorter representation the final term 1 drops and the new final term is increased by 1. Thus, the final element in the short representation is therefore always greater than 1. It can be concluded that

$$[a_0; a_1, a_2, \cdots, a_n] = [a_0; a_1, a_2, \cdots, (a_n - 1), 1]$$

In its turn, an irrational number is represented by an infinite continued fraction $[a_0; a_1, a_2, \cdots]$ and an infinite continued fraction representation for an irrational number is useful because its initial segments provide rational approximations of that number.

As the main purpose of this revision is to introduce the theme for its application in the CFRAC algorithm (discussed in the section 4.3) and since the CFRAC method only uses continued fractions of irrational numbers, lets confine from now on the following results over infinite continued fractions only.

Definition 11. Convergent

Given a continued fraction $[a_0; a_1, a_2, \cdots, \cdots]$ *and an integer* $k \geq 0$, $C_k = [a_0; a_1, a_2, \cdots, a_k]$ *is called the* k*th convergent of the continued fraction* $[a_0; a_1, a_2, \cdots, \cdots]$.

Since the convergents of an irrational number can provide a rational approximation to the number, the infinite continued fraction $[a_1, a_2, \cdots, a_k, \cdots]$ can be defined as a limit of the convergents $C_k = [a_1, a_2, \cdots, a_k]$.

The convergent $C_k = [a_1, a_2, \cdots, a_k]$ is a finite continued fraction and thus it can be written as a simple fraction A_k/B_k, where A_k and B_k are termed, respectively, as the kth partial numerator and denominator.

Proposition 1. *Given a continued fraction, the convergents* A_{k-2}/B_{k-2}, A_{k-1}/B_{k-1} *and* A_k/B_k *verify the following equations:*

$$A_k B_{k-1} - A_{k-1} B_k = (-1)^{k-1} \quad , \quad k \geq 0 \tag{B.2}$$

$$A_k B_{k-2} - A_{k-2} B_k = (-1)^k a_k \quad , \quad k \geq 1 \tag{B.3}$$

Corollary 2. *The* k*th convergent* A_k/B_k *of any continued fraction verifies the following properties:*

- *The* k*th convergents with an even* k *form a strictly increasing sequence, whereas the* k*th convergents with an odd* k *form a strictly decreasing sequence;*
- *Any* k*th convergent with* k *odd is bigger than any* k*th convergent with* k *even;*
- $gcd(p_k, q_k) = 1$.

Theorem 15. *Let* $m \in \mathbb{N}$, $P_0 = 0$, $Q_0 = 1$, $P_k = a_{k-1}Q_{k-1} - P_{k-1}$ *and* $Q_k = (m - P_k^2)/Q_{k-1}$ *for each* $k \geq 1$. *In this conditions, all the continued fraction complete quotients* $[a_0; a_1, a_2, \cdots; \alpha_k]$ *of some* $\sqrt{m}$ *irrational number can be represented as*

$$\alpha_k = (\sqrt{m} + P_k)/Q_k$$

Theorem 16. *Let* $m \in \mathbb{N}$, $P_0 = 0$, $Q_0 = 1$, $P_k = a_{k-1}Q_{k-1} - P_{k-1}$ *and* $Q_k = (m - P_k^2)/Q_{k-1}$ *for each* $k \geq 1$.

The continued fraction k*th convergent of an irrational number* $\sqrt{m}$ *verifies, for each* $k \geq 0$, *the condition:*

$$A_k^2 - mB_k^2 = (-1)^{k+1} Q_{k+1}$$

B.2 Factor Basis and Smooth Numbers

Definition 12. (Factor Base) *A set of prime numbers that may include or not the integer* -1 *is called factor base, denoted by* $\mathcal{B}$.

Definition 13. (Smooth Number) *A smooth number is an integer which factors completely into small prime numbers. A smooth number is related to a factor basis* $\mathcal{B} = \{p_0, p_1, \cdots, p_{j-1}\}$ *if it can be written as*

$$n = \prod_{i=0}^{j-1} p_i^{e_i} \text{ with } e_i \in \mathbb{N}_0$$

The smooth are extremely important in the context of the current existing factorization methods. Consequently, smooth numbers are very important in cryptographic applications which support their security on the hard problem of large integers factorization.

Let $p_0 < p_1 < \cdots < p_{j-1}$ in the definition of smooth numbers (Definition 13) without loss of generality. The complete factorization of a number n using the primes existent in the factor basis $\mathcal{B}$ generates the exponent vector and the parity vector of n in the basis $\mathcal{B}$, respectively,

$$v_{\mathcal{B}}(n) = (e_0, e_1, \cdots, e_{j-1}) \quad \text{and} \quad v'_{\mathcal{B}}(n) = (e_0, e_1, \cdots, e_{j-1}) \,(\text{mod } 2)$$

Lemma 3. *Any set of* $j + 1$ *smooth numbers related to a factor basis with* j *elements has a subset of elements in which the product of all the elements is a square.*

Appendix C

Tables

Computational Information											
$M = 125000$				$N = 8$				$m = 9$			
Frequency											
Template	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8	χ^2	*P-value*	Result
000000001	255	227	251	247	228	243	241	265	4.973322	0.760424	Success
000000011	265	219	244	233	241	216	244	242	8.456924	0.390155	Success
000000101	223	223	235	266	226	254	259	251	9.104076	0.333593	Success
000000111	262	244	280	251	259	234	247	230	9.258714	0.320944	Success
000001001	256	270	251	238	239	257	264	243	6.285631	0.615271	Success
000001011	230	229	229	255	223	265	254	233	7.959116	0.437474	Success
000001101	269	240	236	229	227	217	236	242	8.600971	0.377067	Success
000001111	249	257	291	250	244	246	263	235	12.135414	0.145264	Success
000010001	255	261	265	227	250	232	260	266	8.660285	0.371759	Success
000010011	256	249	261	236	251	261	239	257	4.404549	0.818905	Success
000010101	255	245	291	260	253	252	224	267	15.410360	0.051641	Success
000010111	198	233	231	257	233	253	235	261	13.387350	0.099199	Success
000011001	235	214	229	250	260	253	237	263	8.438918	0.391810	Success
000011011	248	270	239	242	246	252	229	251	4.477631	0.811667	Success
000011101	251	221	250	244	249	247	254	239	3.272299	0.916126	Success
000011111	243	237	267	231	238	235	245	221	5.947756	0.653084	Success
000100011	242	262	250	240	251	237	246	252	2.284095	0.971040	Success
000100101	232	260	277	243	274	240	248	224	11.907693	0.155370	Success
000100111	228	211	287	265	228	249	271	251	19.847214	0.010930	Success

000101001	225	242	221	223	237	242	243	236	6.244323	0.619885	Success
000101011	245	265	259	244	246	264	229	272	8.736545	0.365006	Success
000101101	285	237	230	266	271	215	217	233	20.461531	0.008724	FAILURE
000101111	210	259	237	239	256	239	235	257	7.961234	0.437266	Success
000110011	232	237	233	250	245	241	259	272	5.782526	0.671577	Success
000110101	253	257	226	240	250	250	238	234	3.385629	0.907882	Success
000110111	260	264	238	227	254	262	239	258	6.836398	0.554383	Success
000111001	239	219	234	258	224	235	263	249	7.714448	0.461849	Success
000111011	247	233	251	249	238	256	261	241	2.864519	0.942570	Success
000111101	265	262	247	242	216	260	261	249	8.980153	0.343973	Success
000111111	243	237	255	229	264	235	239	246	3.843190	0.870987	Success
001000011	237	249	226	244	251	244	235	240	2.332817	0.969072	Success
001000101	258	260	225	203	253	227	248	257	12.940381	0.113917	Success
001000111	240	247	249	219	247	246	253	223	5.156558	0.740719	Success
001001011	249	262	262	235	274	241	235	227	8.578729	0.379069	Success
001001101	244	230	226	250	263	239	214	228	8.950496	0.346489	Success
001001111	247	243	272	243	248	254	248	243	3.883438	0.867484	Success
001010011	221	251	220	256	212	239	238	248	10.235267	0.248900	Success
001010101	241	250	256	248	212	256	227	244	7.060941	0.530072	Success
001010111	221	228	214	243	236	251	241	258	8.554368	0.381270	Success
001011011	280	241	250	242	288	244	233	251	14.539724	0.068738	Success
001011101	223	246	230	239	245	262	224	249	6.035667	0.643237	Success
001011111	258	245	270	242	274	252	235	251	8.271570	0.407402	Success
001100101	243	225	224	239	232	236	231	260	6.082270	0.638017	Success
001100111	241	239	242	260	213	217	251	259	9.598707	0.294328	Success
001101011	257	250	217	243	261	252	257	244	6.142643	0.631257	Success
001101101	225	243	262	278	241	261	243	243	9.028875	0.339866	Success
001101111	248	260	250	241	254	255	233	239	2.868756	0.942322	Success
001110101	233	205	252	229	242	224	247	225	11.561345	0.171875	Success
001110111	230	221	254	260	236	246	252	234	5.583402	0.693783	Success
001111011	264	273	257	250	212	249	249	236	10.907838	0.206976	Success
001111101	247	194	270	251	240	264	239	237	15.788483	0.045510	Success
001111111	224	259	257	253	262	233	251	216	9.118904	0.332365	Success
010000011	254	241	225	232	247	250	269	266	7.457070	0.488214	Success
010000111	230	231	253	255	238	247	227	248	3.909917	0.865158	Success

010001011	258	247	234	227	260	215	233	240	7.785412	0.454708	Success
010001111	252	240	262	229	256	250	237	231	4.346295	0.824607	Success
010010011	262	255	254	255	237	238	232	241	3.807178	0.874088	Success
010010111	259	265	265	267	262	246	212	229	13.556817	0.094074	Success
010011011	240	247	223	254	245	250	221	242	4.845163	0.773990	Success
010011111	227	240	298	257	236	253	251	241	15.168870	0.055944	Success
010100011	240	259	242	241	239	258	260	250	3.210867	0.920436	Success
010100111	209	236	227	248	243	235	246	250	7.332088	0.501263	Success
010101011	245	233	247	249	216	260	226	225	8.023725	0.431156	Success
010101111	260	233	251	248	242	249	238	242	2.153817	0.975929	Success
010110011	265	225	233	220	231	231	247	223	9.771351	0.281442	Success
010110111	282	225	244	251	263	238	242	251	9.715216	0.285586	Success
010111011	237	243	240	228	244	250	249	249	1.741801	0.987930	Success
010111111	246	245	268	251	273	268	243	241	8.627450	0.374691	Success
011000111	237	239	219	227	240	243	243	261	5.532562	0.699430	Success
011001111	238	239	219	254	215	246	246	234	7.415762	0.492509	Success
011010111	250	241	241	246	265	261	250	239	3.554037	0.894956	Success
011011111	247	274	238	232	250	236	207	275	14.901960	0.061080	Success
011101111	257	254	230	254	249	238	250	226	4.171532	0.841325	Success
011111111	242	264	251	245	266	252	249	249	4.387602	0.820570	Success
100000000	255	227	251	247	228	243	241	265	4.973322	0.760424	Success
100010000	253	252	243	236	241	241	265	232	3.433292	0.904305	Success
100100000	246	263	234	279	246	248	267	254	9.820073	0.277882	Success
100101000	212	267	248	232	243	227	258	245	9.342388	0.314241	Success
100110000	242	262	239	233	253	265	262	255	6.043081	0.642406	Success
100111000	215	230	255	264	239	240	237	234	7.446478	0.489314	Success
101000000	243	236	266	244	225	263	252	268	8.049145	0.428684	Success
101000100	225	233	262	251	274	226	261	232	10.630336	0.223537	Success
101001000	243	244	231	228	277	234	282	228	14.029205	0.081007	Success
101001100	244	261	237	232	206	252	249	216	11.917225	0.154935	Success
101010000	242	244	248	258	218	264	220	248	7.993009	0.434153	Success
101010100	238	266	223	239	229	267	235	260	8.794799	0.359901	Success
101011000	242	245	268	232	211	241	220	268	12.631104	0.125188	Success
101011100	213	265	224	256	239	243	242	232	9.022520	0.340400	Success
101100000	275	248	276	241	239	236	269	243	11.466020	0.176666	Success

101100100	255	202	248	248	262	238	258	227	11.717042	0.164282	Success
101101000	233	243	276	270	270	236	239	226	12.290052	0.138724	Success
101101100	263	252	270	250	258	265	257	248	8.182600	0.415840	Success
101110000	232	232	205	242	230	248	234	259	10.030847	0.262867	Success
101110100	239	249	266	240	222	262	274	268	11.935231	0.154117	Success
101111000	245	217	237	236	235	247	237	229	5.187274	0.737385	Success
101111100	247	268	232	253	232	251	215	244	7.823542	0.450894	Success
110000000	243	241	234	242	218	232	248	241	4.119633	0.846172	Success
110000010	255	267	254	251	229	274	247	247	8.151884	0.418776	Success
110000100	237	224	247	234	255	229	265	254	6.129933	0.632680	Success
110001000	265	268	228	224	225	245	239	232	9.365690	0.312392	Success
110001010	206	238	243	230	251	270	223	254	12.508241	0.129927	Success
110010000	250	250	230	259	243	264	229	239	4.834571	0.775102	Success
110010010	249	241	239	234	256	240	223	269	5.869378	0.661861	Success
110010100	225	270	253	249	209	227	242	252	11.571936	0.171349	Success
110011000	242	246	249	214	270	272	274	246	13.904224	0.084296	Success
110011010	237	259	233	276	265	238	244	222	10.060504	0.260805	Success
110100000	239	252	273	214	254	259	242	255	9.622009	0.292564	Success
110100010	245	241	269	241	254	232	231	230	5.318610	0.723043	Success
110100100	253	235	224	247	265	231	268	227	8.670876	0.370816	Success
110101000	248	252	257	258	239	234	257	248	3.155790	0.924204	Success
110101010	242	249	214	240	233	271	234	241	8.096807	0.424071	Success
110101100	255	231	267	233	230	226	226	247	7.636069	0.469802	Success
110110000	282	239	257	242	263	252	270	254	11.932053	0.154261	Success
110110010	258	234	220	259	257	258	253	237	6.719889	0.567134	Success
110110100	224	254	258	266	263	223	253	237	8.920840	0.349018	Success
110111000	241	205	227	241	255	250	245	287	16.249221	0.038949	Success
110111010	244	269	258	230	236	234	273	242	8.548013	0.381846	Success
110111100	236	236	246	244	231	263	212	224	8.901775	0.350650	Success
111000000	241	211	210	261	241	237	221	245	13.355575	0.100187	Success
111000010	241	236	244	242	249	251	241	271	3.742569	0.879571	Success
111000100	218	245	247	259	244	254	238	233	4.963789	0.761439	Success
111000110	253	229	255	262	242	257	214	211	12.372667	0.135335	Success
111001000	262	263	235	240	249	253	232	244	4.345235	0.824710	Success
111001010	235	244	237	237	190	227	261	234	16.077636	0.041282	Success

111001100	252	257	251	224	250	235	253	228	4.815506	0.777101	Success
111010000	240	235	260	231	231	265	239	282	10.987276	0.202421	Success
111010010	269	233	237	243	227	245	226	237	6.218903	0.622726	Success
111010100	250	220	256	238	264	261	264	247	7.956997	0.437682	Success
111010110	227	232	264	248	239	242	224	244	5.448888	0.708696	Success
111011000	273	254	260	229	275	275	265	256	16.503421	0.035716	Success
111011010	238	250	243	240	233	229	247	242	1.930333	0.983074	Success
111011100	246	221	226	257	238	245	239	280	10.100753	0.258026	Success
111100000	262	229	226	267	250	239	258	224	8.720657	0.366406	Success
111100010	203	248	241	225	234	260	225	235	12.224384	0.141470	Success
111100100	260	248	251	233	264	257	219	238	7.065178	0.529617	Success
111100110	257	277	230	255	283	257	230	219	17.252253	0.027587	Success
111101000	254	253	238	240	249	228	222	273	7.786471	0.454602	Success
111101010	235	234	252	232	260	262	268	258	7.324674	0.502042	Success
111101100	266	254	250	233	260	244	251	239	4.490341	0.810399	Success
111101110	246	251	252	254	244	226	257	272	6.277157	0.616217	Success
111110000	260	240	231	260	249	272	246	248	6.408494	0.601574	Success
111110010	232	264	266	230	229	244	239	214	10.094398	0.258463	Success
111110100	246	223	263	235	249	239	223	247	5.905389	0.657829	Success
111110110	254	236	254	238	261	257	233	278	8.559664	0.380791	Success
111111000	235	234	236	269	243	262	232	233	6.194542	0.625450	Success
111111010	241	254	279	243	245	259	251	251	6.953965	0.541608	Success
111111100	224	258	258	263	277	253	244	217	12.886364	0.115820	Success
111111110	242	264	251	245	266	252	249	249	4.387602	0.820570	Success

Table C.1: Statistics table of the Non-Overlapping Template Matching Test for a 1000000 bits number generated by a Luna SA HSM. Success means the acceptance of the null hipothesis (the hipothesis that states the sequence is random).

Computational Information											
$M = 125000$				$N = 8$				$m = 9$			
Frequency											
Template	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8	χ^2	*P-value*	Result
000000001	244453	243866	244528	243860	243632	244292	244701	244669	5.408738	0.713129	Success
000000011	244246	244208	244181	244114	243820	243561	244738	243642	4.500244	0.809409	Success
000000101	243851	244593	244777	243523	244060	244104	243564	244391	6.261703	0.617943	Success
000000111	243702	243982	244577	243827	244504	244555	244255	244321	3.625408	0.889242	Success
000001001	244032	243022	244004	244622	244548	243187	245637	244381	20.699494	0.007990	FAILURE
000001011	244698	244584	244556	243427	243887	244210	243951	244075	5.501132	0.702915	Success
000001101	243523	243639	243440	244435	244652	244142	243645	243750	7.923842	0.440945	Success
000001111	244375	244340	243939	243751	244190	243561	244095	244479	3.144085	0.924992	Success
000010001	244485	243898	244730	244184	243837	243727	244233	243804	3.863088	0.869260	Success
000010011	243847	243443	243319	244324	244477	243604	245258	243883	12.699726	0.122607	Success
000010101	244574	244926	244111	243076	244348	244041	244282	244363	8.733123	0.365307	Success
000010111	245060	245221	244994	243917	244369	244509	243387	244139	15.025791	0.058646	Success
000011001	244167	244724	244012	244041	243473	244537	244223	243929	4.329445	0.826244	Success
000011011	243867	243160	243067	244461	244264	243917	243699	244105	10.817311	0.212267	Success
000011101	243067	244551	244076	244231	244241	243863	244414	243712	7.113353	0.524454	Success
000011111	244736	244058	243581	243546	243781	243152	244921	244271	11.696244	0.165280	Success
000100011	244089	244824	244855	244840	244021	244340	243742	243789	7.650449	0.468338	Success
000100101	245048	244017	243820	244750	244499	243501	243690	244420	9.030194	0.339755	Success
000100111	244493	244282	244155	243846	244550	244336	245374	243080	13.062059	0.109732	Success
000101001	243306	243956	244078	244647	243942	244609	243393	244277	7.741933	0.459076	Success
000101011	244633	244637	243955	243962	244392	244611	243852	244226	3.941188	0.862390	Success
000101101	244275	243734	244006	243070	244164	245232	243937	243826	11.353568	0.182459	Success
000101111	244736	245517	244592	243708	243581	244069	243695	244245	13.420083	0.098190	Success
000110011	244057	244905	244416	243916	243558	244491	243194	244137	8.794746	0.359906	Success
000110101	243959	243438	244535	244210	244714	244022	244226	244222	4.422076	0.817178	Success
000110111	243541	243995	243918	244526	244295	244081	243603	244298	3.897736	0.866230	Success
000111001	243983	244313	244399	244477	244372	244599	243870	244616	3.378206	0.908434	Success
000111011	244099	244817	244215	243818	244350	244616	244491	243862	4.402256	0.819131	Success
000111101	244164	243369	244679	243896	243700	244119	244307	243538	6.486571	0.592898	Success
000111111	244355	243762	243397	244143	244240	243177	244624	244326	8.256113	0.408861	Success
001000011	244216	243392	243504	244490	243803	243685	244236	244220	6.060165	0.640493	Success
001000101	243498	243655	244679	243482	244958	244403	243623	244166	10.074525	0.259834	Success
001000111	244586	244954	243751	244636	243751	244086	244045	244486	6.526159	0.588508	Success
001001011	244053	244508	243970	244214	243808	244174	244369	244951	4.227283	0.836057	Success
001001101	243881	243698	243998	244810	244016	243550	245114	244781	10.395340	0.238367	Success
001001111	243841	244387	244253	244205	244393	244313	244779	243597	4.082986	0.849560	Success
001010011	244100	243950	244125	244376	243622	243824	244256	243542	3.535412	0.896424	Success
001010101	244059	244813	243444	243110	244349	243892	243942	244408	9.415434	0.308471	Success
001010111	244204	244033	243643	244147	243953	243847	243902	244675	3.080761	0.929187	Success
001011011	244912	242857	244303	243725	244627	244971	244277	243784	14.886371	0.061393	Success
001011101	244583	244291	244616	244118	244123	244385	244403	244063	2.456181	0.963743	Success
001011111	244810	244883	243820	243696	243847	244068	243715	244280	6.743709	0.564520	Success
001100101	244518	243992	243461	243306	242700	244488	243710	243741	16.370910	0.037369	Success
001100111	244814	244422	244302	244024	243782	244235	244126	243912	3.229471	0.919142	Success
001101011	244364	244153	244315	244377	244295	244422	245016	244298	4.365666	0.822718	Success

001101101	244091	243883	243981	244823	244006	243481	244000	243524	5.987001	0.648688	Success
001101111	244539	243535	243224	245250	243954	244251	244119	243859	11.537209	0.173077	Success
001110101	243411	244230	243955	244667	243251	244032	244234	243788	7.575652	0.475979	Success
001110111	243675	244527	243887	243948	244409	244505	245105	243861	7.119940	0.523749	Success
001111011	243875	243971	244648	244507	244657	244571	244579	243844	5.181710	0.737990	Success
001111101	243657	244331	244545	243995	244623	243452	243396	244321	7.408785	0.493237	Success
001111111	244132	244622	244057	244098	244377	244148	243970	244278	1.459686	0.993357	Success
010000011	243846	244124	243553	244273	243994	243565	244183	244072	3.428345	0.904679	Success
010000111	244608	243104	244241	244239	243871	243560	244709	244038	8.711294	0.367233	Success
010001011	243830	244623	244724	243246	244310	244504	243766	243957	7.645665	0.468825	Success
010001111	243876	244964	244334	244056	244133	244036	243463	244139	5.349674	0.719632	Success
010010011	243607	244188	244624	244481	243775	243362	245140	244674	11.268303	0.186955	Success
010010111	243667	244127	243480	244213	244257	243838	244779	244647	6.080595	0.638204	Success
010011011	244071	243178	244246	244391	244382	243744	244870	243408	9.700114	0.286708	Success
010011111	243900	244060	244085	244398	243843	244164	244505	243687	2.378440	0.967159	Success
010100011	243123	243916	244056	243879	243635	244311	243005	244773	13.285254	0.102405	Success
010100111	244363	244042	244347	244203	243245	244090	243627	244297	5.078078	0.749199	Success
010101011	244612	244267	243763	244267	244005	243488	244370	244640	4.842714	0.774247	Success
010101111	243741	243976	244210	244089	244096	244348	243909	244864	3.457972	0.902427	Success
010110011	243294	243852	244538	244005	244641	244830	244240	244828	9.254527	0.321282	Success
010110111	244193	243872	243903	243633	244239	245064	243781	244259	5.908832	0.657443	Success
010111011	245200	244384	243990	244676	243687	244344	244669	244398	8.826890	0.357109	Success
010111111	243811	244305	243578	244377	244366	244225	244205	244414	2.732177	0.950025	Success
011000111	243840	244062	243487	244488	244764	243949	244036	244594	5.449490	0.708629	Success
011001111	244263	243734	245653	244826	244537	244189	243244	244644	17.599867	0.024435	Success
011010111	244024	244098	243871	243788	244538	244715	244831	243603	6.210717	0.623641	Success
011011111	244361	243475	244224	244784	244656	244316	243641	244265	6.244790	0.619833	Success
011101111	244355	244085	244510	243922	243593	244657	246004	243364	20.654521	0.008124	FAILURE
011111111	243889	243960	244026	244166	244974	244945	245145	244106	10.427673	0.236282	Success
100000000	244453	243866	244528	243860	243632	244292	244701	244669	5.408738	0.713129	Success
100010000	244504	243946	243128	244593	244297	243751	244670	243449	9.891745	0.272706	Success
100100000	242931	244424	244351	243878	243739	243305	244342	244301	10.941242	0.205051	Success
100101000	244963	243850	243797	244962	243910	244587	243206	242816	18.785652	0.016049	Success
100110000	244457	243795	244283	243882	244180	244628	244555	244902	5.495976	0.703486	Success
100111000	244838	244359	244633	243831	244942	244547	244621	243503	9.816602	0.278135	Success
101000000	243727	243152	244648	243802	244232	243868	244753	243636	9.459824	0.305001	Success
101000100	244833	243380	243710	244680	245369	243892	245194	242836	25.067049	0.001514	FAILURE
101001000	243399	244164	244159	244215	243963	244509	243992	244634	4.190828	0.839509	Success
101001100	243606	244559	244571	243736	244194	243884	244779	244062	5.474726	0.705838	Success
101010000	244109	243517	244333	244074	243564	243854	244071	243753	4.241113	0.834740	Success
101010100	243516	244113	243788	243617	243267	243972	244330	243235	10.324847	0.242962	Success
101011000	244780	243559	243547	244034	242941	243870	243505	243949	12.980452	0.112524	Success
101011100	244112	244483	243449	243962	244725	244101	244792	244109	5.917209	0.656505	Success
101100000	245249	244812	244338	243630	243761	244425	243594	245059	14.176682	0.077275	Success
101100100	243430	243505	243835	244242	243963	243562	243370	243874	8.659312	0.371846	Success

101101000	244861	244145	244625	244156	244291	244628	244641	243481	7.200202	0.515195	Success
101101100	243827	244731	244381	244311	243368	243847	243747	244494	6.340930	0.609100	Success
101110000	243638	244074	243644	243585	244681	243499	243566	245242	12.961236	0.113190	Success
101110100	244086	245218	244754	243426	244859	244043	244584	243576	13.098341	0.108510	Success
101111000	244959	244889	243788	244578	244151	244259	244344	243666	7.737185	0.459555	Success
101111100	243861	244147	243938	244155	243790	244212	244232	244432	1.443706	0.993604	Success
110000000	244273	244355	244395	244018	243745	243556	244294	244285	2.905892	0.940124	Success
110000010	245509	244262	244886	244603	244429	244003	245196	244307	16.524209	0.035463	Success
110000100	244430	243883	243904	244780	243846	244284	243708	244726	5.304746	0.724563	Success
110001000	244337	245000	243760	244587	244425	243226	243810	243778	9.657178	0.289916	Success
110001010	243859	244678	244217	244572	243818	244527	243108	243775	8.529913	0.383488	Success
110010000	243837	243674	243947	243837	244473	243393	244101	244667	5.878912	0.660793	Success
110010010	244400	244260	243888	244699	243647	244006	244955	243871	6.163680	0.628903	Success
110010100	244004	244187	244358	244728	242697	244660	244225	243468	13.669196	0.090806	Success
110011000	244314	244004	243887	244104	244020	244084	243893	245108	4.784433	0.780348	Success
110011010	243794	244047	243526	244907	244040	243086	243670	243535	11.881993	0.156547	Success
110100000	243420	243461	244283	243483	244247	243801	245085	243660	11.368616	0.181675	Success
110100010	244406	244646	243566	243972	244505	244049	245041	243717	7.692853	0.464033	Success
110100100	243100	244684	244718	243836	244167	243265	244043	244403	11.227485	0.189140	Success
110101000	243562	243250	243904	244060	243668	244429	244175	243380	8.798268	0.359599	Success
110101010	244069	243567	244107	244774	242923	243568	244758	243467	14.327758	0.073612	Success
110101100	244156	243381	244307	244130	243808	244183	243417	244248	5.306900	0.724327	Success
110110000	244240	244365	243772	243893	243492	243369	243561	244926	9.431988	0.307174	Success
110110010	243501	243512	244159	244003	243958	243435	243721	243598	7.732992	0.459977	Success
110110100	244668	243649	243788	244193	244769	244572	244688	243164	10.512340	0.230890	Success
110111000	244325	244217	243692	243687	245221	243251	243323	244948	15.785245	0.045559	Success
110111010	243385	244671	244444	243137	244214	243969	244075	244620	9.407499	0.309094	Success
110111100	244623	244524	243544	244854	243046	244542	244657	243320	15.014342	0.058867	Success
111000000	243739	243744	244187	244116	244393	243466	244115	244461	3.997094	0.857386	Success
111000010	244792	244404	244568	243736	244569	244005	243783	245042	8.398580	0.395535	Success
111000100	244935	245481	244180	244169	244687	243514	244302	243608	14.535860	0.068824	Success
111000110	244038	244912	245150	244812	244201	244160	244087	244036	8.867551	0.353592	Success
111001000	244068	243746	243834	243612	244375	244819	243958	244961	7.439448	0.490044	Success
111001010	243620	244343	245146	244710	243586	244591	244390	245043	12.853889	0.116977	Success
111001100	244607	244868	243603	244634	244020	243968	244437	244751	7.557544	0.477838	Success
111010000	243779	243556	244104	243391	244876	244057	245161	243996	11.208886	0.190142	Success
111010010	244168	244352	243949	243723	244233	243480	244102	244397	3.256795	0.917224	Success
111010100	243469	243827	243783	244024	243130	244098	243859	244533	8.250125	0.409427	Success
111010110	243844	244132	244603	243794	243850	244093	243975	245041	5.706137	0.680110	Success
111011000	244684	244082	243554	243558	244641	243302	243180	244598	12.997621	0.111931	Success
111011010	244576	243163	244087	245086	244434	245361	244181	243325	18.150761	0.020124	Success
111011100	244613	244631	243648	244850	244726	244028	244124	244199	6.645571	0.575310	Success
111100000	244272	244339	244022	244493	244123	243545	245015	244000	5.652821	0.686056	Success
111100010	244988	245266	244495	243983	244781	244197	243549	243834	12.677387	0.123442	Success
111100100	244361	244045	243423	244382	244015	244371	244480	244755	5.052082	0.751995	Success

111100110	244130	244172	243089	244990	244054	243626	242880	244151	15.633368	0.047939	Success
111101000	243434	243257	244278	243676	244382	244310	244307	242995	12.463733	0.131681	Success
111101010	244246	244070	243924	244062	242600	243785	244163	244047	10.923795	0.206055	Success
111101100	243998	243752	244327	243917	244775	243735	243890	244466	4.201726	0.838480	Success
111101110	244722	244436	244948	244910	244363	243479	244160	244201	9.152564	0.329590	Success
111110000	243711	244028	243880	243667	244462	244398	244669	244302	4.085225	0.849353	Success
111110010	244315	243397	244571	244525	244084	244204	243867	244935	6.903658	0.547062	Success
111110100	243122	243004	244331	243943	244617	244493	244256	243554	13.190013	0.105477	Success
111110110	244377	244157	243750	243583	244630	243230	244071	243600	7.988114	0.434632	Success
111111000	243786	244414	244694	243394	244406	244439	244536	243808	6.315134	0.611978	Success
111111010	244019	243158	243659	244651	244108	244380	244675	243756	8.323470	0.402527	Success
111111100	243873	244625	244474	244260	244530	244650	243662	244193	4.552591	0.804149	Success
111111110	243889	243960	244026	244166	244974	244945	245145	244106	10.427673	0.236282	Success

Table C.2: Statistics table of the Non-Overlapping Template Matching Test for a 10^9 bits number generated by a Luna SA HSM. Success means the acceptance of the null hipothesis (the hipothesis that states the sequence is random).

Statistical Test	**Proportion**
Frequency Test	98/100
Frequency Test Within a Block	100/100
Runs Test	100/100
Test for the Longest Runs of Ones in a Block	100/100
Binary Matrix Rank Test	98/100
Discrete Fourier Transform (Spectral) Test	98/100
Non-overlapping Template Matching Test (1)	100/100
Non-overlapping Template Matching Test (2)	99/100
Non-overlapping Template Matching Test (3)	100/100
Non-overlapping Template Matching Test (4)	100/100
Non-overlapping Template Matching Test (5)	99/100
Non-overlapping Template Matching Test (6)	97/100
Non-overlapping Template Matching Test (7)	99/100
Non-overlapping Template Matching Test (8)	99/100
Non-overlapping Template Matching Test (9)	100/100
Non-overlapping Template Matching Test (10)	100/100
Non-overlapping Template Matching Test (11)	100/100
Non-overlapping Template Matching Test (12)	99/100
Non-overlapping Template Matching Test (13)	97/100
Non-overlapping Template Matching Test (14)	97/100
Non-overlapping Template Matching Test (15)	100/100
Non-overlapping Template Matching Test (16)	97/100

Non-overlapping Template Matching Test (17)	99/100
Non-overlapping Template Matching Test (18)	99/100
Non-overlapping Template Matching Test (19)	100/100
Non-overlapping Template Matching Test (20)	99/100
Non-overlapping Template Matching Test (21)	98/100
Non-overlapping Template Matching Test (22)	100/100
Non-overlapping Template Matching Test (23)	97/100
Non-overlapping Template Matching Test (24)	99/100
Non-overlapping Template Matching Test (25)	99/100
Non-overlapping Template Matching Test (26)	98/100
Non-overlapping Template Matching Test (27)	100/100
Non-overlapping Template Matching Test (28)	98/100
Non-overlapping Template Matching Test (29)	99/100
Non-overlapping Template Matching Test (30)	99/100
Non-overlapping Template Matching Test (31)	99/100
Non-overlapping Template Matching Test (32)	100/100
Non-overlapping Template Matching Test (33)	99/100
Non-overlapping Template Matching Test (34)	98/100
Non-overlapping Template Matching Test (35)	99/100
Non-overlapping Template Matching Test (36)	99/100
Non-overlapping Template Matching Test (37)	100/100
Non-overlapping Template Matching Test (38)	99/100
Non-overlapping Template Matching Test (39)	99/100
Non-overlapping Template Matching Test (40)	99/100
Non-overlapping Template Matching Test (41)	99/100
Non-overlapping Template Matching Test (42)	100/100
Non-overlapping Template Matching Test (43)	99/100
Non-overlapping Template Matching Test (44)	100/100
Non-overlapping Template Matching Test (45)	98/100
Non-overlapping Template Matching Test (46)	99/100
Non-overlapping Template Matching Test (47)	100/100
Non-overlapping Template Matching Test (48)	99/100
Non-overlapping Template Matching Test (49)	99/100
Non-overlapping Template Matching Test (50)	98/100
Non-overlapping Template Matching Test (51)	100/100
Non-overlapping Template Matching Test (52)	99/100
Non-overlapping Template Matching Test (53)	97/100
Non-overlapping Template Matching Test (54)	99/100
Non-overlapping Template Matching Test (55)	99/100

Non-overlapping Template Matching Test (56)	100/100
Non-overlapping Template Matching Test (57)	99/100
Non-overlapping Template Matching Test (58)	100/100
Non-overlapping Template Matching Test (59)	98/100
Non-overlapping Template Matching Test (60)	100/100
Non-overlapping Template Matching Test (61)	100/100
Non-overlapping Template Matching Test (62)	100/100
Non-overlapping Template Matching Test (63)	100/100
Non-overlapping Template Matching Test (64)	97/100
Non-overlapping Template Matching Test (65)	99/100
Non-overlapping Template Matching Test (66)	98/100
Non-overlapping Template Matching Test (67)	98/100
Non-overlapping Template Matching Test (68)	100/100
Non-overlapping Template Matching Test (69)	97/100
Non-overlapping Template Matching Test (70)	99/100
Non-overlapping Template Matching Test (71)	100/100
Non-overlapping Template Matching Test (72)	100/100
Non-overlapping Template Matching Test (73)	97/100
Non-overlapping Template Matching Test (74)	100/100
Non-overlapping Template Matching Test (75)	100/100
Non-overlapping Template Matching Test (76)	100/100
Non-overlapping Template Matching Test (77)	96/100
Non-overlapping Template Matching Test (78)	100/100
Non-overlapping Template Matching Test (79)	99/100
Non-overlapping Template Matching Test (80)	99/100
Non-overlapping Template Matching Test (81)	100/100
Non-overlapping Template Matching Test (82)	100/100
Non-overlapping Template Matching Test (83)	99/100
Non-overlapping Template Matching Test (84)	98/100
Non-overlapping Template Matching Test (85)	98/100
Non-overlapping Template Matching Test (86)	99/100
Non-overlapping Template Matching Test (87)	100/100
Non-overlapping Template Matching Test (88)	100/100
Non-overlapping Template Matching Test (89)	100/100
Non-overlapping Template Matching Test (90)	100/100
Non-overlapping Template Matching Test (91)	100/100
Non-overlapping Template Matching Test (92)	99/100
Non-overlapping Template Matching Test (93)	98/100
Non-overlapping Template Matching Test (94)	100/100

Non-overlapping Template Matching Test (95)	98/100
Non-overlapping Template Matching Test (96)	98/100
Non-overlapping Template Matching Test (97)	98/100
Non-overlapping Template Matching Test (98)	98/100
Non-overlapping Template Matching Test (99)	99/100
Non-overlapping Template Matching Test (100)	99/100
Non-overlapping Template Matching Test (101)	98/100
Non-overlapping Template Matching Test (102)	100/100
Non-overlapping Template Matching Test (103)	98/100
Non-overlapping Template Matching Test (104)	98/100
Non-overlapping Template Matching Test (105)	98/100
Non-overlapping Template Matching Test (106)	98/100
Non-overlapping Template Matching Test (107)	99/100
Non-overlapping Template Matching Test (108)	100/100
Non-overlapping Template Matching Test (109)	99/100
Non-overlapping Template Matching Test (110)	100/100
Non-overlapping Template Matching Test (111)	100/100
Non-overlapping Template Matching Test (112)	99/100
Non-overlapping Template Matching Test (113)	97/100
Non-overlapping Template Matching Test (114)	98/100
Non-overlapping Template Matching Test (115)	98/100
Non-overlapping Template Matching Test (116)	100/100
Non-overlapping Template Matching Test (117)	100/100
Non-overlapping Template Matching Test (118)	99/100
Non-overlapping Template Matching Test (119)	100/100
Non-overlapping Template Matching Test (120)	98/100
Non-overlapping Template Matching Test (121)	96/100
Non-overlapping Template Matching Test (122)	100/100
Non-overlapping Template Matching Test (123)	97/100
Non-overlapping Template Matching Test (124)	99/100
Non-overlapping Template Matching Test (125)	100/100
Non-overlapping Template Matching Test (126)	99/100
Non-overlapping Template Matching Test (127)	99/100
Non-overlapping Template Matching Test (128)	99/100
Non-overlapping Template Matching Test (129)	100/100
Non-overlapping Template Matching Test (130)	97/100
Non-overlapping Template Matching Test (131)	99/100
Non-overlapping Template Matching Test (132)	99/100
Non-overlapping Template Matching Test (133)	100/100

Non-overlapping Template Matching Test (134)	100/100
Non-overlapping Template Matching Test (135)	98/100
Non-overlapping Template Matching Test (136)	100/100
Non-overlapping Template Matching Test (137)	98/100
Non-overlapping Template Matching Test (138)	100/100
Non-overlapping Template Matching Test (139)	100/100
Non-overlapping Template Matching Test (140)	100/100
Non-overlapping Template Matching Test (141)	98/100
Non-overlapping Template Matching Test (142)	100/100
Non-overlapping Template Matching Test (143)	99/100
Non-overlapping Template Matching Test (144)	98/100
Non-overlapping Template Matching Test (145)	100/100
Non-overlapping Template Matching Test (146)	99/100
Non-overlapping Template Matching Test (147)	100/100
Non-overlapping Template Matching Test (148)	100/100
Overlapping Template Matching Test	97/100
Maurer's "Universal Statistical" Test	100/100
Linear Complexity Test	97/100
Serial Test (1)	99/100
Serial Test (2)	99/100
Approximate Entropy Test	98/100
Cumulative Sums Test (mode = 0)	99/100
Cumulative Sums Test (mode = 1)	98/100
Random Excursions Test (1)	98/100
Random Excursions Test (2)	99/100
Random Excursions Test (3)	100/100
Random Excursions Test (4)	100/100
Random Excursions Test (5)	99/100
Random Excursions Test (6)	99/100
Random Excursions Test (7)	97/100
Random Excursions Test (8)	99/100
Random Excursions Variant Test (1)	99/100
Random Excursions Variant Test (2)	99/100
Random Excursions Variant Test (3)	99/100
Random Excursions Variant Test (4)	99/100
Random Excursions Variant Test (5)	100/100
Random Excursions Variant Test (6)	100/100

Random Excursions Variant Test (7)	99/100
Random Excursions Variant Test (8)	100/100
Random Excursions Variant Test (9)	99/100
Random Excursions Variant Test (10)	99/100
Random Excursions Variant Test (11)	99/100
Random Excursions Variant Test (12)	99/100
Random Excursions Variant Test (13)	99/100
Random Excursions Variant Test (14)	99/100
Random Excursions Variant Test (15)	100/100
Random Excursions Variant Test (16)	100/100
Random Excursions Variant Test (17)	100/100
Random Excursions Variant Test (18)	100/100

Table C.3: Pass rates test for each one of the NIST Statistical Test Suit [Rukhin et al., 2010] based on 100 tests for 100 10^6 bits number generated by a Luna SA HSM.

Bibliography

[mod, 1980a] (1980a). Data Encryption Algorithm - Modes of Operation, ANSI X3.106-1983. American National Standards Institute.

[mod, 1980b] (1980b). FIPS 81: DES Modes of Operation. US National Institute of Standards and Technology.

[des, 1985] (1985). FIPS 112, Announcing the Standard for Password Usage.

[des, 1999] (1999). FIPS 46-3, Data Encryption Standard. US National Institute of Standards and Technology.

[dss, 2000] (2000). FIPS PUB 186-2, Digital Signature Standard (DSS). U.S.Department of Commerce/National Institute of Standards and Technology.

[aes, 2001] (2001). FIPS 197, Specification of the Advanced Encryption Standard (AES). United States of America, US National Institute of Standards and Technology.

[FIP, 2002] (2002). Security Requirements for Cryptographic Modules. Federal Information Processing Standards Publication (FIPS PUB) 140-2. U.S.Department of Commerce/National Institute of Standards and Technology (NIST).

[bat, 2012] (version of 2012). Batch-gcd. `http://facthacks.cr.yp.to/batchgcd.html`. accessed in December, 2014.

[Bernstein, 2004] Bernstein, D. J. (2004). How to find the smooth parts of integers. `http://cr.yp.to/papers.html#smoothparts` (accessed in December 2014).

[Bernstein et al., 2013] Bernstein, D. J., Chang, Y.-A., Cheng, C.-M., Chou, L.-P., Heninger, N., Lange, T., and van Someren, N. (2013). Factoring RSA keys from certified smart cards: Coppersmith in the wild. Cryptology ePrint Archive, Report 2013/599. `http://eprint.iacr.org/`.

[Bleichenbacher, 1998] Bleichenbacher, D. (1998). Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS1. pages 1–12. Springer-Verlag.

[Boneh, 1999] Boneh, D. (1999). Twenty Years of Attacks on the RSA Cryptosystem. *NOTICES OF THE AMS*, 46:203–213.

[Boneh and Durfee, 1999] Boneh, D. and Durfee, G. (1999). Cryptanalysis of RSA with private key d less than $n^{0.292}$. *Lecture Notes in Computer Science*, 1592:1–11.

[Brillinger, 1981] Brillinger, D. (1981). Time Series: Data Analysis and Theory. Holden-Day.

[Cavallar, 2000] Cavallar, S., L.-W. t. R. H. D. B. L. A. M. P. M. B. e. a. (2000). Factorization of a 512-bit RSA Modulus. *Eurocrypt*.

[Coppersmith, 1996] Coppersmith, D. (1996). Finding a Small Root of a Bivariate Integer Equation; Factoring with High Bits Known. *Lecture Notes in Computer Science*, 1070:178–189.

[Coppersmith, 1997] Coppersmith, D. (1997). Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities. *J. Cryptology*, 10:233–260.

[Coppersmith et al., 1996] Coppersmith, D., Franklin, M., Patarin, J., and Reiter, M. (1996). Low-Exponent RSA with Related Messages. In *In* Proceedings of the Advances in Cryptology – Eurocrypt '96 Conference, volume 1070. Lecture Notes in Computer Science.

[da Costa Boucinha, 2011] da Costa Boucinha, F. (2011). A Survey of Cryptanalytic Attacks on RSA. Master's thesis, Instituto Superior Técnico - Universidade de Lisboa.

[D'Agapeyeff, 2008] D'Agapeyeff, A. (2008). *Codes and Ciphers - A History Of Cryptography*. Hesperides Press.

[Davis and Fenstermacher, 1984] Davis, D., I. R. and Fenstermacher, P. (1984). Cryptographic Randomness from Air Turbulence in Disk Drives. In *Advances in Cryptology - Crypto '94*, volume 839. Springer-Verlag Lecture Notes in Computer Science.

[Delaurentis, 1984] Delaurentis, J. M. (1984). A further weakness in the Common Modulus Protocol for the RSA cryptoalgorithm. *j-CRYPTOLOGIA*, 8(3):253–259.

[Denker, 2003] Denker, J. (2003). High Entropy Symbol Generator. <http://www.av8n.com/turbid/paper/turbid.htm>.

[Diffie and Hellman, 1976] Diffie, W. and Hellman, M. E. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654.

[Durfee, 2002] Durfee, G. (2002). Cryptanalysis of RSA Using Algebraic and Lattice Methods. *PhD Thesis*.

[Eastlake 3rd et al., 2005] Eastlake 3rd, D., Schiller, J., and Crocker, S. (2005). Randomness Requirements for Security. <http://www.rfc-editor.org/info/bcp106>. BCP 106, RFC 4086.

[Ferguson and Schneier, 2003] Ferguson, N. and Schneier, B. (2003). Practical Cryptography. Wiley Publishing Inc.

[Franklin et al., 1996] Franklin, M., Patarin, J., and Reitert, M. (1996). Low-exponent RSA with related messages. pages 1–9. Springer-Verlag.

[Gifford, 1988] Gifford, D. (1988). Natural Random Number. MIT/LCS/TM-371.

[Goldreich and Levin, 1989] Goldreich, O. and Levin, L. A. (1989). A Hard-core Predicate for All One-way Functions. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 25–32, New York, NY, USA. ACM.

[Hamano and Kaneko, 2007] Hamano, K. and Kaneko, T. (2007). The Correction of the Overlapping Template Matching Test Included in NIST Randomness Test Suit. *Communications and Computer Sciences*, (E90-A(9)):1788–1792.

[Hastad, 1985] Hastad, J. (1985). On Using RSA with Low Exponent in a Public Key Network. In *CRYPTO*, volume 218 of *Lecture Notes in Computer Science*, pages 403–408. Springer.

[Heninger et al., 2012] Heninger, N., Durumeric, Z., Wustrow, E., and Halderman, J. A. (2012). Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *Proceedings of the 21st USENIX Security Symposium*.

[Herrmann and May, 2008] Herrmann, M. and May, A. (2008). Solving Linear Equations Modulo Divisors: On Factoring Given Any Bits. *Lecture Notes in Computer Science*, 5350:406–424.

[Hoffstein et al., 2008] Hoffstein, J., Pipher, J., and Silverman, J. (2008). *An Introduction to Mathematical Cryptography*. Springer Publishing Company, Incorporated, 1st edition.

[Howgrave-Graham, 1998] Howgrave-Graham, N. (1998). *Computational mathematics inspired by RSA*. PhD thesis, University of Bath (United Kingdom).

[Howgrave-Graham, 2001] Howgrave-Graham, N. (2001). Approximate Integer Common Divisors. *Lecture Notes in Computer Science*, 2146:51–66.

[Jakobsson and Juels, 1998] Jakobsson, M., S. E. H. B. and Juels, A. (1998). A practical secure random bit generator. In *Proceedings of the Fifth ACM Conference on Computer and Communications Security*.

[Katz and Lindell, 2007] Katz, J. and Lindell, Y. (2007). *Introduction to Modern Cryptography: Principles and Protocols*. Chapman & Hall/CRC Cryptography and Network Security Series. Taylor & Francis.

[Kiayias, 2009] Kiayias, A. (2009). Cryptography Primitives and Protocols. Notes by S. Pehlivanoglu, J. Todd, and H. S. Zhou.

[Lehmer and Powers, 1931] Lehmer, D. H. and Powers, R. E. (1931). On Factoring Large Numbers. *Bulletin of the American Mathematical Society*, 37(10):770–776.

[Lenore Blum and Shub, 1986] Lenore Blum, M. B. and Shub, M. (1986). A Simple Unpredictable Pseudo-Random Number Generator. *SIAM Journal on Computing*, 15:364–383.

[Lenstra et al., 2012] Lenstra, A. K., Hughes, J. P., Augier, M., Bos, J. W., Kleinjung, T., and Wachter, C. (2012). Public Keys. In Springer, editor, *Lecture Notes in Computer Science*. In Reihaneh Safavi-Naini and Ran Canetti.

[Lenstra et al., 1982] Lenstra, A. K., jun., H. W. L., and Lovász, L. (1982). Factoring Polynomials With Rational Coefficients. *Math. Ann.*, 261:515–534.

[Marsaglia, 1995] Marsaglia, G. (1995). DIEHARD Statistical Tests. <http://www.stat.fsu.edu/pub/diehard/>. (accessed in March 2015).

[May and Ritzenhofen, 2008] May, A. and Ritzenhofen, M. (2008). Solving Systems of Modular Equations in One Variable: How Many RSA-Encrypted Messages Does Eve Need to Know? In *Public Key Cryptography*, pages 37–46.

[Menezes et al., 1996] Menezes, A. J., Vanstone, S. A., and Oorschot, P. C. V. (1996). *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA.

[Mister and Adams, 1996] Mister, S. and Adams, C. (1996). Practical S-box Design. Selected Areas in Cryptography.

[Morrison and Brillhart, 1975] Morrison, M. A. and Brillhart, J. (1975). A Method of Factoring and the Factorization of F_7. *Mathematics of Computation*, (29):183–208.

[Nyberg, 1991] Nyberg, K. (1991). Perfect Non-linear S-boxes. Springer-Verland. Advances in Cryptography, Eurocrypt '91 Proceedings.

[Pomerance, 1982] Pomerance, C. (1982). Analysis and CFomparison of Some Integer Factoring Algorithms. Computational Methods in Number Theory, Part I. *Math. Centre Tracts. Amsterdam: Math. Centrum*, 154:89–139.

[Pomerance, 1996] Pomerance, C. (1996). A Tale of Two Sieves. *Notices of the AMS*.

[Revesz, 1990] Revesz, P. (1990). *Random Walk in Random and Non-Random Environments*. Singapore: World Scientific.

[Riesel, 1994] Riesel, H. (1994). Prime Numbers and Computer Methods for Factorization. *2nd Ed. Birkhäuser, Boston*.

[Rivest, 1992a] Rivest, R. (1992a). The MD4 Message-Digest Algorithm. RFC 1320.

[Rivest, 1992b] Rivest, R. (1992b). The MD5 Message-Digest Algorithm. RFC 1321.

[Rivest, 1990] Rivest, R. L. (1990). *Cryptography - Handbook of Theoretical Computer Science*, volume 1, chapter 13, pages 717–755. Elsevier. Posted pdf has 'cryptology' rather than 'cryptography' as the title, but is otherwise the same as what was published in book.

[Rivest et al., 1978] Rivest, R. L., Shamir, A., and Adleman, L. (1978). A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Commun. ACM*, 21(2):120–126.

[Rukhin et al., 2010] Rukhin, A., Soto, J., Nechvatal, J., Barker, E., Leigh, S., Levenson, M., Banks, D., Heckert, A., Dray, J., Vo, S., Smid, M., Vangel, M., and III, L. E. B. (2010). A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. NIST - National Institute of Standards and Technology, Technology Administration, U.S. Department of Commerce. Special Publication 800-22.

[Schneier, 1996] Schneier, B. (1996). Applied Cryptography: Protocols, Algorithms, and Source Code in C. John Wiley & Sons. 2nd Edition.

[Shor, 1997] Shor, P. W. (1997). Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.*, 26(5):1484–1509.

[Simmons, 1983] Simmons, G. J. (1983). A "Weak" Privacy Protocol Using the RSA Crypto Algorithm. *j-CRYPTOLOGIA*, 7(2):180–182.

[Stinson, 2002] Stinson, D. (2002). *Cryptography: Theory and Practice*. CRC/C&H, 2nd edition.

[Vazirani and Vazirani, 1984] Vazirani, U. V. and Vazirani, V. V. (1984). Efficient and Secure Pseudo-Random Number Generation. *Proceedings of the 25th IEEE Symposium on the Foundations of Computer Science*, pages 458–463.

[Vazirani and Vazirani, 1985] Vazirani, U. V. and Vazirani, V. V. (1985). Efficient and Secure Pseudo-Random Number Generation. *Advances in Cryptology: Proceedings of CRYPTO '84, Springer-Verlag*, pages 193–202.

[Von Neumann, 1963] Von Neumann, J. (1963). Various techniques used in connection with random digits. In *Von Neumann's Collected Works*, volume 5.

[W. Alexi and Schnorr, 1984] W. Alexi, B.-Z. Chor, O. G. and Schnorr, C. P. (1984). RSA and Rabin Functions: Certain Parts are as Hard as the Whole. *Proceedings of the 25th IEEE Symposium on the Foundations of Computer Science*, pages 449–457.

[W. Alexi and Schnorr, 1988] W. Alexi, B.-Z. Chor, O. G. and Schnorr, C. P. (1988). RSA and Rabin Functions: Certain Parts are as Hard as the Whole. *SIAM Journal on Computing*, 17:194–209.

[Wiener, 1990] Wiener, M. J. (1990). Cryptanalysis of Short RSA Secret Exponents. *IEEE Transactions on Information Theory*, 36:553–558.