



HAL
open science

Real-Time and Portable Chaos-based Crypto-Compression Systems for Efficient Embedded Architectures

Mohammad Abu Taha

► **To cite this version:**

Mohammad Abu Taha. Real-Time and Portable Chaos-based Crypto-Compression Systems for Efficient Embedded Architectures. Electronics. UNIVERSITE DE NANTES, 2017. English. NNT : . tel-01563932

HAL Id: tel-01563932

<https://hal.science/tel-01563932v1>

Submitted on 18 Jul 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Public Domain

Thèse de Doctorat

Mohammed ABUTAHA

*Mémoire présenté en vue de l'obtention du
grade de Docteur de l'Université de Nantes
sous le sceau de l'Université Bretagne Loire*

École doctorale : Sciences et technologies de l'information, et mathématiques (ED STIM 503)

Discipline : Electronique/spécialité:Sciences de l'Information et de la Communication.

Unité de recherche : Institut d'Electronique et de Télécommunications de Rennes UMR CNRS 6164(IETR)

Soutenance le 12 Juillet 2017

Real-Time and Portable Chaos-based Crypto-Compression Systems for Efficient Embedded Architectures

JURY

Présidente et Rapporteur : **M^{me} Danièle FOURNIER**, Professeur des universités, Université de Toulouse
Rapporteur : **M. Laurent NANA**, Professeur des universités, Université Bretagne Occidentale
Examineur : **M. Damien SAUVERON**, Maître de Conférences, HDR, Université de Limoges
Directeur de thèse : **M. Safwan EL ASSAD**, Maître de Conférences, HDR, Université de Nantes
Co-directeur de thèse : **M. Olivier DEFORGES**, Professeur des universités, INSA de Rennes
Co-encadrant de thèse : **M^{me} Audrey QUEUDET**, Maître de Conférences, Université de Nantes

Contents

I	INTRODUCTION	15
1	Preface	17
2	Context	18
3	Objectives and motivations	18
4	Thesis outline and contributions	19
5	Published work in journal and international conferences	20
6	Submitted papers for journal and conferences	20
II	BACKGROUND	21
1	State-of-the-art	23
1.1	Introduction	23
1.2	Chaos-based generator related works	24
1.3	Encryption schemes	25
1.3.1	Chaos-based block cipher cryptosystems	25
1.3.2	Stream ciphers	26
1.4	Real time applications	32
1.5	High efficiency video coding (HEVC)	34
1.5.1	HEVC tools	34
1.6	Selective video encryption related works	40
1.7	Conclusion	41
2	Parallel Programming and Software Security Tools	43
2.1	Introduction	43
2.2	Parallel programming models	43
2.2.1	MPI	44
2.2.2	OpenMP	46
2.2.3	Pthread	46
2.3	A brief introduction to OpenMP	46
2.3.1	OpenMP parallel principles	46
2.3.2	OpenMP memory architecture	46
2.3.3	OpenMP syntax	48
2.3.4	OpenMP parallel constructs	48
2.3.5	OpenMP data environment	49
2.3.6	Critical sections	51
2.3.7	Clauses/directives summary	52
2.3.8	OpenMP problems	52
2.4	Pthread-based multi-threaded programming	53
2.4.1	Pthread data types	53
2.4.2	Creating Pthread	53

2.4.3	Waiting for Pthreads to finish	54
2.4.4	Pthreads mutexes	54
2.4.5	Pthread conditions variables	55
2.4.6	Pthread barriers	57
2.5	Generating random numbers	58
2.6	Software security analysis	58
2.6.1	Static software security tools	60
2.6.2	Dynamic software security tools	61
2.7	Conclusions	62
III CONTRIBUTIONS		65
3	Efficient Implementation of Chaos-based Generators	67
3.1	Introduction	67
3.2	Proposed chaotic generator	68
3.3	Parallel implementation	74
3.3.1	Parallel implementation of the chaotic generator using OpenMP	74
3.3.2	Parallel implementation of the chaotic generator using Pthread	74
3.3.3	OpenMP vs Pthread	75
3.3.4	Computing performance of the chaotic generator	76
3.3.5	Statistical tests of proposed chaos based generator	76
3.4	First Application	86
3.4.1	Scheme of the proposed Pseudo-chaotic Number Generator as a RNG	87
3.5	Second Application: Proposed chaos-based stream cipher	91
3.5.1	Encryption computation performance and security analysis of the proposed stream cipher	92
3.5.2	Security analysis of proposed chaos-based stream cipher	94
3.5.3	Statistical analysis of the proposed chaos stream cipher	97
3.6	Software security implementation	100
3.7	Conclusion	104
4	Real Time Selective Encryption	105
4.1	Introduction	105
4.2	Real time encoder: Kvazaar	106
4.3	Proposed video encryption system	106
4.3.1	Encryption of intra prediction parameters	106
4.3.2	CABAC level encryption	106
4.3.3	Chaos-based encryption system	108
4.4	ROI encryption in HEVC	109
4.4.1	Tile-based encryption system	109
4.4.2	Encryption propagation in inter video coding	109
4.5	Results and discussions	110
4.5.1	Experimental setup	110
4.5.2	Objective measurements	112
4.5.3	Subjective evaluations	120
4.5.4	Complexity evaluations	123
4.6	Conclusion	124

IV	Conclusions and Future work Perspectives	125
5	Conclusions and Future work Perspectives	127
5.1	Conclusions and future work	127
	Appendices	139
A	Synthèse des travaux réalisés	141
A.1	Contexte et objectifs:	141
A.2	Résumé des travaux:	142
B	Static and dynamic analysis tools	149
B.1	Software security analysis tools	149
B.1.1	Static Software security analysis tools	149
B.1.2	Dynamic Software security analysis tools	151
B.2	Some of tools and limitations	153
B.2.1	Csur tool	153
B.2.2	Boon tool	154
B.2.3	Cqual tool	154
B.2.4	Parfai tool and coverity tool	154
B.2.5	Blast tool	154
C	Code Documentation	155
C.1	Class List	155
C.2	File List	155
C.3	key Struct Reference	155
C.3.1	Detailed Description	156
C.3.2	Member Data Documentation	156
C.4	cartes.c File Reference	159
C.4.1	Detailed Description	159
C.4.2	Macro Definition Documentation	160
C.4.3	Function Documentation	160
C.5	cartes.h File Reference	163
C.5.1	Macro Definition Documentation	163
C.5.2	Function Documentation	164
C.6	main.c File Reference	166
C.6.1	Detailed Description	167
C.6.2	Function Documentation	167
C.7	util.c File Reference	167
C.7.1	Detailed Description	167
C.7.2	Function Documentation	168
C.8	util.h File Reference	168
C.8.1	Typedef Documentation	169
C.8.2	Function Documentation	169

List of Tables

1.1	Computation performance.	31
1.2	Used coding scheme for IPMs in HEVC. The first bit is coded using a CABAC context. . .	37
1.3	Derivation process of the chroma intra prediction mode	38
2.1	OpenMP synchronization barriers	52
2.2	Clauses/Directives Summary	53
2.3	Pthread Types	53
2.4	Pthread simple code example	55
2.5	Pthread Mutex and conditional variable code example	56
2.6	Pthread barrier functions	57
2.7	Pthread barriers code example	59
3.1	Structure of the secret key.	72
3.2	Structure of parameters.	72
3.3	Number of samples computed by each thread ($N_s = 10$).	75
3.4	NCpB for Pthread and OpenMP implementation.	76
3.5	Generation time for sequential and parallel implementation with three delays.	77
3.6	Bit rate for sequential and parallel implementation with three delays.	77
3.7	NCpB for sequential and parallel implementation with three delays.	79
3.8	NCpB performance of some PRNG	79
3.9	Experimental and theoretical values of the Chi-Square test for the proposed generator. . .	81
3.10	CRE with delay=1.	83
3.11	Nist Test values with delay 1	85
3.12	Nist Test values with delay 2	86
3.13	Nist Test values with delay 3	86
3.14	Nist Test values for proposed PCNG.	88
3.15	Computing Performance of the proposed PCNG	90
3.16	Performance results of proposed Stream Cipher (V2) with different data sizes	94
3.17	Performance results of proposed Stream Cipher (V3) with different data sizes	94
3.18	Performance results comparison of some stream ciphers	95
3.19	The NPCR, UACI and HD	97
3.20	Chi-square value of histograms for different ciphered/plain images with different sizes . .	99
3.21	Correlation coefficient values for the previous plain/cipher images.	100
4.1	Encrypted bit of the chroma intra prediction mode.	107
4.2	Encrypted syntax elements.	108
4.3	The set of benchmark video sequences used in the experiment.	110
4.4	PSNR and SSIM values between original and encrypted videos (QP = 22).	111
4.5	PSNR and SSIM values for three video sequences with different QP.	111
4.6	Comparative evaluation, using weighted PSNR and SSIM for three sequences encoded by HM at (QP = 32).	112

4.7	Mean PSNR (Y) values in dB of the three video classes encoded by HM (QP = 22).	112
4.8	Mean SSIM values of the three video classes encoded by HM used (QP = 22).	112
4.9	BD-rate and complexity increase of the proposed encryption scheme in Intra and Inter coding (4×4 tile configuration).	113
4.10	BD-rate and complexity increase of the proposed encryption scheme in Intra and Inter coding (4×3 tile configuration).	114
4.11	Bjontegaard's difference for three video sequences with IPM Encryption.	114
4.12	The EQ for proposed SE and the state of the art [1] (QP = 22) encoded by HM.	115
4.13	NIST test of chaos based Key stream sequences.	121
4.14	The NPCR, UACI and HD	121
4.15	Ranking scale used in our subjective evaluation experiment.	122
4.16	ANOVA on the whole dataset, Df: number of degree-of-freedom and F-value: Fisher test	123

List of Figures

1.5	Statistical tests of Jallouli et al. PCNG.	33
1.6	HEVC video encoder [2].	35
1.7	HEVC partitioning [2].	36
1.8	Intra prediction modes on HEVC.	37
1.9	Derivation process of the three most probable modes.	38
1.10	Scanning mode on the HEVC	39
1.11	Main functions of CABAC engine.	39
1.12	Wavefront parallel processing (WPP).	40
3.4	Byte conversion.	73
3.6	Generation time for parallel and sequential implementation.	78
3.7	Bit Rate for parallel and sequential implementation.	78
3.8	NCpB for parallel and sequential implementation.	79
3.9	Mapping of three generated sequences for delays 1, 2 and 3.	80
3.10	Histograms of three generated sequences for delays 1, 2 and 3.	81
3.11	Space mapping division.	82
3.12	Auto and cross correlation for generated sequences, with delay 1.	84
3.13	Nist test results for generated sequences.	85
3.15	Statistical tests of the proposed PCNG.	89
3.16	Stream cipher encryption/decryption structure	92
3.17	Encryption time for parallel and sequential stream cipher.	95
3.18	Encryption throughput for parallel and sequential stream cipher.	96
3.19	NCpB for parallel and sequential stream cipher.	96
3.21	Histogram of the lena plain image and its ciphered image	98
3.20	NIST test cipher-image results.	98
3.22	Histogram of the Boat plain image and its ciphered image	100
3.23	Histogram of the Camera man plain image and its ciphered image	101
3.24	Histogram of the Peppers plain image and its ciphered image	102
3.25	Horizontal, Vertical and Diagonal correlation of the Boat plain image and its ciphered image	103
4.1	Intra Prediction Modes (IPMs) in the HEVC standard.	107
4.2	Selective encryption in HEVC at CABAC level.	108
4.3	MVs and in-loop filter restrictions.	109
4.5	The EDR for frame # 169 in BasketballDrive video sequence.	116
4.6	The EDR for frame # 184 in Kimono video sequence.	116
4.7	The EDR for frame # 37 in PeopleOnstreet video sequence	117
4.8	Frame #9 of HEVC videos encrypted with the proposed ROI encryption: (a) Correctly decrypted videos. (b) Encrypted videos.	118
4.9	Histogram for frame # 300 BasketballDrive video sequence.	119
4.10	Histogram for frame # 8 Kimono video sequence.	120
4.11	Subjects visibility scores including 95% confidence intervals for (QP = 22).	123

A.4	a) Image en clair de Caméraman, b) Image chiffrée correspondante, c) Histogramme de l'image en clair, d) Histogramme de l'image chiffrée.	145
A.5	Proportion des tests de NIST.	146
A.6	Nombre de cycles nécessaires pour chiffrer un octet en fonction de la taille des données.	146
A.7	Selective encryption in HEVC at CABAC level.	147
B.1	Frama-c analysis report.	151
B.2	Leak analyzer report.	152
B.3	Valgrind analysis report.	153
B.4	Callgrind analysis report.	153
B.5	DRD analysis report.	154
C.1	Code scheme parallel version.	170

I would like to dedicate this thesis to my family in Palestine ...

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Prof. Safwan El Assad for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to thank the rest of my thesis Co-supervisors: Prof. Olivier Deforges, and Dr. Audrey QUEUDET, for their insightful comments, academic and social support and encouragement. Prof. Olivier helps me a lot in my research conducted at INSA-Rennes. I am also very grateful to Dr. Audrey who was always so helpful and provided me with her assistance throughout my dissertation.

I would like to thank Dr. Wassim Hamidouche, for his help specially in the video encryption part.

My sincere thanks also goes to IETR lab directors who provided me an opportunity to join their team as intern, and who gave access to the laboratory and research facilities. Without their precious support it would not be possible to conduct this research.

I gratefully acknowledge the funding received towards my PhD from the European Celtic-Plus project 4KREPROSYS - 4K ultraHD TV wireless REMOTE PRO-duction SYStems.

Last but not the least, I would like to thank my family for supporting me spiritually throughout writing this thesis and my life in general.



INTRODUCTION

INTRODUCTION

1 Preface

For the time being, the range of cryptography applications have been expanded a lot with the fast changing world of information and communications technology; cryptography is essentially required to ensure that data are protected. The purpose of information security is to preserve: confidentiality, integrity and availability known as the CIA triad. In this context, confidentiality is a set of rules that restricts the information access, integrity is the assurance that the information is trustworthy and accurate, and availability ensures that reliable access to the information is performed by authorized people [3, 4].

In the following we provide the basic terminology of cryptography [5]:

1. *Cryptography* refers exactly to the methodology of concealing the content of messages. The information that we need to hide is called *plaintext*. It is the original text, it could be in a form of characters, numerical data, executable programs, pictures, or any other kind of information. The data that will be transmitted is called *ciphertext*. It is the term refers to the string of "meaningless" data, or unclear text that nobody can understand, except the recipients. It corresponds to the data that will be transmitted through the network. Cipher is the algorithm that is used to transform plaintext to ciphertext. This transformation is called encryption. It is a mechanism of converting readable and understandable data to something that appears to be random and senseless. In contrary the decipher is the process of converting the random and senseless data into readable one.
2. *Secret Key*: In cryptography, a secret Key (private key) is a variable that is used with an encryption/decryption algorithm. The algorithm doesn't need to be kept secret, but the key does. The secret key plays important roles in both symmetric and asymmetric cryptography.
3. *Symmetric encryption* refers to the process of converting *plaintext* into *ciphertext* and vice versa, using the same secret key. On the other hand, asymmetric encryption refers to the process of converting *plaintext* into *ciphertext* and vice versa, with different secret keys.
4. *Internet security* is a catch-all term for a very broad issue covering security for transactions made over the Internet. Generally, Internet security encompasses browser security, the security of data entered through a Web form, and overall authentication and protection of data sent via Internet Protocol.
5. *Types of Attacks*: It is crucial to know the major difference between two dominant types of threats, *active* and *passive*. An active threat is one that actively tries to damage or destroy your information. On the other hand, passive attackers want to keep tracking and monitoring what you are doing and when something interesting, like a credit card number or personal information, comes into view, they stealthily take a snapshot and send it back to their home server without being observed.
6. *Cryptanalysis* comprises the principles and methods of deciphering *ciphertext* without knowing the key, typically this includes finding and guessing the secret key, it's a complex process involving statistical analysis, analytical reasoning, math tools and pattern-finding. Usually a cryptanalyst tries to break the cipher without knowing the secret key, and this with several levels of difficulties based on the available resources. Chosen plain-text attack is the easiest one for the attacker: the attacker has access to the system without knowing the secret keys. Then, he has the possibility to choose

a set of plain-text messages and to encrypt them. If a cryptosystem can resist to chosen plain-text attack, then it can resist to all other attacks such as cipher text only, known plain-text and chosen cipher attacks. The chosen plain-text attack can be realized by the plain-text sensitivity attack or differential attacks introduced by Eli Biham and Adi Shamir [6]. Brute force is the attacker who is trying all of the possible keys that may be used in either decryption or encryption processes [7].

7. *Authentication* is a process in which the credentials presented by user are match those stored in the system database.

2 Context

In the ever changing world of huge data communications, low cost of internet connections, and very fast emerging in software development, security is becoming more and more challenge. Security is now a crucial requirement since global computing is inherently insecure. As the data goes from sender to receiver on the internet, for example, it may pass through several points along the way, giving other users the chance to intercept, and even modify it.

Technological advances in digital content processing, production and delivery have risen up into new signal processing applications in which security threats can no longer be handled in traditional mode. These applications range from multimedia content (image, video, audio, etc.), production and distribution to advanced biometric signal processing for access control, identity verification and authentication. In many of these cases, security and privacy risks may prevent the adoption of new image and video processing services.

Encryption is important because it allows to securely preserve data that you don't want anyone else to have access to. Governments, organization, individuals use it not only to protect classified, personal information to guard against attacks, but also, to securely protect folder contents, which could contain emails archives, chat histories, credentials information, credit card numbers, or any other sensitive information [8].

Security in real-time and embedded systems is a subject that has received an increasing amount of attention from industry and academic point of view in recent years. These systems are being deployed in a wide range of application areas [9, 10]. Embedded and real-time systems are facing more and more security problems. Malicious attacks on the system from suspicious or malicious code lead to system exception and security degradation.

3 Objectives and motivations

The use of cryptographic techniques in image and video processing applications is becoming increasingly common. The cryptographic techniques used in these applications must be able to protect the multimedia data against attacks. The confidentiality of images and videos contents is a hot topic and should be considered with a particular attention to both compression and encryption requirements. Chaos-based systems are more suitable to protect these huge data (images and videos). Indeed, there exists a good interesting relationship between chaos and cryptographic systems. The chaotic properties can be found in the classic Shannon's paper on cryptography [11], for example, the ergodicity, the sensitivity to initial condition or control parameters, deterministic dynamics, and structure complexity.

The performance of chaos-based crypto and crypto-compression systems consists in a trade-off between robustness against cryptanalysis and computational cost. For instance, in paid-for video services, the content must be protected against outsiders (non-authorized viewers) with a cryptographic end-to-end access control solution (encryption system). But at the same time the content must be protected against the valid customer to avoid them from broadcasting the video content illegally.

Many cipher algorithms are used nowadays to ensure system privacy and protection. Among them let us cite stream and block ciphers. Stream ciphers are based on producing an "infinite" random keystream, and using that to encrypt one bit or byte at a time (e.g. AES in counter mode, Rabbit, RC4), whereas block

ciphers work on larger chunks of data (i.e. blocks) at a time, often combining modes for additional security (e.g. AES in Cipher Block Chaining (CBC) mode).

The extremely rapid development of the Internet of Things (IoT) brings growing attention to the information security issue. Realization of cryptographically strong Pseudo Random Number Generators (PRNGs), is crucial in securing sensitive data. They play an important role in cryptography and in network security applications. For data protection, Pseudo Chaotic Number Generators (PCNGs) are the central element of any strongly secure chaos-based block and stream ciphers [12, 13].

High Efficiency Video Coding (HEVC) is the latest video coding standard which has been jointly standardized by the International Telecommunication Union (ITU), the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC), under the Joint Collaborative Team on Video Coding (JCT-VC). This team is composed of experts from the Video Coding Experts Group (VCEG) and the Moving Picture Experts Group (MPEG) which represent the ITU and the ISO/IEC, respectively. HEVC was designed to replace the successful Advanced Video Coding (AVC) standard, with the target of reducing the bitrate by 50% for the same visual quality [2].

A huge work has been devoted for video protection and access control. The most used methods allow to process the whole video as a unique data without taking into account the structure of the compressed video. However, these methods are not suitable for applications with real-time requirements.

This thesis aims at designing new solutions that answer these challenges. The work of this thesis will focus on designing very efficient i.e robust and fast PCNGs, chaos-based stream ciphers for securing images, and chaos-based crypto-compression systems for protecting video.

4 Thesis outline and contributions

Chapter 1 is dedicated to the state-of-the-art, including the fundamental techniques of classical and chaos-based cryptography and the HEVC standard. First, a related work on chaos-based generators is introduced. After that the standard and the chaos-based encryption schemes (block and stream) are presented. Finally, an overview of High Efficiency Video Coding with a brief state-of-the-art of the selective encryption is provided.

Chapter 2 provides a complete review of some existing parallel programming methods and tools. We point out those used in our thesis. Both the basic fundamentals of OpenMP and the Pthread libraries and their usage are described. OpenMP directives and clauses are summarized. In addition the pthread-based multi-threaded programming is detailed. Furthermore, a Linux pseudo-random generator is also described in this chapter. Finally, dynamic and static software security analysis tools are presented.

Chapter 3 develops our first contribution. It consists of designing and implementing in an efficient and secure manner a chaos-based generator. The chaotic system uses two non-linear recursive filters, a technique of disturbance and a chaotic multiplexing. The non-linearity is achieved by using chaotic maps. Based on the previous chaotic generators, two principal applications are implemented and tested. The first application concerns the generation of a Random Number Generator (RNG) using a Pseudo-Chaotic Number Generator (PCNG). The algorithm is refreshed many times by using entropy source from Linux kernel. The second application is the realization of a chaos-based stream cipher. The parallel version of the proposed chaos-based stream cipher is provided, and a comparison with other known chaos-based stream cipher is performed. The security performance of the two applications is analyzed using cryptanalytic attacks and statistical tests such as: Histogram, Chi-square test, correlation and the NIST. Experimental results highlight the robustness of the proposed systems. Also, the obtained generation and encryption speeds demonstrate their suitable use in real-time applications.

Chapter 4 describes our second contribution on a selective encryption solution for protecting HEVC video. The selective encryption is performed over a set of HEVC syntax elements in a format compliant with the standard. Thus, the bit-stream can be decoded with a standard HEVC decoder and only the secret key is needed for decryption. An encryption system of Region of Interest (ROI) makes use of the independent

tile concept of HEVC that splits the video frame into separable rectangular areas is also proposed. Tiles are used to extract the ROI from the background and only the tiles forming the ROI are encrypted. In Inter coding, tiles in-dependency is guaranteed by restricting the motion vectors of non-ROI to use only the unencrypted tiles in the reference frames.

Chapter 5 concludes this work and a summary of achieved goals is provided. The future work perspectives and potential improvements are finally addressed.

5 Published work in journal and international conferences

1. **M. AbuTaha**, S. El Assad, M. Farajallah, A. Queudet, and O. Deforge, “Chaos-based cryptosystems using dependent diffusion: An overview,” in 2015 10th International Conference for Internet Technology and Secured Transactions (ICITST). IEEE, 2015, pp. 44–49, **Published**.
2. **M. A. Taha**, S. El Assad, A. Queudet, and O. Deforges, “Design and Efficient Implementation of a Chaos-based Stream Cipher,” in Int. J. Internet Technology and Secured Transactions, Accepted paper, 2017, pp. 1 – 16, **Published**.
3. **M. A. Taha**, S. El Assad, O. Jallouli, A. Queudet, and O. Deforges, “Design of a pseudo- chaotic number generator as a random number generator,” in The 11th International Conference on Communications, 2016, pp. 401 – 404, **Published**.
4. O. Jallouli, S. El Assad, **M. A. Taha**, M. Chetto, R. Lozi, and D. Caragata, “An efficient pseudo chaotic number generator based on coupling and multiplexing techniques,” in International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2016), 2016, pp. 35–40, **Published**.
5. O. Jallouli, **M. Abutaha**, S. El Assad, M. Chetto, A. Queudet, and O. Deforges, “Comparative study of two pseudo chaotic number generators for securing the iot,” in Advances in Computing, Communications and Informatics (ICACCI), 2016 International Conference on. IEEE, 2016, pp. 1340–1344, **Published**.
6. Nabil Abdoun, Safwan El Assad, **Mohammad Abu Taha**, Rima Assaf, Olivier Deforges, Mohamad Khalil, “Hash Function based on Efficient Chaotic Neural Network” in 2015 10th International Conference for Internet Technology and Secured Transactions (ICITST). IEEE, 2015, **Published**.
7. Nabil Abdoun, Safwan El Assad, **Mohammad Abu Taha**, Rima Assaf, Olivier Deforges, Mohamad Khalil, “Secure Hash Algorithm Based on Efficient Chaotic Neural Network” in The 11th International Conference on Communications, 2016, pp. 405 – 410, **Published**.

6 Submitted papers for journal and conferences

1. **Mohammed Abutaha**, Wassim Hamidouche, Ari Koivula, Jarno Vanne, Naty Sidaty, Olivier Deforge and Safwan El Assad , “End-to-End Real-Time Encryption of ROI in HEVC” 25th EUSIPCO 2017 : European Signal Processing Conference, **Submitted**.
2. **Mohammed Abutaha**, Wassim Hamidouche, Naty Sidaty, Jarno Vanne, Olivier Deforges, Safwan El Assad, “Real Time Selective Encryption of ROI in the HEVC Standard” , in IEEE Transactions on Multimedia , **Submitted**.



BACKGROUND

State-of-the-art

1.1 Introduction

Cryptography is a needful tool for protecting information in computer systems. Modern cryptography has a strong relation with various disciplines like mathematics, computer science and electrical engineering along with data security. There are massive applications of cryptography in recent times, like passwords, e-commerce, smart cards etc. Chaos in cryptography was discovered by Matthews in the 1990s. Nowadays Chaos has been a hot research topic due to its interactive and interesting properties that depend on parameters and standard. This research has been attracting many researchers in the last decade. Chaos' properties, like randomness and ergodicity, have been proved to be convenient for developing the means for data protection. It is used in many systems and applications like biological, biochemical, meteorology and reaction systems. Furthermore, Chaos has potential applications in several functional blocks of a digital communication system: compression, encryption and modulation. Chaos-based cryptography is a research domain across two fields, i.e., chaos (nonlinear dynamic system) and cryptography (computer and data security) [14, 15].

With the development of internet technology, there is a growing demand for cryptographic techniques to secure transmitted multimedia contents (audios, images, videos) over the internet and mobile-phone networks. In all chaos-based cryptosystems, the chaotic generator is an important component of the system and so, a part of the effectiveness of the cryptographic system depends significantly on it [16].

It is difficult to imagine a fully-designed cryptographic application that doesn't use random number generators. Session keys, initialization vectors, salts to be hashed with passwords, unique parameters in digital signature operations, and nonces in protocols are all assumed to be random by system designers. Regrettably, many cryptographic applications don't have a reliable source of real random bits, such as thermal noise in electrical circuits. PRNG is used by real-world secure systems to generate cryptographic keys, initialization vectors, random nonces, and other values presumed to be random [17].

Generally, encryption can effectively protect sensitive information transmitted through insecure channels. As mentioned previously chaos has favourable properties that are suitable for encryption such as high sensitivity to initial values and system parameters, unpredictability, pseudo-randomness and ergodicity [18]. In addition, chaos-based cryptography is more modular than traditional cryptography. It is also more secure when stream ciphers are involved, and this is due to the strong non-linearity in such systems.

A huge work has been directed for image and video privacy. Video encryption is a hot research topic in the last decades. Very recently, an increasing attention has been devoted to the usage of chaotic systems to implement the encryption process of videos. The main advantage of such encryption lies in the observation

that a chaotic signal looks like noise for non-authorized users ignoring the mechanism for generating it. Secondly, time evolution of the chaotic signal strongly depends in the initial conditions and the control parameters of the generating functions. Slight variations in these quantum produce quite different time evolutions [19, 20, 21, 22].

In this chapter we review some related works in the literature that associated with our thesis. First, we introduce the chaos based generators and their related works. After that we present some chaos-based (block and stream), and standard ciphers of the literature. Finally, an overview of High efficiency video coding with a brief state-of-the-art of the Selective Encryption (SE) is given.

1.2 Chaos-based generator related works

Random sequences can be generated by using any non-linear dynamical systems. This generation is governed by a set of differential equations, iterative equations or simply chaotic maps [23, 24]. Many chaotic maps and chaotic generators have been proposed in the literature.

El Assad et al. [25] surveyed and analysed some digital chaotic generators in finite precision: the Logistic map, the piecewise linear chaotic map (PWLCM) and the Frey map. They demonstrated that these chaotic maps have a limited cycle length and do not exhibit very good statistical properties when used alone. Thus, to improve the properties of these maps, they integrate these maps with a recursive structure and implement a technique of disturbance of the pseudo-chaotic orbit [26].

Rene 'Lozi [27], presented a new model of a weakly coupled logistic and symmetric tent maps, based on a matrix of coupling and using single or double precision numbers. He showed that the 3-coupled tent maps with small value of perturbation can be used as a generator of pseudo-chaotic numbers. The generated sequences have uniform distribution and their orbits have a very long period which are greater than 10^9 but less than 10^{12} . He demonstrated that the 3-coupled symmetric tent maps model seems a sterling model of generator of chaotic numbers with a uniform distribution over the interval $[-1,1]$. In [28], Lozi used a double threshold chaotic sampling and mixing in a weakly coupled tent maps in order to increase the randomness of the generated sequences. However, the double threshold chaotic sampling technique decreases the bit rate performance.

Li et al. [29, 30, 31] demonstrated the statistical properties of digital piecewise linear chaotic maps. They indicated their roles in cryptography and pseudo-random coding. They proposed a pseudo-random number generator (PRNG) with very good cryptographic properties based on a system of two chaotic maps.

Kurgansky et al. presented a one-dimensional piecewise affine maps which is comparable to pseudo-billiard or so-called strange billiard systems. They showed that the one-dimensional *Piecewise Affine Maps (PAMs)* are equivalent to *planar Pseudo-Billiard Systems (PBSs)*. The reachability problem for PAMs is still open, however the more general model of rational one-dimensional maps is shown to be universal with undecidable reachability problem [32].

Thomas et al [33], proposed a hardware random number generator based on a *Linear Feedback Shift Register (LFSR)* and a *Cellular Automata Shift Register (CASR)*. The output of the generator is resulted from the xor operation between the LFSR and the CASR.

Other well-known bi-dimensional chaotic maps such as *2D-Standard map*, *2D-Cat map*, *2D-Baker map* were discussed and used in symmetric ciphers [27, 34, 35, 36].

Lian et al. [24] elaborated the performance of the 2D Standard, Cat and Baker maps. They demonstrated that the Cat map has the smallest key space. In this work the key space is enlarged and the key sensitivity is increased. The key spaces for Standard map and Baker map are both larger than that of the Cat map. However to keep high key sensitivity, the number of iterations should be larger than 4 for the Standard map and bigger than 12 for the Baker map. These maps are more suitable for cryptosystems in which the same key is used in different iterations. However, to achieve a good confusion property, the average distance change in the whole image must be greater than 40%.

Most of the previous chaos-based generators operate with `floating-point` data operations. This raises a problem when the computer's resolution of the sender and receiver are slightly different. Indeed, in this case, the chaotic sequences generated by the emitter in a cryptographic system will be different from the one generated by the receiver. To overcome this problem a `fixed finite precision` of N bits is used. However, the chaotic dynamics are degraded with a finite precision N and short cycles may occurred [23, 26].

Shannon clarifies [11], that the fundamental techniques to encrypt a block of bytes are substitution and permutation. A chaos-based encryption algorithm relies on chaotic maps. Indeed, the substitution and permutation operations are performed according to a chaotic sequence. In the following section we report some of them.

1.3 Encryption schemes

Symmetric ciphers can be divided into block ciphers and stream ciphers. Block ciphers encrypt a particular block of plaintext bits at a time with the same key. In this method the encryption of any plaintext bit in a given block depends on every other plaintext bit in the same block. For example, the Advanced Encryption Standard (AES) which is a block cipher uses a block length of 128 bits (16 bytes). Stream ciphers encrypt bits individually. This is performed by adding a bit from a key stream to a *plaintext* bit. There are synchronous stream ciphers in which the key stream depends only on the key, and asynchronous ones where the key stream also depends on the ciphertext. In the following we discuss some of the existing block/stream chaos-based image encryption schemes.

1.3.1 Chaos-based block cipher cryptosystems

Recently, many chaos-based cryptosystems, more efficient in terms of time consuming and of resistance against cryptanalysis, than the previous ones have been investigated [1, 7, 34, 37, 38, 39, 40, 41]. These *cryptosystems* have a very high security level and they are based on dependent confusion-diffusion layers, generally achieved by substitution-diffusion processes that use dynamic keys supplied by a chaotic sequence. The chaotic sequence is produced by a chaotic generator which is the heart of any chaos-based cryptosystem and so a big part of the efficiency of the system depends on it. Compared to the conventional cryptographic algorithms (3DES, AES), chaos-based cryptosystems have several advantages such as: more flexibility, more modularity, a low power consuming, and easily implemented, which make them more suitable for large scale-data encryption, such as images and videos.

All chaos-based and non chaos cryptosystems must achieve the `confusion` and `diffusion` effects. The confusion effect is measured by how much a change in the secret key affects the ciphered message. The diffusion effects is measured by how much a change in the plain message affect the ciphered message. In the literature, there are mainly two types of chaos-based cryptosystems. The structure of the first type is composed of two layers: a `confusion` layer followed by a `diffusion` layer that work separately (see in Figure 1.1). The `confusion` process is applied rc times on the block (or on the whole image), then the `diffusion` process is applied rd times on the output of the `confusion` process, and finally, the two processes are repeated r times. Both layers required image-scanning (for $rc = rd = r = 1$). Most of chaos-based cryptosystems of first type are considered insecure upon chosen/known plain text attacks. El Assad et al. [42] gave in their paper an overview of main chaos-based cryptosystems of first type. The structure of the second type of *cryptosystems* is similar to the structure of the first type of *cryptosystems*, but the confusion and diffusion processes are performed sequentially on each pixel of the plain block or plain image as shown in Figure 1.2. This type of *cryptosystems* are more efficient, in terms of security and speed performance, than the first type of *cryptosystems*. Indeed, first, the `diffusion` process at the pixel level is governed by the confusion process, second, a single scan of plain image pixels is needed to perform the confusion and diffusion effects.

In the following we will recall the main chaos-based *cryptosystems* of the second type.

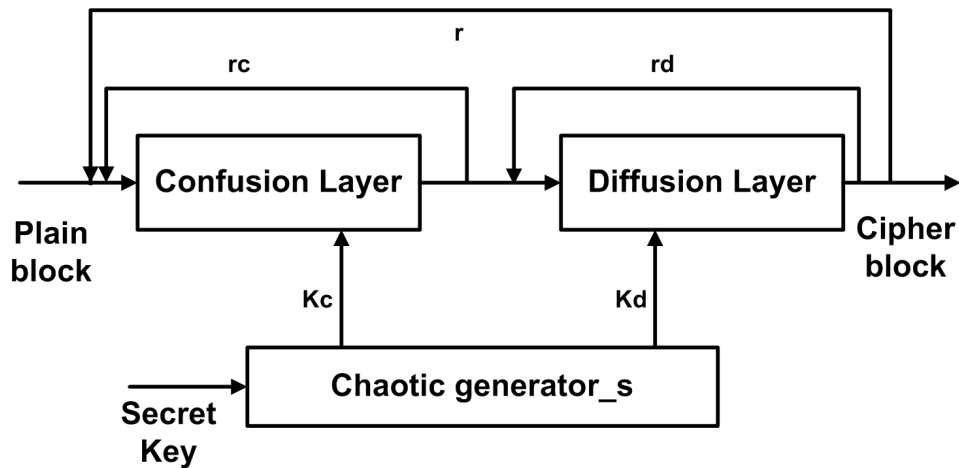


Figure 1.1 – General structure of chaos-based cryptosystems

Yang et al. [38], used a permutation operation, as a confusion layer, achieved by a modified

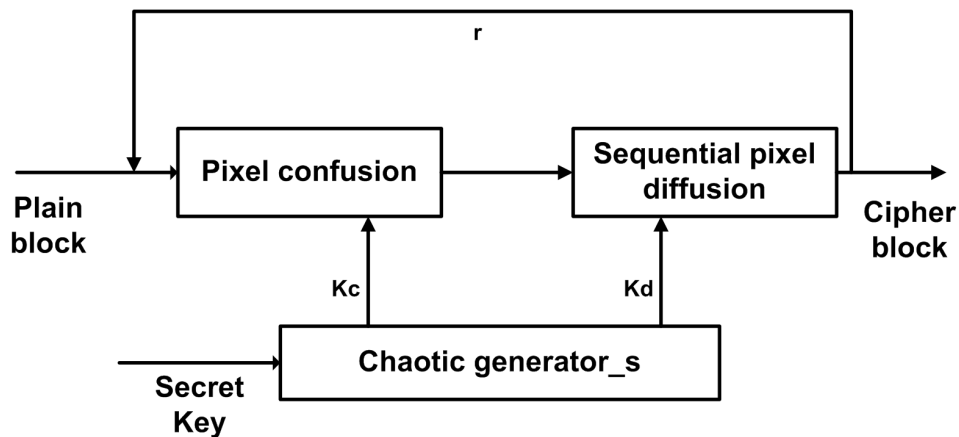


Figure 1.2 – Dependent diffusion structure of chaos-based cryptosystems

standard map to avoid the problem of permutation of the corner pixel ($s = 0, t = 0$), while using a logistic map as a diffusion layer. In addition a keyed hash function is used to generate a $128 - bit$ hash value from both the plain image and the secret hash keys. The hash value plays the role of the key for encryption and decryption while the secret hash keys are used to authenticate the decrypted image. In their paper, Wang et al. [40] introduced the idea of mixing the two layers of permutation and diffusion into a single layer of dependent permutation-diffusion. As a result, one image scanning is required instead of two scanning stages, to accelerate the encryption algorithm.

In Zhang's model [41], two *cryptosystems* were designed based on the architecture of Figure 1.2. The first one consists of a dependent diffusion layer based on the reverse 2-D cat map. The second algorithm presents new conversion from a pseudo-random position to another pseudo-random one for the confusion effect. The diffusion layer in the cryptosystems is based on the logistic map. In these versions, Zhang tried to achieve the confusion and the diffusion effects sequentially. Farajallah [1, 39] proposed an efficient cryptosystem with a very high speed compared to the main chaos-based cryptosystem of the literature.

1.3.2 Stream ciphers

A typical *stream cipher* encrypts plaintext one byte at a time, although a stream cipher can be designed to encrypt one bit at a time or on units larger than a byte at a time. Figure 1.3 represents a stream cipher

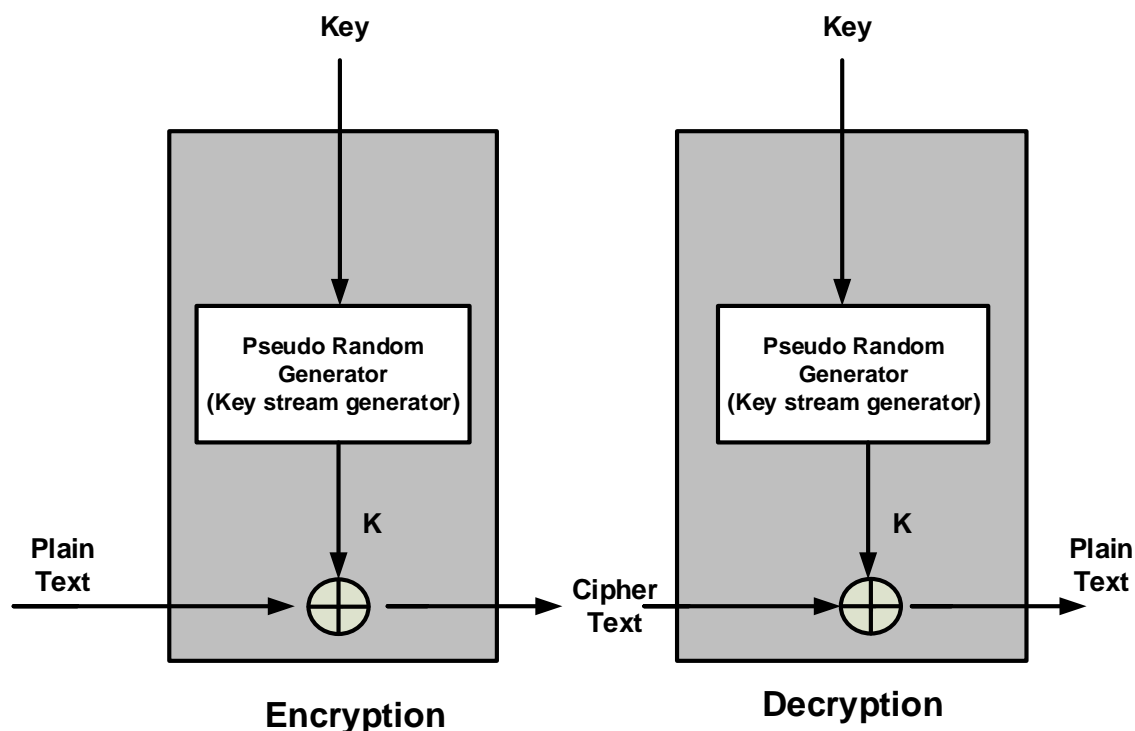


Figure 1.3 – Stream cipher diagram.

structure diagram. In this structure, a key is an input to a pseudo-random bit generator that produces a stream of specified bit numbers that are apparently random. The output of the generator, called a keystream sequences, combines one byte at a time with the plaintext stream using the bitwise exclusive-OR (XOR) operation to produce a ciphertext. Three design considerations must be taken into account when we design a *stream cipher* algorithm [43]:

1. The `keystream` sequence should have a large period. A pseudo-random number generator uses a function that produces a deterministic stream of bits that may repeat. The longer the period of repeat the more difficult it will be to do cryptanalysis.
2. The more random the `keystream` is, the more randomized the ciphertext is, making cryptanalysis more difficult.
3. The key needs to be sufficiently long in order to guard against `brute-force` attacks.

In the following we recall the main related works in `standard` and `chaos-based` stream ciphers.

1.3.2.1 AES-CTR and eSTREAM software

AES-CTR Mode

Counter mode, a standard introduced by Diffie and Hellman in 1979, is one of the best known modes used for stream ciphers. Counter mode switches a block cipher into a stream one. It generates the next keystream block by encrypting successive values of a counter. After each block encryption, the counter must be different and this can be done simply by incrementation of the counter by some constant, typically one. CTR mode has significant efficiency advantages over the Cipher Feedback (CFB) and Output Feedback (OFB) modes without weakening the security. In particular its tight security has been proven. On the other hand most of the perceived disadvantages of CTR mode are not valid criticisms, but rather caused by a lack of knowledge [44].

Rabbit

Rabbit is a *stream cipher* algorithm developed in 2004 as a fast software encryption method. It is one of the most effective algorithm proposed in the eSTREAM project. Rabbit is directed to be used in both software and hardware applications. The Rabbit algorithm takes a 128-bit key and a 64-bit IV vector as input. At each iteration, it generates a 128-bit output. The output is pseudo-random in its nature. The heart of this cipher consists of 513 internal state bits. clearly the output generated in each iteration is some combination of these state-bits. The 513 bits are divided into eight 32-bit state variables, eight 32-bit counters and one counter carry bit. The state functions which update these state variables are non-linear and thus build the basis of the security provided by this cipher [45, 46]. The designers provided the security analysis considering several possible attacks: algebraic, correlation, and statistical attacks. They conclude that no huge weakness of Rabbit has been found. However in 2009, Kircanski and Youssef in their paper [47] provide a differential fault analysis attack on Rabbit algorithm. The fault model in which they analyse the cipher is the one in which the attacker is assumed to be able to fault a random bit of the internal state. The attack requires around 128-256 faults, a precomputed table of size around $2^{41.6}$ bytes, and enables to recover the complete internal state of Rabbit in about 2^{38} steps.

Salsa20/r

Salsa20/r is one of the eSTREAM finalist algorithms for software implementation, where $r = 8, 12, 20$ represents the number of iterations of the round function. The algorithm is constructed on a pseudo-random function based on a 32-bit addition, bitwise XOR and rotation operations, which maps a 256-bit key, a 64-bit nonce (IV initial vector), and a 64-bit stream position to a 512-bit output [46, 48]. The Salsa20/8 version is very fast but not secure enough. Its weakness comes from a differential cryptanalysis performed by Tsunoo et al. [49]. Salsa20/12 and Salsa20/20 algorithms seem to be secure so far, because no better attack than the brute-force attack has been reported.

HC-128 and HC-256

HC-128 is an efficient software stream cipher, which consists of two secret tables, each one with 512 32-bit elements. At each step they update one element from one of the two tables using a non-linear feedback function. All the elements of the two tables are updated every 1024 steps. At each step, one 32-bit output is generated from the non-linear output function. HC-256 is a new version that differs from HC-128 by the size of secret tables which is 1024 32-bit elements instead of 512 32-bit ones. All the elements of the two tables are updated every 2048 steps. At each step, HC-256 produces one 32-bit output [46, 50, 51]. However, in 2010, the authors in [52] provide a differential fault analysis attack on HC-128. The attack is based on the fact that, some of the inner state words of HC-128 may be exploited several times without being updated. Consequently, the complete internal state is recovered using about 7968 faults.

SOSEMANUK

SOSEMANUK is a software stream cipher that has a key length ranging from 128 to 256 bits. It takes an initial value IV vector of 128 bits, and has two main components: a *linear feedback register (LFSR)* and a *finite state machine (FSM)*. The LFSR operates on 32-bit words and at every clock a new 32-bit word is computed. The FSM has two 32-bit memory registers: at each step the FSM takes an input word from the LFSR, updates the memory registers and produces a 32-bit output [46, 53]. In 2011 the authors in [54] made a differential attack on SOSEMANUK. The attack needed around 6144 faults to recover the secret inner state of the cipher.

1.3.2.2 Chaos-based stream ciphers related work

Abderrahim et al. [55] in their paper proposed a chaos-based stream cipher based on symbolic dynamic description and synchronization. Their main contribution concerns a pseudo-random number generator (PRNG) based on an appropriate mixture of perturbed chaotic maps. The synchronization of the emitter/receiver is performed by a symbolic dynamic-based method. One of the characteristics of their proposed stream cipher is that the chaotic symbolic dynamic sequences are easy to produce. The obtained bit rate, with an Intel Core i7 processor clocked at 3.5 GHz, and 8Gb of RAM is 10 Mbps.

Lu et al. [56], proposed a one-way-coupled chaotic map lattice for cryptography of a self-synchronizing stream cipher. The system performs the computation into real numbers, and incorporates some algebraic operations on integer numbers. The encryption/decryption operations is done in parallel using multiple chaotic maps. The authors claim that the system has a good security level, and good reliability against strong channel noise. They provide an encryption speed (around 914 Mbps on a 2 GHz CPU).

In 2007 li et al. [57] published a stream cipher also based on a spatiotemporal chaotic system as done previously in [56]. The chaotic system uses coupled logistic maps, and simple algebraic computations. The system produces parallel keystreams for encrypting plaintexts via bitwise XOR. The encryption speed is 700 Mbits in a computer with a 1.8 GHz CPU and 1.5 GB RAM. Security analysis is performed to prove the robustness of the system. However, the *cryptosystem* displays weakness in the keystream generation [57]. The encryption is made by generating a keystream mixed with blocks generated from the plaintext. The obtained keystream remains identical for every encryption procedure. Moreover, its generation does neither depend on the plaintext nor on the ciphertext, that's to say, the keystream remains unmodified for every plaintext with the same length. Knowing the keystream leads to guessing the key. These drawbacks are detailed in [58].

Shubo et al. [59] presented an improved chaos-based stream cipher algorithm based on discrete chaotic maps. In this algorithm, one logistic chaotic system generates the random changing parameter to control the parameter of the other. The algorithm only disturbs the control parameter of the chaotic system. The VLSI architecture with low hardware cost and fast speed is designed, and the FPGA realization is shown. The encryption speed is 571.429 Mbps with an AMD Athlon(tm) 64 X2 Dual Core processor.

In 2007, Fu et al. [60] designed a chaotic stream cipher relies on logistic map. They claimed that this cipher could resist various common attack methods. In [61], the security of Fu's chaotic cipher is analysed and the information leak of chaotic map is indicated. They succeeded to guess the initial state and obtained two sampling quantified sequences which are generated by other two chaos initial states. A compression attack is proposed to recover the chaos initial state from sampling quantified sequence.

Yin et al. [62] published a new stream cipher with the discretized coupled map lattices (CML) which operates on binary numbers. CML have been recently used to construct ciphers. However, the complicated operations on real numbers make these CML-based ciphers difficult to analyze.

In 2016, Jallouli et al. [63] part of our research team members in IETR laboratory, proposed a Pseudo random chaotic generator (PCNG). In the following subsections we will provide in detail the structure and the computation performance of this PCNG.

1.3.2.3 Jallouli et al. PCNG

The structure of the PCNG is presented in Figure 1.4. It uses four coupled chaotic maps (two PWLCM maps, STmap and Logistic map) and includes a multiplexing chaotic technique [63]. The secret key of the system is formed by:

- The initial conditions X_{p1} , X_s , X_{p2} and X_l of the four chaotic maps: (PWLCmap1, STmap, PWLCmap2 and Logistic respectively), ranging from 1 to 2^N-1 ,
- The control parameter Pp1, Ps and Pp2 of PWLCmap1, STmap and PWLCmap2, in the range $[1, 2^{N-1} - 1]$, $[1, 2^N - 1]$ and $[1, 2^{N-1} - 1]$ respectively.

The internal state function consists of two main steps.

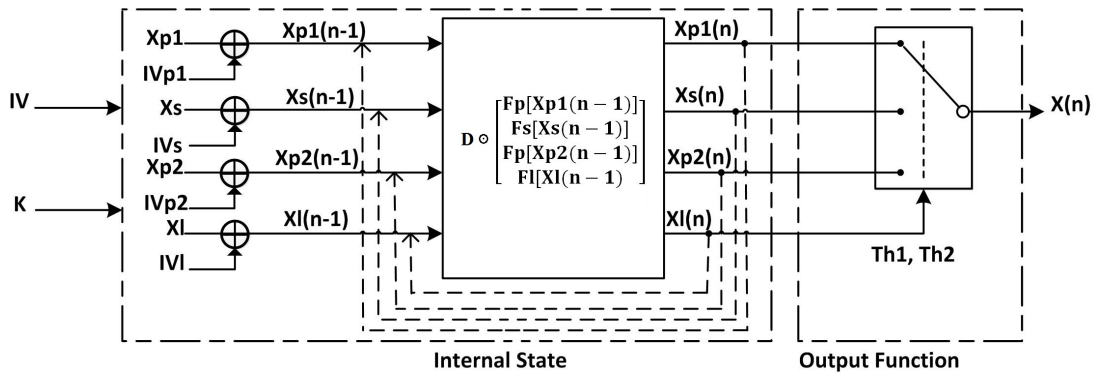


Figure 1.4 – The architecture of the PCNG2.

- First, the initial values $Xp1(0)$, $Xs(0)$, $Xp2(0)$ and $Xl(0)$ of the four chaotic maps (PWLCmap1, STmap, PWLCmap2 and Logistic respectively) are calculated by:

$$\begin{cases} Xp1(0) = Xp \oplus IVp1 \\ Xs(0) = Xs \oplus IVs \\ Xp2(0) = Xp \oplus IVp2 \\ Xl(0) = Xl \oplus IVl \end{cases} \quad (1.1)$$

where

$$\begin{cases} IVp1 = lsb(IV) \\ IVs = L_{cir}[lsb(IV), 3] \\ IVp2 = L_{cir}[IVs, 3] \\ IVl = L_{cir}[lsb(IV), 2] \end{cases} \quad (1.2)$$

with \oplus denotes the XOR operator, $lsb(IV)$ is the 32 least significant bits of IV and $L_{cir}[S, q]$ performs the q -bits left circular shift on the binary sequence S .

- Second, the four chaotic maps are coupled by a binary diffusion matrix to produce the future samples $Xp1(n)$, $Xs(n)$, $Xp2(n)$ and $Xl(n)$ from which the output function produces the output sequence $X(n)$, by using a chaotic switching technique.

The equation of the system is given by:

$$\begin{bmatrix} Xp1(n) \\ Xs(n) \\ Xp2(n) \\ Xl(n) \end{bmatrix} = \mathbf{D} \odot \begin{bmatrix} Fp[Xp1(n-1)] \\ Fs[Xs(n-1)] \\ Fp[Xp2(n-1)] \\ Fl[Xl(n-1)] \end{bmatrix} \quad (1.3)$$

where D is the binary diffusion matrix:

$$D = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \quad (1.4)$$

and \odot is the operator defined as we can see in the following equation :

$$\begin{bmatrix} Xp1(n) \\ Xs(n) \\ Xp2(n) \\ Xl(n) \end{bmatrix} = \begin{bmatrix} Fp[Xp1(n-1)] \oplus Fs[Xs(n-1)] \oplus Fp[Xp2(n-1)] \\ Fp[Xp1(n-1)] \oplus Fs[Xs(n-1)] \oplus Fl[Xl(n-1)] \\ Fp[Xp1(n-1)] \oplus Fp[Xp2(n-1)] \oplus Fl[Xl(n-1)] \\ Fs[Xs(n-1)] \oplus Fp[Xp2(n-1)] \oplus Fl[Xl(n-1)] \end{bmatrix} \quad (1.5)$$

The obtained samples of the sequence $X(n)$ are controlled by the chaotic switching technique, using the obtained sample $Xl(n)$ and two threshold $Th1$ and $Th2$, defined as follows:

$$X(n) = \begin{cases} Xp1(n), & \text{if } 0 < Xl(n) < Th1 \\ Xs(n), & \text{if } Th1 \leq Xl(n) < Th2 \\ Xp2(n), & \text{otherwise} \end{cases} \quad (1.6)$$

where $Th1 = 0.8 \times 2^N$ and $Th2 = 0.9 \times 2^N$.

Computing performance

The experiment is made using a two 32-bit multicore Intel Core(TM) i5 processors running at 2.60 GHz with 16 Gb of main memory. This hardware platform was used on top of an Ubuntu 14.04 Trusty Linux distribution, and the programming is performed in C code. Authors provide, for different sizes of data bytes, the average generation time in micro second $GT(\mu s)$, the average bit rate en Mega bit par second $BR(Mbit/s)$, and the average of the needed number of cycles to generate one byte, $NCpB(Cycles/B)$. The average is calculated by using 100 different secret keys. The results obtained in Table 1.1 has a little bit better computing performance than our proposed generator of Chapter 3 since our PCNG uses a recursive cell structure with one order (delay 1).

Table 1.1 – Computation performance.

Data (Bytes)	GT(μs)	BR(Mbit/s)	NCpB(Cycles/B)
64	2	171.81	121.06
128	4	212.44	97.91
256	8	255.36	81.45
512	13	306.12	67.9
1024	23	348.445	59.69
2048	46	349.11	59.58
4096	52	621.07	33.49
16384	196	666.35	31.21
32768	338	774.13	26.87
65536	654	801.61	25.95
125000	1179	804.68	25.95
196608	1600	982.54	21.17
393216	2801	1122.67	18.53
786432	4237	1484.75	14.01
3145728	16727	1504.48	13.83
12582912	66666	1509.95	13.78

Structure of the Jallouli et al. PCNG

We first present the structure of the Jallouli et al. PCNG, then we discuss its computing performance and finally present its security analysis. We report below the security analysis in terms of key size, keystream attack and key sensitivity attack.

The Key size of the PCNG composed of all initial conditions and parameters is large enough to resist the brute force attack. $|K| = 4 \times 32 + 32 + 2 \times 31 = 222$ bits.

In order to verify key sensitivity, they calculate the Hamming Distance (HD) of two sequences generated with only one bit change (lsb bit) in the parameter X_p . The calculation is performed between two sequences over 100 random secret keys.

The obtained average value of Hamming distance is equal to 0.499887. This value is close to the optimal value of 50%, which indicated the high sensitivity on one bit change in the secret key.

Mapping and Histogram

The Mapping indicates the dynamic attitude of the system. The resulting mapping of a given produced sequence appears random in comparison with a mapping nature of a known map (see Figure 1.5).

A practical PCNG must produce sequences that have uniform distribution in the whole phase space. Visually, the obtained histogram in Figure 1.5 for a given generated sequence is uniform. To confirm this result they applied the Chi-Square and they obtained 1030.832 as an experimental value, which is smaller than the theoretical value 1073.642651, then the histogram is uniform. Notice that, the uniformity of a sequence generated by our proposed PCNG of Chapter 3, is better than one produced by Jalloulli et al. PCNG. Indeed, when the experimental value of Chi-Square is smaller than the theoretical one; this means that the uniformity of the generated sequence is better.

Auto and Cross-correlation

One of good property of a PCNG is that, the generated sequences must be uncorrelated. Thus, the cross-correlation of two sequences x and y (generated with slightly different keys) must be close to zero (additional details are reported in Chapter 3). Figure 1.5 shows that the sequences produced by Jalloulli et al. PCNG are not correlated.

NIST Test

The National Institute of Standards and Technology (NIST) test is a statistical package that consists of 188 tests and sub-tests that were proposed in order to confirm the randomness of an arbitrarily long binary sequences (more details regarding NIST test are clarified in Chapter 3). Figure 1.5 gives the results for sequences generated by Jalloulli et al. PCNG. As we can see only one sub-test is not passed. In contrast, sequences in our proposed PCNG of Chapter 3 have successfully passed all the NIST tests. Therefore, the proposed chaotic generator PCNGs of Chapter 3 is robust against statistical attacks and its security performance are better than the Jallulli et al. PCNG.

1.4 Real time applications

A real-time application (RTA) is an application software that works within a time frame that the user senses as immediate or current. Thus, time expressed as a resource of fundamental concern in real-time systems, and tasks must be scheduled and executed to meet their timeliness concerns. Examples of such RTA resources are IP-telephony, XoIP, video conferencing, Video on Demand (VOD) and Audio Video on Demand(AVOD). Real-time system software's are more and more often. In this case, one of the crucial challenges is to synchronize multiple concurrent tasks. Additionally, many real-time systems play a high impact role in their environments and failure to perform correctly may result in significant costs or human risks. Thus, real-time systems must also be highly reliable (i.e., they must be deployed correctly), and available (i.e., they must work continuously) [64, 65, 66].

Furthermore, many real-time systems is that they are belong to embedded systems, i.e., they are ingredients of a larger system that contacts with the physical world. This is often the main source of complexity in

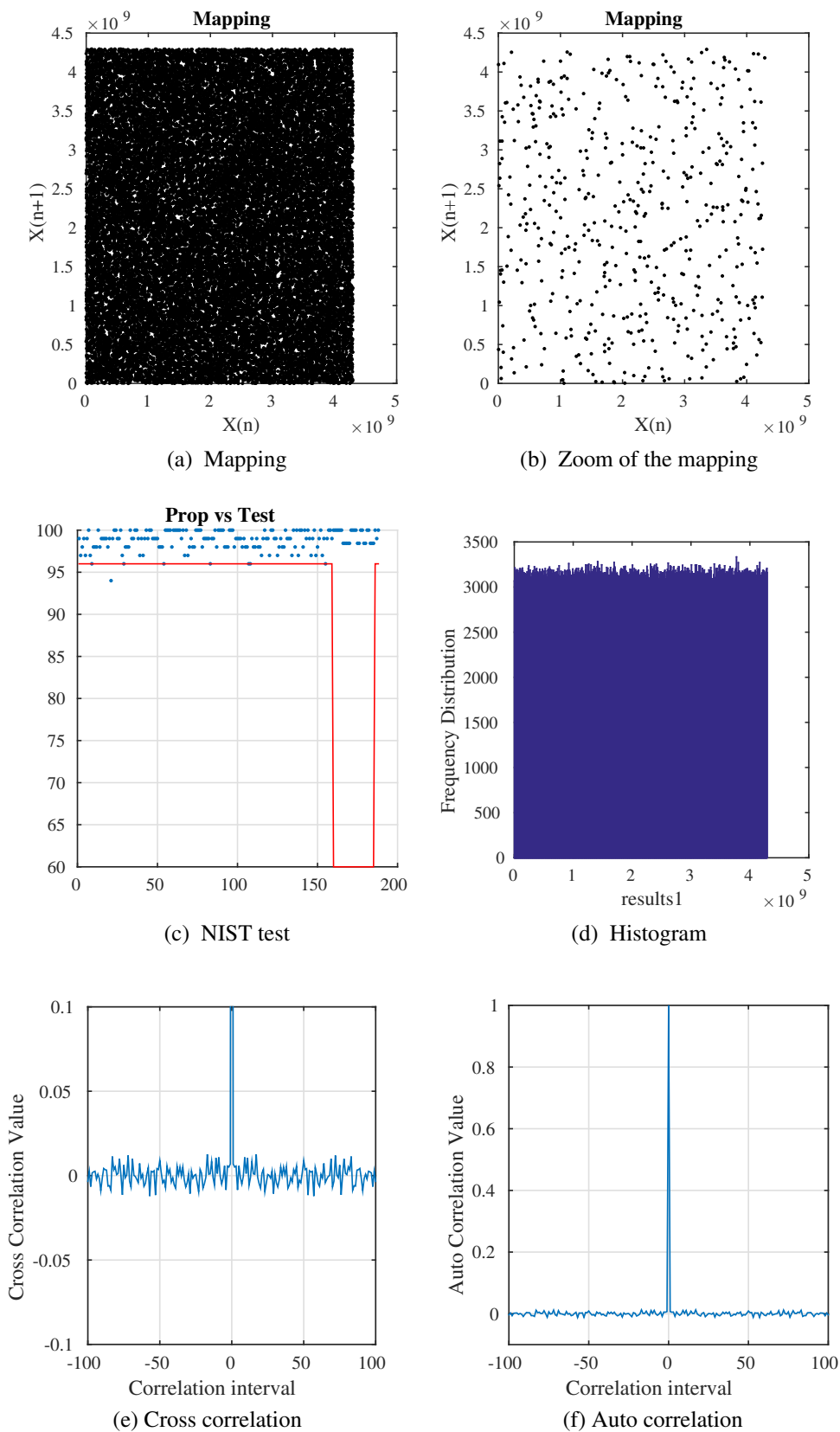


Figure 1.5 – Statistical tests of Jalloulli et al. PCNG.

real-time systems.

An embedded system is an integration between electronic and computer system designed to serve a particular purpose, like microwave ovens, washing machines, dishwashers, pacemakers and digital cameras. The most important hardware components are the processing elements that are responsible both for controlling and driving the embedded system, and for performing its computation. For complicated and intensive systems, like audio and video Digital Signal Processing (DSP) systems, specialized processors with a high computational power for a limited cost are used. The first embedded systems were composed of different hardware components, and were used only for military and space exploration projects. Progressively, the miniaturization of integrated circuits led to the integration of more and more hardware components within a single chip called a System-on-Chip (SoC). Nowadays, embedded systems are often based on heterogeneous Multiprocessor Systems-on-Chips (MPSoCs). An heterogeneous MPSoC interconnect all the elements of an embedded system. In RTA the system must perform its function within specified time limits. Moreover, it should be reactive. The system is continuously responding to events from the external environment. Multi-threading approach aims to increase utilization of a single core by using thread-level as well as instruction-level parallelism. For that reason we will use parallel and efficient implementation techniques to develop new *cryptosystems* for image/video encryption that are suitable for RTA [67, 68, 69, 70, 71].

1.5 High efficiency video coding (HEVC)

High Efficiency Video Coding (HEVC) is the last video coding standard issued by the ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG) [2, 72]. The main objective of the HEVC standardization efforts is to enable 50% bitrate reduction for similar video quality [73], compared to its predecessor H.264/AVC [74]. In the upcoming years, HEVC is expected to replace the previous video coding standards in the emerging applications, such as High Dynamic Range (HDR), Virtual Reality (VR), High Frame Rate (HFR), High Resolutions (4K, 8K), etc. In such applications, multimedia contents security and confidentiality are of paramount importance for copyright and privacy protection. Thus, a huge work has been devoted for these purposes in the last decade [19, 20, 21, 75, 76, 77, 78, 79, 80, 81]. In this section we provide a brief overview on HEVC encoder and we recall some work directed to encrypt video in HEVC codec.

1.5.1 HEVC tools

Several tools defined in the HEVC standard enable a bit-rate saving of 50%-60% with respect to the H.264/AVC. These new tools provide larger coding blocks, quad-tree block partitioning, more accurate Intra and Inter predictions, optimized entropy coding and the new in-loop *Sample Adaptive Offset (SAO)* filter. Video compression consists in removing spatial and temporal redundancies in the video and thus, it considerably decreases the required data to represent the video. The video compression processes in HEVC is shown in Figure 1.6. The HEVC frame is split into coding tree units (CTUs) of fixed sizes, from 16x16 up to 64x64. Each CTU can be recursively split in a quad-tree structure to Coding Units (CUs). CUs are the basic unit of the prediction in HEVC. These CUs are composed of three Coding Blocks CBs (one luma, and two chroma) in 4:2:0 colour format representation. Figure 1.7 demonstrates the partitioning process in HEVC. The decision to use intra or inter prediction is performed at the CU level. CUs are predicted in intra mode from reconstructed neighbouring samples in the same slice. For I slices, only intra prediction mode is used, while in P and B slices CUs can be in intra or inter prediction mode [2, 82]. In our thesis work, we focus on three used tools in HEVC, including entropy coding, Intra prediction mode and parallel tools in HEVC (tiling concept). In Chapter 4 we will use these three tools to provide a selective encryption for video contents.

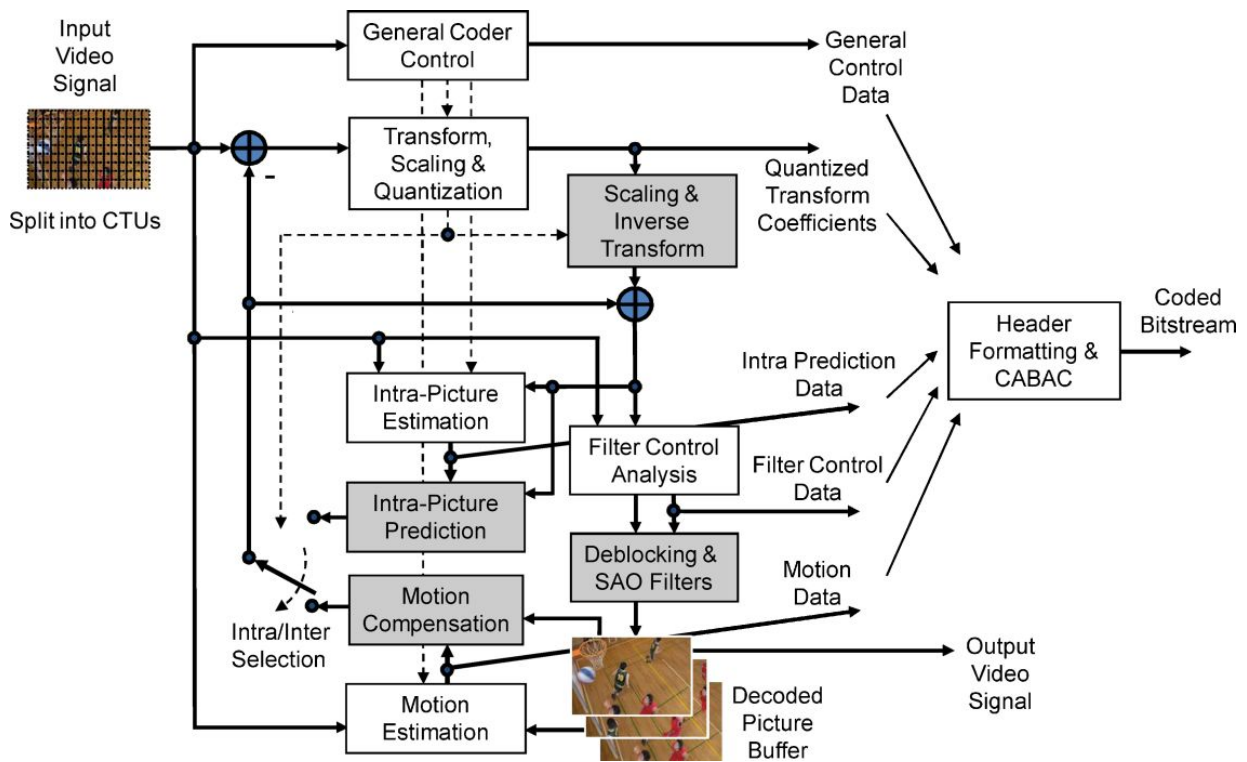


Figure 1.6 – HEVC video encoder [2].

1.5.1.1 Intra prediction

HEVC encoder enables higher compression efficiency partly by offering 35 intra prediction modes (IPM). As illustrated in Figure 1.8, these modes are composed of 33 Intra_Angular prediction modes (from 2 \rightarrow 34), the Intra_Planer mode (mode 0) and the Intra_DC prediction mode (mode 1). For an efficient coding of the 35 IPMs, a list of the Most Probable Mode (MPM) is defined in HEVC. This list of three modes is derived from the Intra prediction modes of the neighbouring blocks. Three syntax elements are used to signal the Intra prediction mode for luma prediction block in the bitstream. The first flag is signalled to determine if one of the MPM is used. In this case, the second flag (one bit for the first MPM and two bits for the two last MPMs) is signalled to indicated which of the last MPM is selected. The 32 remaining modes outside the MPM list are coded by a fixed-length 5-bin value that are bypass coded. Table 1.2 provides the coded scheme for the luma IPMs, MPM0, MPM1 and MPM2 coded by 2, 3, and 3 bits respectively, the first bit in red color is coded using a CABAC context and other bins are bypassed.

Figure 1.9 indicates the neighboring intra prediction mode of left and top prediction unit (PU), the two MPM are X and Y in case of X equal to Y, and the third MPM is planar in case of neither of X or Y is planar, otherwise is set to DC in case of neither X and Y is DC, otherwise is set to intra angular 26. In case that X equal to Y and X,Y greater than 2 (not angular), the three MPM are set to Planar, DC, 26 [2]. An adaptive scanning method is applied in the HEVC for transform coefficients, which is used with the block sizes of 4×4 and 8×8 to benefit from the statistical distribution of the active coefficients in 2-D transform blocks. Angular (6-14) modes use vertical scan, Angular (22-30) modes use Horizontal scan, and the other modes refer to diagonal scanning (see Figure 1.10). The derivation process of chroma IPMs may come from the luma one, as clarified in Table 1.3 [2, 83].

1.5.1.2 Inter prediction

Inter prediction uses available reconstructed pictures before as an indicator for motion compensation which is the key tool to represent the video contents. Using the block-wise displacements between

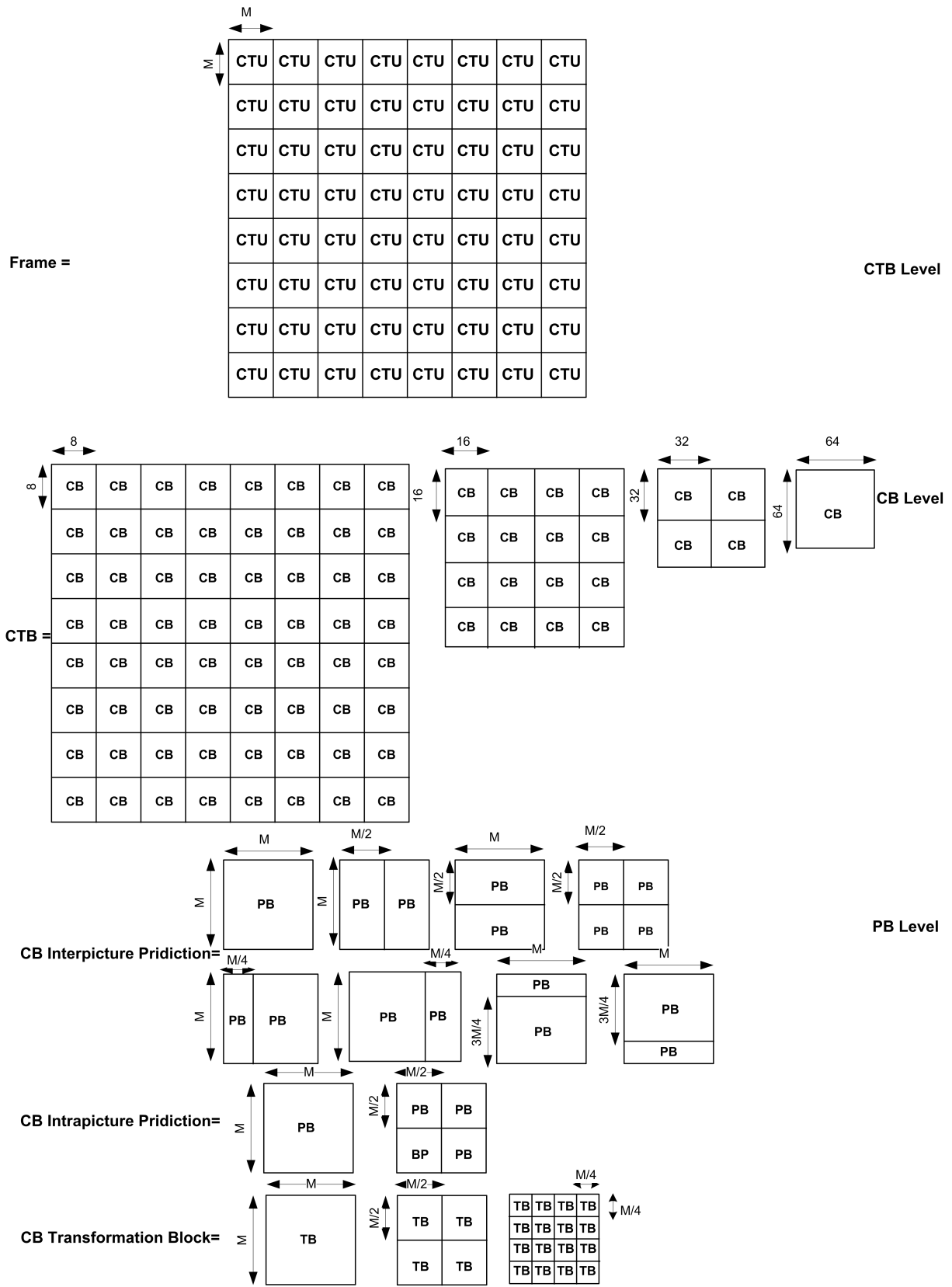


Figure 1.7 – HEVC partitioning [2].

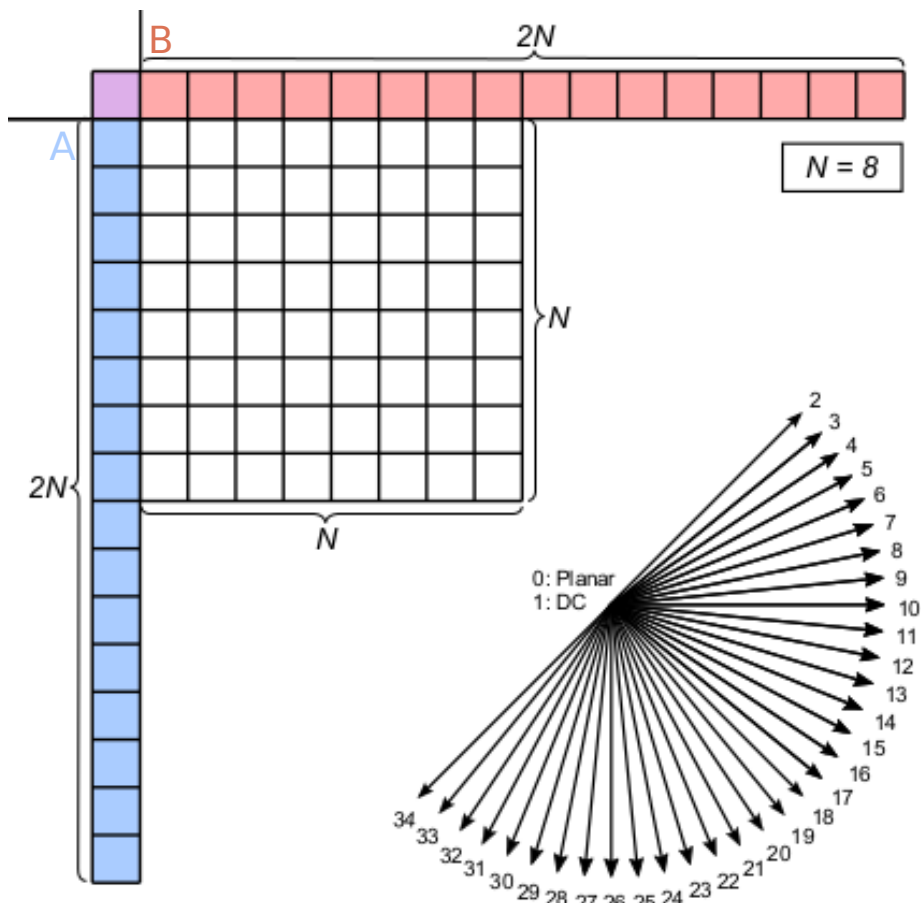


Figure 1.8 – Intra prediction modes on HEVC.

Table 1.2 – Used coding scheme for IPMs in HEVC. The first bit is coded using a CABAC context.

Number of bits	Code	Coded Mode
2	10	MPM0
3	110	MPM1
3	111	MPM2
6	000000 ⋮ 011111	32 remaining IPMs

positions in the current picture and previously encoded one we can determine the changes between successive pictures. Motion estimation is the process of finding the best match choices between the current Prediction Block (PB) and an area in previous or following frames. The encoder may choose to do this in order to predict the same picture with different weights (weighted prediction). HEVC uses candidate list indexing. Motion Vector MV is a vector which specifies the moving direction of the predicted block, by calculating the difference between the current PB and the reference one. There are two Motion Vector (MV prediction modes): Merge and AMVP (advanced motion vector prediction) [2].

1.5.1.3 Transformation and Quantization

Transformation and quantization on HEVC are determined using fixed-point integer operations with output and intermediate values not being more than 16-bit word length. HEVC have four

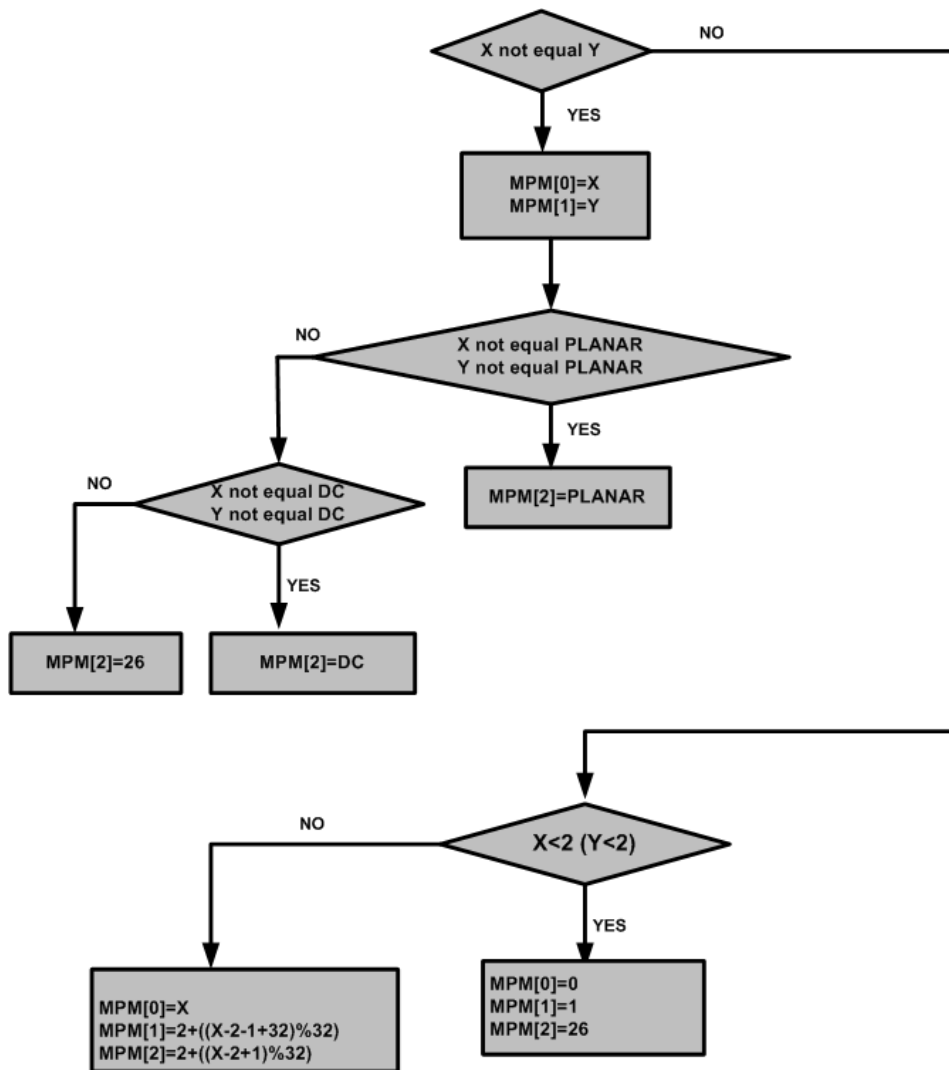


Figure 1.9 – Derivation process of the three most probable modes.

Table 1.3 – Derivation process of the chroma intra prediction mode

Intra_chroma_pred_mode	Luma Intra Pred Mode			
	0	26	10	1
<i>Planar (i.e., mode-0)</i>	34	0	0	0
<i>Angular (i.e., mode-26)</i>	26	34	26	26
<i>Angular(i.e., mode-10)</i>	10	10	34	10
<i>DC (i.e., mode-1)</i>	1	1	1	34
<i>Derived (i.e., use the luma mode)</i>	0	26	10	1

transform sizes: 4x4, 8x8, 16x16 and 32x32. Like AVC, the transforms are integer transforms based on the Discrete Cosine Transform (DCT). The derivation process of the 4 × 4 luma intra-prediction is based on the Discrete Sine Transform (DST). The basis matrix uses coefficients requiring seven bits storage, so it is more accurate than AVC. HEVC introduces several new features and tools for the transform coefficient coding to help improve upon H.264/AVC: dependent coefficient scanning, last significant

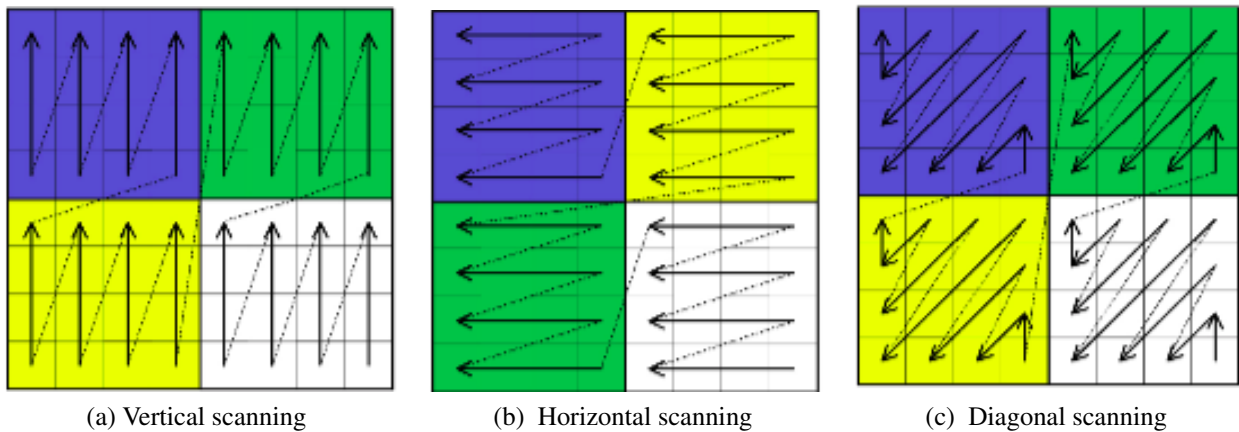


Figure 1.10 – Scanning mode on the HEVC .

coefficient coding, multilevel significance maps, improved significance flag context modeling, and sign data hiding. HEVC followed a development process in which it was iteratively refined to improve coding efficiency and suitability for hardware and software implementation. The quantization process in the HEVC uses the same scheme as the one used in the AVC: Uniform Reconstruction Quantization (URQ) scheme. This scheme is controlled by the Quantization Parameter (QP) [2].

1.5.1.4 HEVC entropy coding

HEVC performs entropy coding using *Context based adaptive binary arithmetic (CABAC)*. The CABAC mechanism consists of three main functions: binarization, context modeling and arithmetic coding [84]. The binarization function in first step converts syntax elements to binary symbols (bin). Subsequently, the context modeling updates the probabilities of bins, and finally the arithmetic coding compresses the bins into bits according to the estimated probabilities. Five binarization methods are used in HEVC: Unary (U), Truncated Unary (TU), Fixed Length (FL), Truncated Rice code with an adaptive context p (TRp) and the k th-order Exp-Golomb (EGk) codes. The arithmetic coder can be performed either by a context coded which is the estimated probability of a syntax element or by bypass coded that considers equal probability of 0.5 for each bin.

The three main functions of the CABAC are shown in Figure 1.11. As illustrated in this figure, the format compliant selective encryption (encrypt the sensitive information of the video contents), is performed between binarization and arithmetic coding.

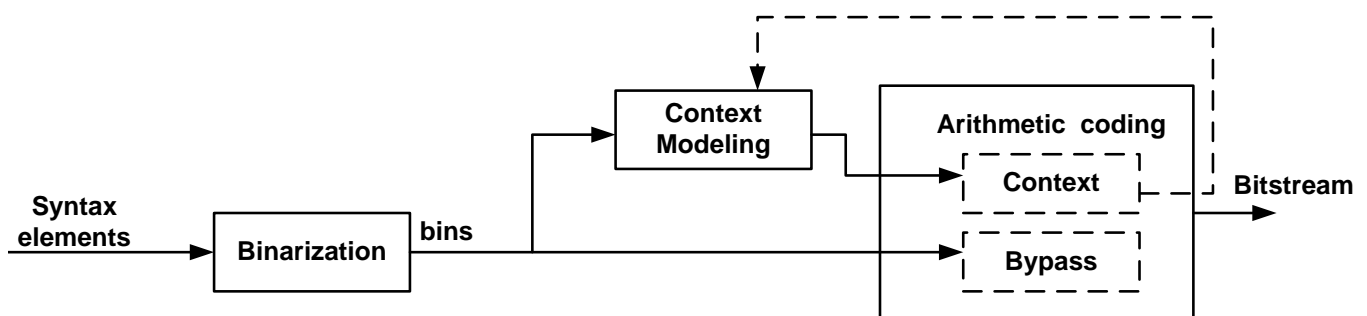


Figure 1.11 – Main functions of CABAC engine.

1.5.1.5 Parallel tools in HEVC

HEVC defines two new tools, enabling parallel encoding/decoding of a single picture, namely *Tiles* and *Wavefront* [84, 85, 86]. In the HEVC standard, the picture can be split into different tiles where each consists of an integer number of separately decodable Code Tree Blocks (CTBs). This concept constraints Motion Vectors (MVs) and intra prediction inside the tile boundaries. The CABAC context is initialized at the beginning of each tile. This new feature, introduced in HEVC standard, offers a flexible classification of CTUs, a preferable correlation of pixels compared to slice and a superior coding efficiency as tiles do not contain header information. Tiles provide better rate distortion performance in case of high parallelism levels. HEVC supports wavefront parallel processing (WPP) which enables parallelization of arithmetic entropy encode/decode within a frame. With WPP, each CTB row can be decoded in parallel. Figure 1.12 depicts the methodology that used by WPP.

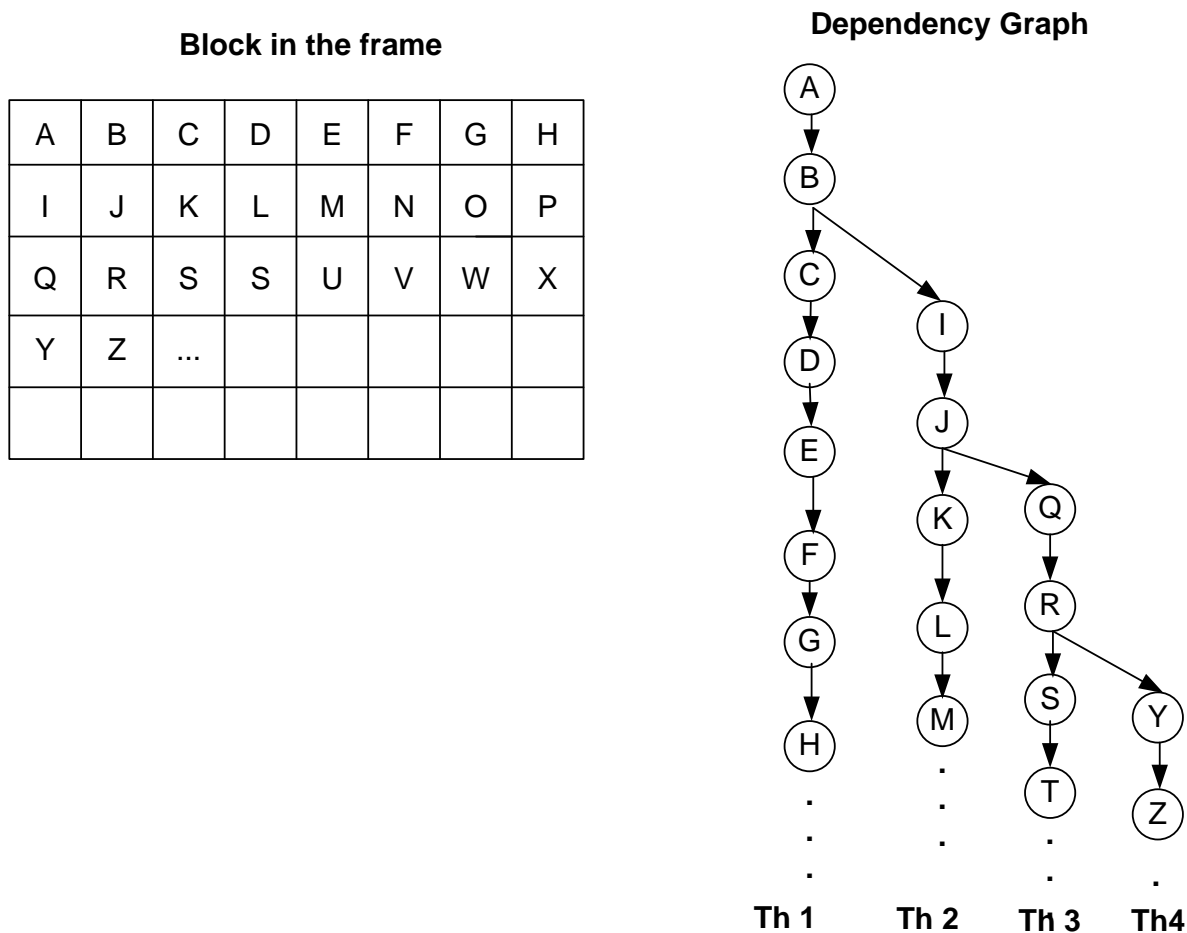


Figure 1.12 – Wavefront parallel processing (WPP).

1.6 Selective video encryption related works

For the time being, a couple of encryption algorithms have been proposed for HEVC video. Shahid et al. [19] proposed a selective encryption scheme by joint encryption and compression system lies on *Context Adaptive Binary Arithmetic Coding (CABAC)* bin string. Hamidouche et.al [75] proposed a fast and secure selective chaos-based crypto-compression system for HEVC and the scalable version of HEVC (SHVC).

Boyadjis et al. [76] proposed an extended selective encryption method for H.264/AVC and HEVC streams. Their approach tackles the main security challenges of selective encryption. The contribution in [76] is the improvement of visual distortion induced by selective encryption.

Schwarz et al. [88] handled an overview of the Scalable Video Coding (SVC) Extension of the H.264/AVC Standard and its functionalities that provide enhancements to transmission and storage applications. SVC has achieved significant improvements in coding efficiency with an increased degree of supported scalability relative to the scalable profiles of prior video coding standards.

Lui et al. [89] presented a chaos-based selective encryption scheme implemented on the H.264/AVC standard. The scheme deploys two Rényi chaotic maps to generate a pseudorandom key sequence which is used to hide the selected H.264/AVC syntax elements. It provides sufficient protection against full reconstruction while keeping the format compliance property so as not to cause decoding error without the key.

Lei et al. [22] have introduced a selective encryption scheme to encrypt the CABAC bit stream using a chaotic stream cipher based on the discrete Piece-Wise Linear Chaotic Map (PWLCM). For encryption scheme, each binarization process has a specific encryption and decryption operation. It is a format compliant, but it affects the compression ratio and the bit rate since it encrypts the Unary Code (UC), Truncated Unary code (TU), and Fixed Length Code (FLC). As a result, not all parameters during the binarization process can be encrypted while preserving the format compliance property.

Xu et al. [91] proposed an encryption selection control module to encrypt video syntax elements dynamically which is controlled by the chaotic pseudo-random sequence. A novel spatiotemporal chaos system and binarization method are used to generate a key stream for encrypting the chosen syntax elements.

Several works proposed the encryption of ROI in the video. Peng et al. [77] presented an encryption scheme for ROI of H.264 video based on *flexible macroblock ordering (FMO)* and chaos, where the ROI was the human face areas. Dufaux et al. [20] proposed an effective approach to encrypt ROI based on code stream-domain encryption. Work in [21] enables rectangular region privacy by de-identifying faces. This solution guarantees that face recognition software cannot reliably recognize de-identified faces even though part of the facial details are preserved. In [78] the authors investigated the privacy protection in the H.264/SVC (Scalable Video Coding). This solution detects face regions (ROI) first and then encrypts these ROI in the transform domain by scrambling the sign of the non-zero TCs at all SVC layers.

1.7 Conclusion

In this chapter an overview of existing chaos-based generator and chaos based block/stream cipher algorithms in the literature has been presented. A concise description of real-time applications (RTA) and embedded systems was reported. A brief description of HEVC video coding has been also provided. Intra/Inter prediction tools on HEVC with entropy coding using CABAC was described. A state-of-the-art of selective encryption and ROI encryption on HEVC was reported. In next chapter, we handle the programming techniques and the software security tools that were used in our thesis work.

Parallel Programming and Software Security Tools

2.1 Introduction

The need for high quality and high speed systems is one of the most demanding aspects when using the Internet, mainly for Real-Time Applications (RTA). Real-Time Applications services like IPTelephony, XoIP, Videoconferencing, Video on Demand (VOD), Audio Video on Demand (AVOD) and others have become a successful business on the Internet; several business organizations provide RTA services and make big business out of it. At the same time, the security of the software produced must be assured since the number of threats specifically targeting software is increasing. At the hardware level, the new proliferation of ever more powerful computers at an increasing rate is undeniably challenging. In this context, many-core processors offer new possibilities to high performance computing. To exploit all the potential of these processors, new programming techniques are needed to spread tasks onto as many processors as possible, to increase the speedup of applications. Among them, parallel programming techniques have been drawing much attention during the past few years.

In this chapter we review some of the existing parallel programming methods and tools pointing out those used in our thesis. Software security elements are also presented. We review both static and dynamic tools designed to reduce software vulnerabilities.

2.2 Parallel programming models

Processors' speeds cannot be significantly increased anymore (the higher the clock speed the more heat is generated) because processors manufacturers can no longer cool the processors fast enough. Hence, multicore systems have become more popular. In order to benefit from these systems, programmers turned to parallel programming. Parallelism is achieved thanks to multiple processes running at the same time on multiple processors [92]. It explicitly breaks the task down into small units of execution, where each unit can be executed in parallel on a single processor. Thus, multiple parts of the same task can run in parallel [93]. Parallel programming can be implemented using several different software interfaces, or parallel programming models. The programming model used in any application depends on the underlying hardware architecture of the system on which the application is expected to run: *shared memory* or *distributed memory* architecture. In *shared-memory* multiprocessor architectures, threads can be used to implement parallelism. Threads are lightweight processes, that exist within a single operating system process. The

threads share the same memory address space and state information of the process that contains them. Parallel programming can be implemented for shared memory systems using *automatic parallelization* [94], *POSIX threads* [95], *Solaris threads* [95], or *OpenMP* [96]. Among the *distributed memory* programming models, the *Message Passing Interface (MPI)* model [97] is commonly used to parallelize applications. With the emergence of multi-core systems, hybrid programming models have also been developed. Within a single node, fast communication through shared memory can be exploited, and a networking protocol can be used to communicate across the nodes. Programs can then take advantage of both the shared-memory and the distributed-memory modes. Figure 2.1 and Figure 2.2 illustrate the shared and distributed memory systems [98] respectively. The former is the most widely used. Typically, the cores have special L1 caches, while other caches, of higher levels (i.e. L2 and L3) may or may not be shared between the cores. The adopted distributed-memory systems are called clusters [99]. They consist of a collection of systems

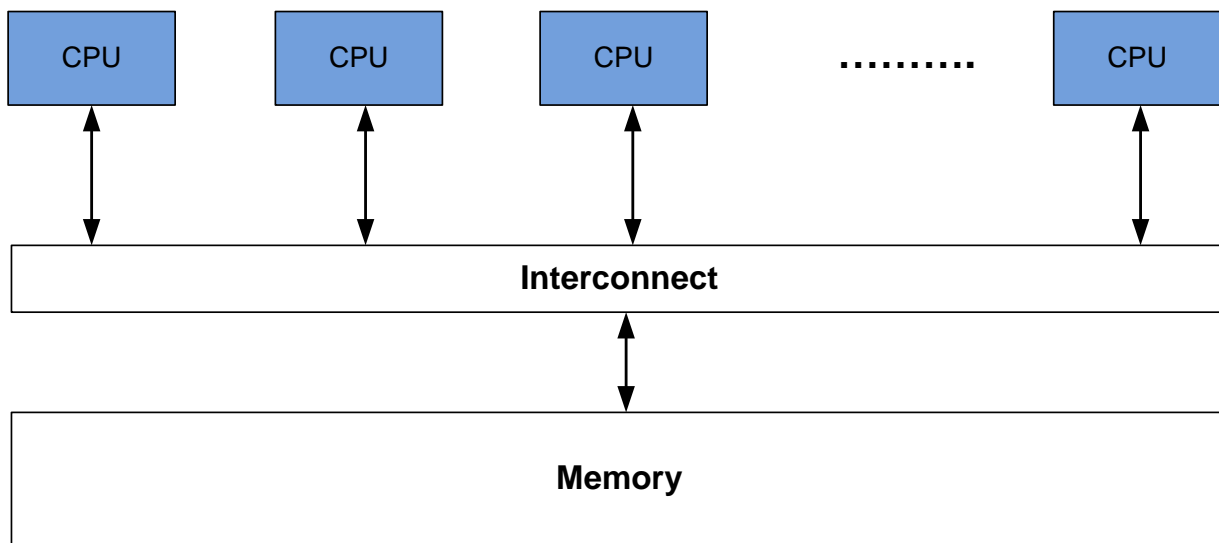


Figure 2.1 – Shared memory system

for example, PCs interconnected by a commodity interconnection network such as the Ethernet. A shared memory program achieves its parallelism through *threading*. A *process* is an executing instance of an application that can contain one or more thread. A *thread* is defined as an independent stream of instructions that can be scheduled to run on a given processor. In UNIX environment [100], as shown in Figure 2.3, a thread exists within a process and uses the process resources. The thread has its own independent flow of control as long as its parent process exists and the operating system supports it. Threads within the same process share resources, therefore modifications made by one thread to the shared system resources (such as data variables or files) will be seen by all other threads. In the following three sub sections, we outline three existing methods that are used to make a program parallel: *MPI*, *OpenMP* and *Pthread*. Note that the last two have been considered in this thesis to parallel the sequential version of the proposed chaotic generator.

2.2.1 MPI

MPI stands for Message-Passing Interface [101]. It is a very explicit programming model. The programmer implements the distribution of the tasks, the communication between them, and decides how the work is to be allocated between the various threads. MPI is not a new programming language, but defines a library of functions that can be called from the C, C++, and Fortran programs. MPI is a robust and flexible application programming interface (API) for developing parallel programs. All of the programs that are executed on the most powerful computers use message-passing. MPI is a low-level language. Thus, there is a large amount of details that the programmer needs to know. In message-passing programs, a program running on one core is usually called a process, and two processes can communicate by calling functions: one

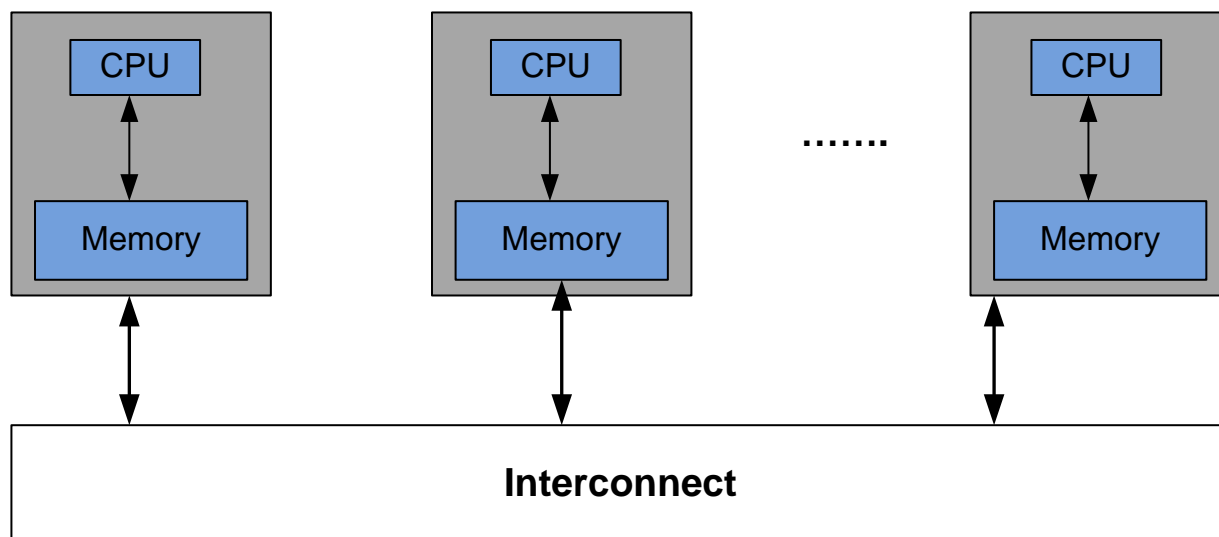


Figure 2.2 – Distributed memory system

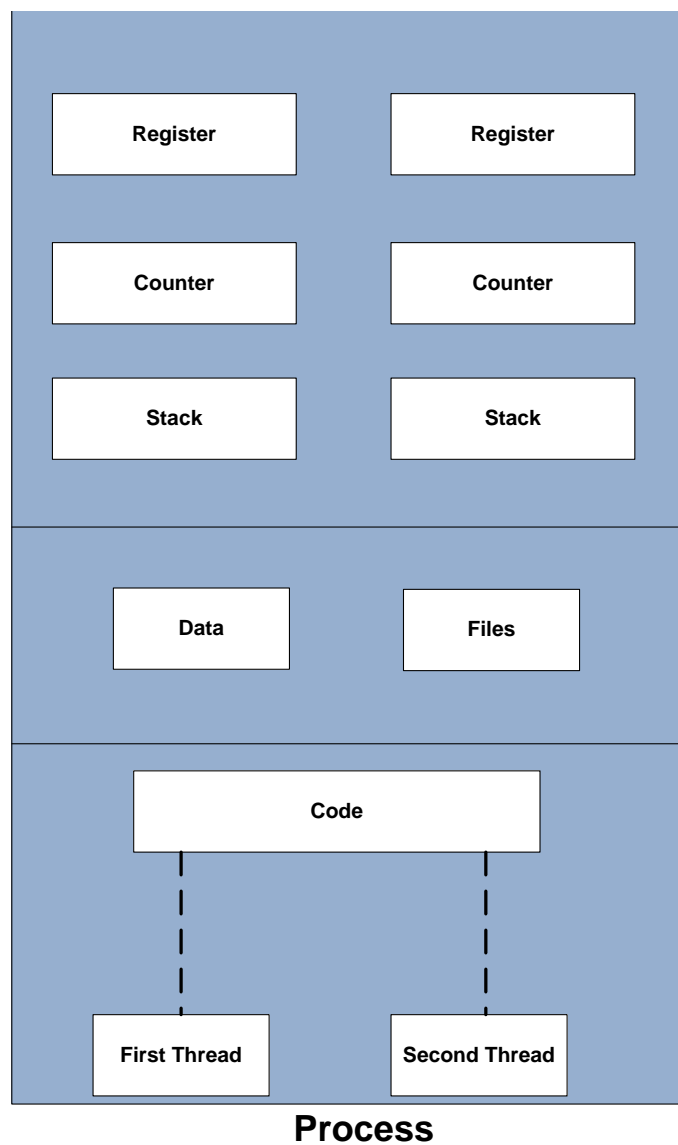


Figure 2.3 – Threads in a Unix process

process performs a `send` operation while the other performs a `receive` operation. The details of compiling and running the program depend on the system. For compiling and linking MPI programs `mpicc` command is used. Indeed, `mpicc` is a shell script that is associated with the C compiler, namely a *wrapper*. It simplifies the running of the compiler by telling it where to find the necessary header files and which libraries to link with the object file [97, 102, 103].

2.2.2 OpenMP

OpenMP is an API for writing multithreaded applications [96]. It is designed for multi-processor or multi-core, shared memory machines. It is made of a set of compiler directives, library routines and environment variables for parallel application programmers. OpenMP provides the capabilities to incrementally parallelize a sequential program, unlike message-passing libraries which typically require an all-or-nothing approach (the user can specify the number of processes that should be started). Most major platforms have been implemented including Unix/Linux platforms and Windows. They currently support programs in Fortran, C and C++. In order to use the OpenMP function prototypes and types, one must include the following header file: `#include <omp.h>`. For compiling and linking a program using OpenMP, the additional flag `fopenmp` must be included in the `gcc` [98, 104, 105].

2.2.3 Pthread

Pthread is a library of functions that programmers can use to implement parallel programs [106]. Unlike the MPI, Pthread is used to implement shared-memory parallelism. Pthread is not a programming language (such as C or Java). It is a library that can be linked with C programs. For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard. Implementations that comply with this standard are referred to as *POSIX threads*, or *Pthreads* [98, 106, 107, 108, 109]. Programs must be compiled with the `-lpthread` directive. In the following sections we will comprehensively study the fundamentals of the OpenMP and Pthread-based multi-threading programming approaches.

2.3 A brief introduction to OpenMP

2.3.1 OpenMP parallel principles

OpenMP programs accomplish parallelism exclusively through the use of threads. OpenMP uses the fork-join model of parallel execution (see Figure 2.4): Programs begin as a single process called the Master thread. The Master thread executes in sequential mode until the parallel region construct is encountered. The Master thread then creates a team of parallel threads (`fork`) that simultaneously execute statements in the parallel region. After executing the statements in the parallel region, the team threads synchronize and terminate (`join`) and the Master thread can continue its execution.

2.3.2 OpenMP memory architecture

As depicted in Figure 2.5, the different threads can manipulate two kinds of data: *shared* data and/or *private* ones.

1. *shared data*: All threads can access data in shared memory. Shared variables exist in only one memory location and all threads can read or write to that address.
2. *private data*: The data can only be accessed by threads that own it.

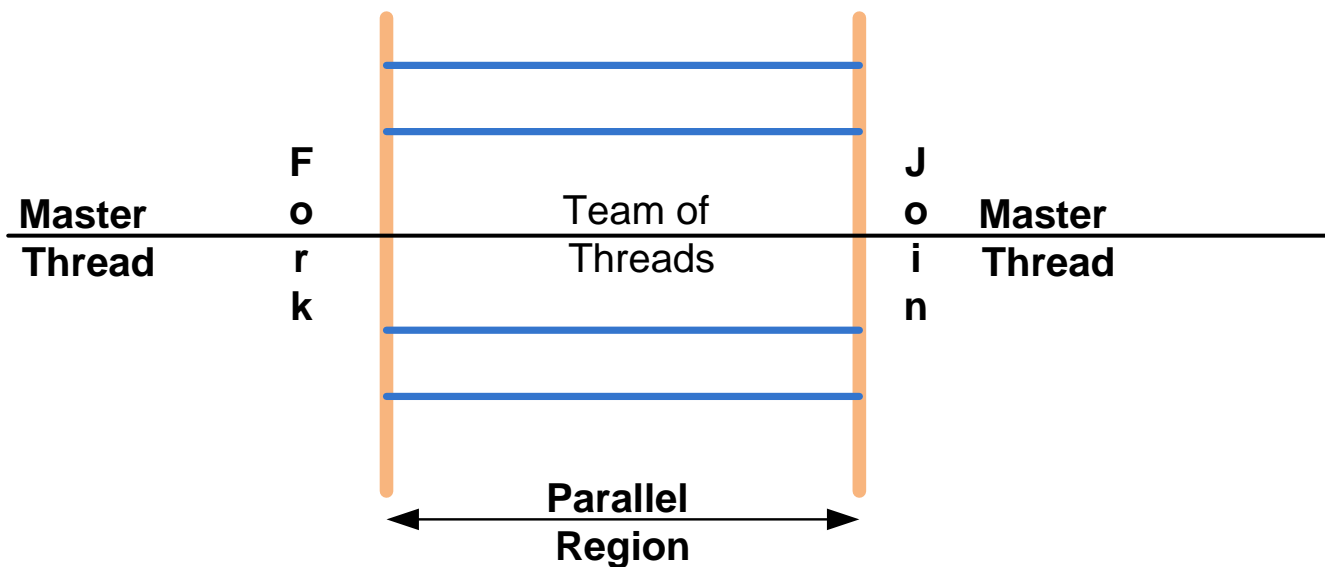


Figure 2.4 – The fork-join model [110]

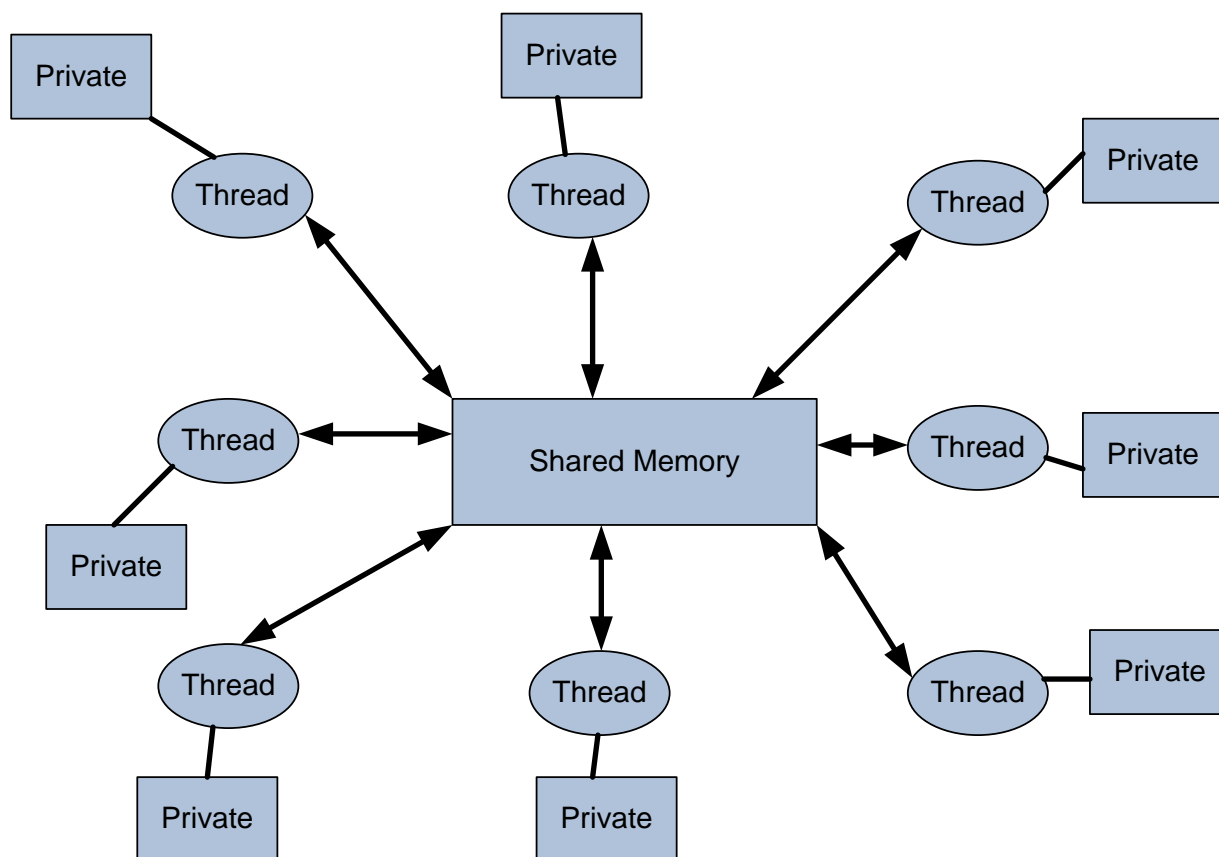


Figure 2.5 – Shared vs. Private data

2.3.3 OpenMP syntax

It is the responsibility of the developer to introduce OpenMP directives in his code. OpenMP directives are comments in source code that specify parallelism for shared memory machines. An *OpenMP directive* has the following general form :

sentinel directive-name [clause [clause]...]

The `sentinel` is a string of characters whose value depends on the used language. `Clauses` control the behavior of an OpenMP directive (e.g. data scoping, schedule, initialization, number of threads used, etc.). The following sample shows how to set the number of threads and define a parallel region. By default, the number of threads is equal to the number of logical processors on the machine. For example, if you have a machine with one physical processor that has hyperthreading enabled, it will have two logical processors and, therefore, two threads.

```
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel num_threads(4)
    {
        int i = omp_get_thread_num();
        printf("Hello from thread %d\n", i);
    }
}
```

2.3.4 OpenMP parallel constructs

2.3.4.1 Parallel regions

A *parallel region* is a block of code that will be executed simultaneously by multiple threads. When a thread reaches a `parallel` directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team. Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code. There is an implicit barrier at the end of a parallel section. Only the master thread continues execution past this point.

Example :

```
#pragma omp parallel
{
    //code block
}
```

2.3.4.2 Iterative work-sharing constructs (the loop for)

A *work-sharing construct* divides the execution of the enclosed code region among the members of the team that encounter it. A team of threads is formed (i.e. a parallel region). Loop iterations are split among threads with an implicit barrier. Each loop iteration must be independent of other iterations. The `DO / FOR` directive specifies that the iterations of the loop immediately following must be executed in parallel by the team. This assumes that a parallel region has already been initiated, otherwise it executes in serial on a single processor.

```
#pragma parallel for
for (i=0; i<N; i++)
{
//code block
}
```

2.3.4.3 Non-iterative work-sharing constructs (the sections)

A *non-iterative work-sharing construct* divides the execution of the enclosed code region among several threads. One thread is working on each section. Each section is executed once by a thread in the team. There is an implicit barrier at the end of a `section's` directive. Independent section directives are nested within a `sections` directive. An example of this code is given in the following piece of code.

```
#pragma omp sections
{
#pragma omp section
{
code_1 ();
}
#pragma omp section
{
code_2 ();
}
}
```

2.3.5 OpenMP data environment

2.3.5.1 The basic data scoping

Several directives accept clauses that allow a user to control the scope attributes of variables. If no data scope clauses are specified for a directive, the default scope for variables affected by the directive is shared except for-loop indexes that are private.

The basic data scoping controls in OpenMP consist of the `shared` and the `private` clauses.

The **shared** clause:

The `shared` clause specifies that variables will be shared by all the threads in a team, meaning that all threads access the same storage area for shared data.

The **private** clause:

The variables specified in a `private` list are private to each thread. When an assignment to a private variable occurs, each thread assigns it to its local copy of the variable. Variables declared `private` in a parallel region are undefined upon entry to the parallel region. If the first use of a private variable within the parallel region is in a right-hand-side expression, the results of the expression will be undefined (i.e. this is probably a coding error, see in Section 2.3.5.2 how the `firstprivate` clause can solve this problem). Likewise, variables declared `private` in a parallel region are undefined when serial execution resumes at the end of the parallel region. In this case the `lastprivate` clause can solve this problem.

Let us consider the following example :

```
#pragma omp parallel for shared(a,b,c,n) private(temp,i)
for (i=0; i<n; i++){
temp = a[i] / b[i];
c[i] = temp + cos(temp);
}
```

The defined variables `a`, `b`, `c` and `n` are shared among all threads, while `temp` and `i` are private variables locally used by each thread.

2.3.5.2 Advanced data scoping clauses

Advanced data scoping clauses can be used to more accurately control the sharing behavior of variables within a parallel construct. These include `firstprivate`, `lastprivate`, `reduction`, `threadprivate`, and `copyin` clauses.

The `firstprivate` clause:

Considering a variable `var`, a `firstprivate(var)` would initialize `var` with the value the variable had before entering the parallel construct.

In the following example, without using the `firstprivate` clause, two different runs will give two different results for all threads. This clearly demonstrates that the value of `i` is random (not initialized) inside the parallel region and that any modifications to it are not visible after the parallel region (i.e. the variable keeps its value from before entering the region). If `i` is made `firstprivate`, then it is initialized with the value that it has before the parallel region.

```
#include <stdio.h>
#include <omp.h>
int main (void)
{ int i = 10;
  #pragma omp parallel private(i)
  { printf("thread %d: i = %d\n", omp_get_thread_num(), i);
    i = 1000 + omp_get_thread_num();
  }
  printf("i = %d\n", i);
  return 0;
}
```

The `lastprivate` clause: Considering a variable `var`, a `lastprivate(var)` clause will copy the last thread loop (stack) value of `var` to the (global) `var` storage when the parallel loop is completed. In other words, the thread that executes the last iteration on section, updates the clause of the variable. In the example provided below the `lastprivate` clause allows to transfer the `i` value from the parallel region to the outside context.

```
pragma omp parallel
{ #pragma omp for lastprivate(i)
  for (i=0; i<n-1; i++)
    a[i] = b[i] + b[i+1];
}
a[i]=b[i];
```

The `reduction` clause:

A variable that accumulates the result is called a `reduction` variable. In parallel loops `reduction` operators and variables must be declared. Here is an example:

```
float sum, prod;
sum = 0; prod = 1;
#pragma omp parallel for reduction(+:sum) reduction(*:prod)
for (i=0; i<n; i++){
sum = sum + a[i];
prod = prod * a[i];
}
```

Each thread has a private `sum` and `prod`, initialized to operators identity.

Copyin clause:

The most common way to initialize `threadprivate` variables on entry to a parallel region is by using the `copyin` clause. This copies the value in the master thread's copy of the thread private variable to the copies in all the other threads in the team. In the following example the value of the variable `i` within the parallel area will be 1.

```
int i ;
#pragma omp threadprivate(i)
i=1 ;
#pragma omp parallel copyin(i)
{
printf(" Parallel value%i\n", i) ;
}
```

2.3.6 Critical sections

Synchronization methods are usually used to impose order constraints as well as to protect access to shared data. To address these two issues *implicit barriers* and *non-implicit barriers* are supported by OpenMP.

2.3.6.1 Implicit and non-implicit barriers

By default, there is an implicit barrier at the end of a parallel region. When all threads have completed the execution of the parallel region, a single thread continues the statements that follow. This implicit barrier can be removed with the `nowait` clause. Depending upon situations, this behaviour may be beneficial, because it can make full use of available resources and reduce the amount of time that threads are idle.

The barrier directive

This directive synchronizes all the threads in a team. When encountered, each thread waits until all the others of that team have reached this point.

An example concerning implicit and non-implicit barriers is given in Table 2.1.

2.3.6.2 The critical directive

The `critical` directive restricts access to the enclosed code to only one thread at a time. A thread waits at the beginning of a critical region until no other thread is executing the critical region. In Figure 2.6, a critical directive restricts access to the enclosed code to only one thread at a time. When a thread enters a critical section, it is guaranteed to see all modifications made by all the threads that had entered the critical section earlier. *Mutual Exclusion* is a property of concurrency control, which is instituted for the purpose of preventing race conditions; it is the requirement that one thread of execution never enters its critical section at the same time that another concurrent thread of execution enters its own critical section.

```
int x=0;
#pragma omp parallel
#pragma omp critical
{
    x=x+1;
}
```

Table 2.1 – OpenMP synchronization barriers

Code
<pre> #pragma omp parallel shared (A, B, C) private(id) { id=omp_get_thread_num(); A[id] = funct1(id); #pragma omp barrier // Each thread waits until all threads arrive #pragma omp for for(i=0;i<N;i++){ C[i]=funct3(i,A); } // Implicit barrier #pragma omp for nowait for(i=0;i<N;i++){ B[i]=funct2(C, i); } // No implicit barrier due to nowait A[id] = funct4(id); } </pre>

2.3.7 Clauses/directives summary

In Table 2.2 we reported a summary for Clauses/Directives in OpenMP. It summarizes which clauses are accepted by which OpenMP directives. For example the `private` clause works with all OpenMP directives, while the `NOWAIT` clause works only with `DO/FOR` and `SECTIONS` directives.

2.3.8 OpenMP problems

Users choosing to implement a parallel operation with OpenMP must really understand how threading works. In particular, the question of how global and local memory spaces are handled by the language is difficult for unexperienced people to understand. More generally, OpenMP is difficult to program, deploy, modify and maintain [111].

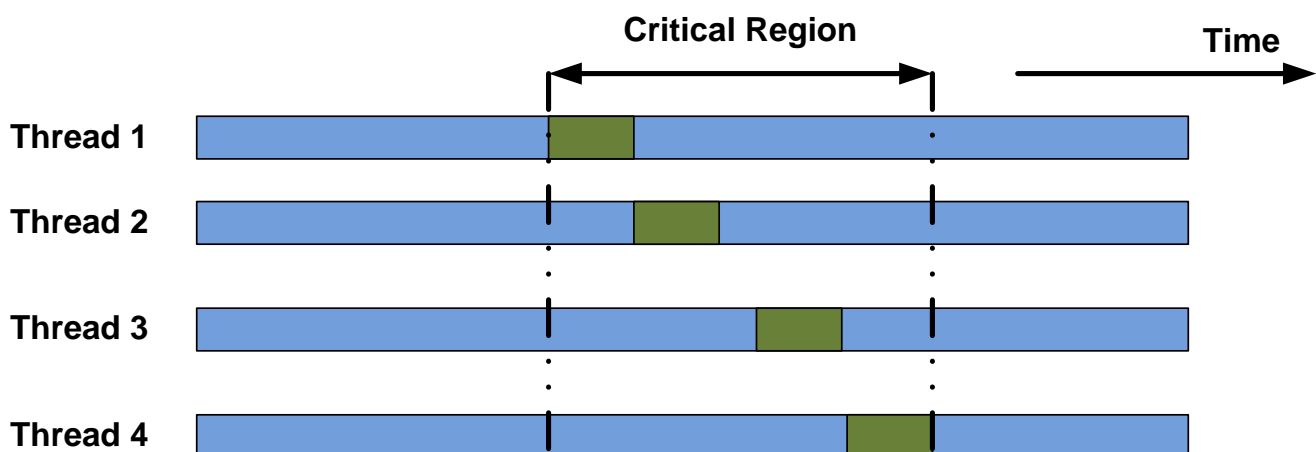


Figure 2.6 – A critical region over time

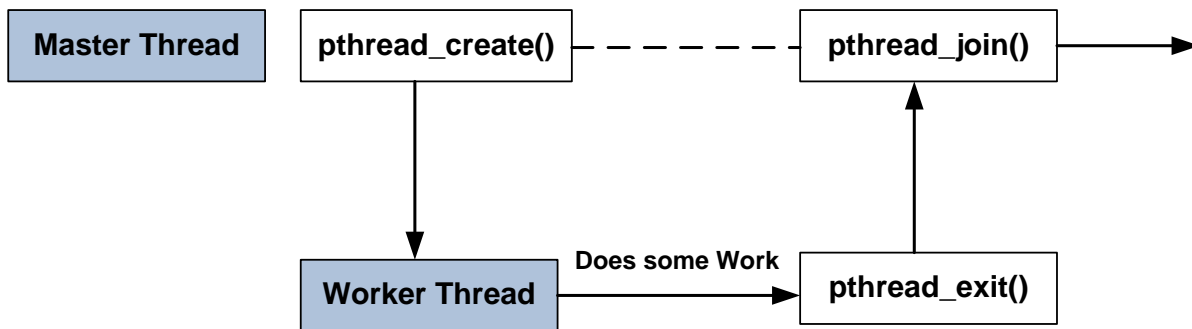


Figure 2.7 – Pthread creation and joining

```
void *(*start_routine)(void*), void *restrict arg);
```

This function can be called any number of times from anywhere within the code. In the following instance we provide the description of the function arguments:

thread: A pointer to an object of type `pthread_t` which is also a unique identifier for the new thread returned by the subroutine;

attr: A pointer to a previously allocated and initialized attribute object of type `pthread_attr_t` that may be used to set more desired thread attributes. `NULL` can be passed as a default value ;

start_routine(): The function that the using thread will execute once it is created;

arg: A single argument that may be passed to **start_routine()**. It must be passed by reference as a pointer cast of type `void`. `NULL` may be used if no argument is to be passed. In Table 2.4 we provide a simple pthread example. First, from the `main()` function, the `pthread_create()` function starts a new thread in the calling process. The new thread starts its execution by invoking `threadFunc()`; `arg` is passed as the sole argument of `threadFunc()`. `pthread_join()` function waits for the thread specified by an `idthread` to terminate. If that thread has already terminated, then `pthread_join()` returns immediately.

2.4.3 Waiting for Pthreads to finish

Waiting for threads to finish is performed by calling the `pthread_join()` function. Joining ensures that when the program exits, all threads have completed their execution. This is quite similar to the `wait()` call used for child processes. The definition of this function is:

```
int pthread_join(pthread_t id, void** status);
```

where `id` is the id of the master thread wishes to join. `status` will hold the value returned by the thread waiting it. `pthread_join()` subroutine blocks the calling thread until the specified thread `id` terminates. Figure 2.7 shows how the pthread creation and joining process are organized.

2.4.4 Pthreads mutexes

Mutex is an abbreviation for "mutual exclusion". Mutex variables are one of the important ways of implementing thread synchronization to ensure the coherence of shared data when concurrent thread read/write operations occur. A mutex variable works like a "lock" protecting access to a shared data resource. The main concept of a mutex as used in Pthreads is that only one thread can own (i.e lock) a mutex variable at any specified time. Thus, even if several threads try to lock a mutex, only one thread will be successful. No other thread can own a mutex until the owning thread unlocks that mutex. Threads must "take turns"

Table 2.4 – Pthread simple code example

Simple example
<pre> /* Simple Pthreads example. */ 1.#include <stdlib.h> 2.#include <stdio.h> 3.#include <pthread.h> 4.void* threadFunc(void* arg) 5.{ 6. int i, n; 7. /* Get the value of the argument passed in. */ 8. n = (int) arg; 9.for (i = 0; i < n; i++) 10. printf("Loop %d: Mohammed!\n", i + 1); 11. pthread_exit(NULL) ; 12. } 13. int main(int argc, char argv[]) 14. { 15. thread_t* idThread; 16. puts("Lets create a thread!"); 17. pthread_create(&idThread, NULL, threadFunc, (void*) 5); 18. pthread_join(idThread, NULL); 19. return 0; 20.} This will produce the following output: Lets create a thread! Loop 1: Mohammed! Loop 2: Mohammed! Loop 3: Mohammed! Loop 4: Mohammed! Loop 5: Mohammed! </pre>

accessing the protected data. Critical sections are parts of the code where threads access one or several shared resources. If two or more threads try to access the same resource or set of resources, they must therefore be treated atomically. Any other threads attempting to execute a critical section will be blocked until the lock on that critical section is released. Mutexes are the simplest and most primitive way of delimiting critical sections so that threads behave nicely to one another and Pthreads supply a family of calls for using them. The two most important calls are `pthread_mutex_lock()`, that locks a mutex, and the cryptically titled `pthread_mutex_unlock()`.

2.4.5 Pthread conditions variables

While a mutex allows threads to synchronize by controlling their access to data, a condition variable lets threads synchronize on the value of a data. Cooperating threads wait until the data reaches a particular state or until a certain event occurs. Condition variables provide a kind of system notification for threads.

Table 2.5 – Pthread Mutex and conditional variable code example

```
1. #include <stdio.h>
2. #include <pthread.h>
3. #include <stdlib.h>
4. pthread_mutex_t fill_mutex;
5. int arr[10];
6. int flag=0;
7. pthread_cond_t cond_var=PTHREAD_COND_INITIALIZER;

8. void *fill() {
9. int i=0;
10. printf("\nEnter values\n");
11. for(i=0;i<4;i++) {
12. scanf("%d",&arr[i]);
13. }
14. pthread_mutex_lock(&fill_mutex);
15. pthread_cond_signal(&cond_var);
16. pthread_mutex_unlock(&fill_mutex);
17. pthread_exit(NULL);
18. }

19. void *read() {
20. int i=0;
21. pthread_mutex_lock(&fill_mutex);
22. pthread_cond_wait(&cond_var,&fill_mutex);
23. pthread_mutex_unlock(&fill_mutex);
24. printf("Values filled in array are");
25. for(i=0;i<4;i++) {
26. printf("\n %d \n",arr[i]);
27. }
28. pthread_exit(NULL);
29. }

30. main() {
31. pthread_t thread_id ,thread_id1;
32. pthread_attr_t attr;
33. int ret;
34. void *res;
35. ret=pthread_create(&thread_id ,NULL,&fill ,NULL);
36. ret=pthread_create(&thread_id1 ,NULL,&read ,NULL);
37. printf("\n Created threads");
38. pthread_join(thread_id ,&res);
39. pthread_join(thread_id1 ,&res);
40. }
```

Table 2.6 – Pthread barrier functions

Function	Description
<code>pthread_barrierattr_init()</code>	Initializes a barrier's attributes object
<code>pthread_barrier_init()</code>	Initializes a barrier
<code>pthread_barrier_destroy()</code>	Destroys a barrier
<code>pthread_barrier_wait()</code>	Synchronizes participating threads at the barrier

As mentioned earlier, if Pthreads did not offer condition variables, but only provided mutexes, the threads would need to poll the variable to determine when it reached a certain state. A pthread condition variable has a data type of `pthread_cond_t`. One can initialize it statically as it is done in the example provided in Table 2.5, or can initialize it dynamically by calling `pthread_cond_init()`, which is defined as follows:

```
int pthread_cond_init(pthread_cond_t *cv,
                     const pthread_condattr_t *cattr);
```

After the condition variable initialization, a thread can use it in one of the following two ways: the thread can either wait on the condition variable or can call `pthread_cond_wait()` or `pthread_cond_timedwait()` functions. Both of these functions suspend the caller until another thread signals on the condition variable. In addition, the `pthread_cond_timedwait()` call lets you specify a time-out argument. If the condition is not signaled in a specified time, the thread is released from its waiting state. The example provided in Table 2.5 illustrates the use of both the mutex and the condition variables. We create two threads:

- the first one named "fill" is in charge of filling values into the array "arr",
- the second one named "read" whose role is to read the values of the array.

Both threads lock the mutex using `pthread_mutex_lock()` function. The "fill" thread fills the array with 4 values and then locks the mutex using `pthread_mutex_lock()`. Once obtained, it "signals" the condition variable. The "read" thread on the other hand tries first to access the lock. The `pthread_cond_wait()` function then atomically releases the mutex and causes the calling thread to block on the condition variable `cond_var` and go into wait state, thus preventing any potential blocking. Interblocking can't occur because the lock is automatically unlocked by the system when `pthread_cond_wait()` is called and locked again when returning from the function.

2.4.6 Pthread barriers

A barrier is a mechanism of synchronization that forces several threads to wait at a specific point of the code until all of them have finished.

```
#include <pthread.h>
int pthread_barrier_init(pthread_barrier_t *barrier,
                        const pthread_barrierattr_t *attr,
                        unsigned int count);
```

where a `barrier` is a pointer to an object of type `pthread_barrier_t`, its attributes being determined by `attr`. The `count` parameter holds the number of threads that must be synchronized, each thread having to perform a call to the `pthread_barrier_wait()` function. The main thread creates the barrier object and initializes it with a count representing the total number of threads that must be synchronized to the barrier before the threads may carry on. The pthread barrier functions are summarized in Table 2.6.

In the example shown in Table 2.7, we used a count of 3: one for the `main()` thread, one for `thread1()`, and one for `thread2()`. To simplify this example, we have the threads sleep to cause

a delay, as if computations were occurring. To synchronize, the main thread simply blocks itself on the barrier, knowing that the barrier will unblock only after the two worker threads have joined it as well.

2.5 Generating random numbers

Generating random numbers is essential in many cryptographic applications like key generation, protocols, nonce as well as in the Internet; for example, for choosing TCP sequence numbers. For these issues, we need generators which are able to construct large amounts of secure random numbers [112], [113]. To this end, True Random Number Generators (TRNGs) which extract randomness from physical processes are usually used. The sequences generated by TRNGs cannot be reproduced. However, generating random numbers by this way is time-consuming and expensive. Another way to generate random numbers is to use deterministic random number generators in which the seed is reseeded many times during the generation of the sequence. In this section, we will describe the Linux PRNG that we used in our work. The Entropy source of the proposed RNG comes from Linux PRNG [114]. The process of entropy extraction includes three steps: 1) updating the pools contents, 2) extracting random bits to be output, and 3) decrementing the entropy counter of the pool. This process involves hashing the pool contents using Secure Hashing Algorithm (SHA-1), and adding the results to the pool [114]. Within the kernel, the interface for receiving random values from the PRNG is the function `get_random_bytes(*buf, nbytes)` which relies on two device drivers named `/dev/random` and `/dev/urandom`. `/dev/random` will block after the entropy pool is exhausted. It will remain blocked until additional data has been collected from the sources of entropy that are available. This can slow down random data generation. `/dev/urandom` will not block. Instead it will reuse the internal pool to produce more pseudo-random bits. This step will increase the uniformity and the randomness of the generated sequence. As an illustration, we give below a sample of code that allows to exploit the `/dev/urandom` entropy source. This device is a special file that can be read just like any file:

```
int byte_count = 64;
char data[64];
FILE *fp;
fp = fopen("/dev/urandom", "r");
fread(&data, 1, byte_count, fp);
fclose(fp);
```

The Linux RNG internal architecture is depicted in Figure 2.8. Random bits are extracted from one of the three pools: they are extracted from the `urandom` pool when the user uses `/dev/urandom` and when the kernel calls `getchar_random_byte()`. The secondary pool will be accessed when the user handles `/dev/random`. Finally the primary pool is accessed when one of the two other pools does not have enough entropy and needs re-filling.

2.6 Software security analysis

Software security analysis is another unavoidable factor to ensure the quality at the code source level and to eliminate every security gap [115]. Since it is still possible to read data out of memory even if the application no longer has pointers to it, it is necessary to incorporate data security within the source code. In cryptographic applications, secret information (e.g., encryption keys) must be kept in memory for the minimum amount of time possible and should be written over, not just released, when no longer needed. One first step consists in wiping such sensitive data from memory once it is no longer needed in order to prevent any malicious attacks. The idea is to zero-fill buffers which contained sensitive information. In practice, we used the following instructions to scrub (i.e., zero) a buffer and guarantee that the compiler will not optimize it away:

Table 2.7 – Pthread barriers code example

```
#include <stdio.h>
#include <time.h>
#include <pthread.h>
pthread_barrier_t barrier; // barrier synchronization object
void * thread1 (void *not_used)
{
    time_t now;
    time (&now);
    printf ("thread1 starting at %s", ctime (&now));
    // do the computation
    // let 's just do a sleep here...
    sleep (20);
    pthread_barrier_wait (&barrier);
    // after this point, all three threads have completed.
    time (&now);
    printf ("barrier in thread1() done at %s", ctime (&now));
}
void * thread2 (void *not_used)
{
    time_t now;
    time (&now);
    printf ("thread2 starting at %s", ctime (&now));
    // do the computation
    // let 's just do a sleep here...
    sleep (40);
    pthread_barrier_wait (&barrier);
    time (&now);
    printf ("barrier in thread2() done at %s", ctime (&now));
}
int main () // ignore arguments
{
    time_t now;
    pthread_t idthread1, idthread2;
    // create a barrier object with a count of 3
    pthread_barrier_init (&barrier, NULL, 3);
    pthread_create (&idthread1, NULL, thread1, NULL);
    pthread_create (&idthread2, NULL, thread2, NULL);
    time (&now);
    printf ("main() waiting for barrier at %s", ctime (&now));
    pthread_barrier_wait (&barrier);
    // after this point, all three threads have completed.
    time (&now);
    printf ("barrier in main() done at %s", ctime (&now));
    pthread_exit( NULL );
    return (EXIT_SUCCESS);
}
```

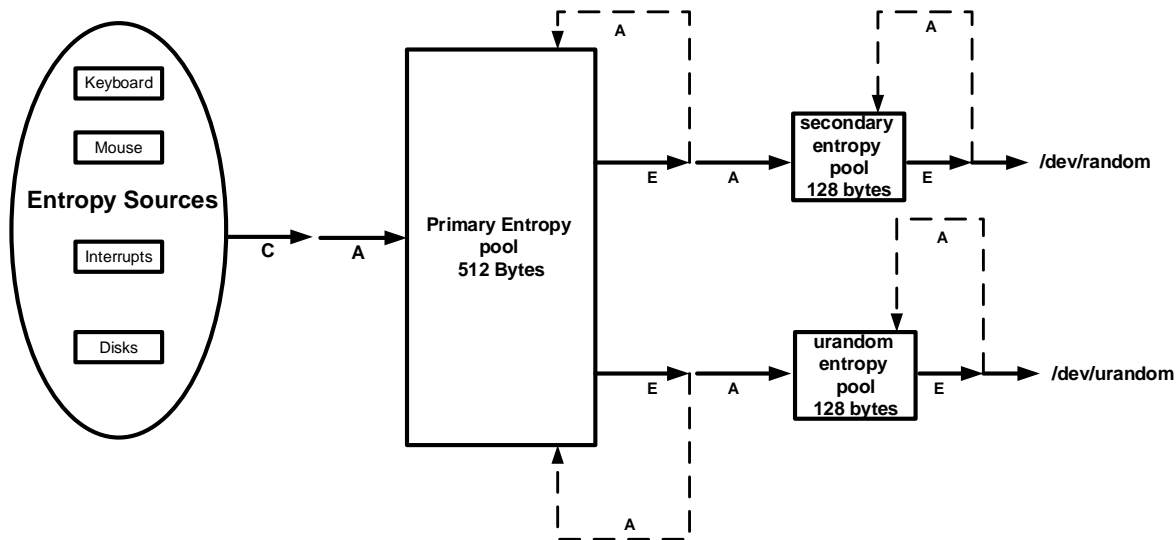


Figure 2.8 – Linux RNG

```
static void * (* const volatile memset_ptr)(void *, int, size_t) = memset;
static void secure_memzero(void * p, size_t len)
{
    (memset_ptr)(p, 0, len);
}

/* Zero sensitive information. */
secure_memzero(key, sizeof(key));
```

The `secure_memzero()` function relies on a function pointer `memset_ptr` that points itself to the `memset()` function. It exploits the key and the key size and will put zero value on the allocated memory related to the key by invoking `memset()`. The function `memset()` is used to write a specific value in a buffer that was allocated before. We used this function to write a zero value in the buffer. Some compilers optimize away the call to `memset()` function. To work around this, we declared `memset_ptr` as a *volatile* pointer. Since a *volatile* pointer can be manipulated outside the scope of the application, the code is not optimized by the compiler, thus keeping the program unchanged. Furthermore, the data in main memory may leak to the disk through virtual memory, thus representing another source of the most serious leaks (leaks to physical mediums). One solution, which is easy enough to apply, is to deactivate the swap space altogether, thus preventing data from being written to the page file by locking it in memory. In our implementation, we used the `mlock()` function that locks pages in the address range starting at address and continuing for length bytes. All pages that contain a part of the specified address range are guaranteed to be resident in main memory when the call returns successfully. Thereafter, the pages are guaranteed to stay in main memory until later unlocked.

In order to validate the correctness of our solution, we conducted a security code review using several static and dynamic techniques: Clang, Gdb, Valgrind, DRD, Callgrind and Leak-analysis tools.

2.6.1 Static software security tools

Static analysis is a software analysis performed without actually executing, or running, the software. Static analysis tools look at applications in a non-runtime environment. In our work we applied the following static software security tools to check vulnerability and threats.

Clang Static Analyzer:

The `Clang Static Analyzer` is a source code analysis tool that finds threats in C, C++, and Objective-C programs. Currently it can be run as a standalone tool invoked from the command line. The analyzer is open-source and is part of the Clang project. Like the rest of Clang, the analyzer is implemented as a C++ library that can be used by other tools and applications. The Clang Static Analyzer runs in a reasonable amount of time by both bounding the amount of checking work it will do as well as using "intelligent" algorithms to reduce the amount of work it must do in order to find bugs [116, 117].

Cppcheck:

`Cppcheck` is a C and C++ programming languages static code analysis tool [118]. It has been designed and developed by Daniel Marjamäki. `Cppcheck` is a free software available under the GNU General Public License. `Cppcheck` supports a wide variety of static checks that may not be covered by the compiler itself. These checks are static analysis checks that can be performed at a source code level. The program is directed towards static analysis checks that are strict, rather than heuristic in nature.

Coverity:

`Coverity` is a trademark of software development products from Synopsys [119], consisting primarily of static code analysis and dynamic code analysis tools. The tools enable the developer to find threats and security vulnerabilities in source code written in C, C++, Java, and JavaScript. `Coverity` was an organization founded in the Computer Systems Laboratory at Stanford University in Palo Alto, California and with headquarters in San Francisco it is now owned by Synopsys. In June 2008, `Coverity` acquired Solidware Technologies. In February 2014, `Coverity` announced a convention to be gained by Synopsys, an electronic design automation company. The tool was used to examine over 150 open source applications for bugs; 6000 bugs found by the scan were fixed across 53 projects. This was prior to the launch of the current "Coverity Scan" service. `Coverity` helps decrease risk and lower overall project cost by identifying critical quality threats and potential security vulnerabilities during development, with accurate and actionable treatment guidance, based on patented techniques and a decade of research and development and analysis of over 10 billion lines of proprietary and open source code. One disadvantage that it is not open source.

Frama-C:

`Frama-C` stands for `Framework for Modular Analysis of C programs`. `Frama-C` is a set of interoperable program analyzers for C programs. `Frama-C` has been developed by Commissariat à l'Énergie Atomique et aux Énergies Alternatives (CEA-List) [120], and Inria. `Frama-C`, as a static analyzer, inspects programs without executing them. `Frama-C` is Open Source software. It works on Windows and Unix (Linux, Mac OS X, ...), prove formal properties on the code. Using specifications written in ANSI/ISO C Specification Language enables it to ensure properties of the code for any possible behaviour. `Frama-C` is an extensible and collaborative platform dedicated to source-code analysis.

2.6.2 Dynamic software security tools

Dynamic analysis is a software analysis performed at run time. In the following sections we review some of the dynamic software security tools that we applied in all our source code.

Valgrind

Valgrind is a framework for building dynamic analysis tools [121]. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile the programs in detail. Valgrind can also be used to build new tools. The Valgrind distribution currently includes six production-quality tools: a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache and branch-prediction profiler, and a heap profiler. It also includes three experimental tools: a stack/global array overrun detector, a second heap profiler that examines how heap blocks are used, and a SimPoint basic block vector generator.

Callgrind

Callgrind is a profiling tool that records the call history among functions in a program's run as a call-graph [122]. By default, the collected data consists of the number of instructions executed, their relationship to source lines, the caller/callee relationship between functions, and the numbers of such calls. Optionally, cache simulation and/or branch prediction (similar to Cachegrind) can produce further information about the runtime behaviour of an application. The profile data is written out to a file at program termination. It collects flat profile data: event counts (data reads, cache misses, etc.) are attributed directly to the function they occurred in. This cost attribution mechanism is called self or exclusive attribution. Callgrind extends this functionality by propagating costs across function call boundaries. For presentation of the data, and interactive control of the profiling, two command line tools are provided:

1. **callgrind_annotate**: This command reads in the profile data, and prints assorted lists of functions, optionally with source annotation.
2. **callgrind_control**: This command enables you to interactively observe and control the status of a program currently running under Callgrind's control, without stopping the program. You can get statistical information as well as the current stack trace, and you can request zeroing of counters or dumping of profile data.

For graphical visualization of the data, `KCachegrind`, must be used. It is a KDE/Qt based GUI that makes it easy to navigate the large amount of data that `Callgrind` produces.

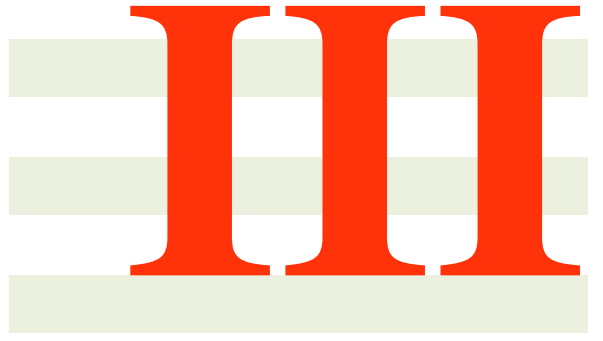
DRD: A thread error detector

DRD is a Valgrind tool for finding errors in multithreaded C and C++ programs [123]. It is the tool specified for any program that uses the POSIX threading primitives or that uses threading concepts built on top of the POSIX threading primitives. It can detect data races when one or more threads access the same memory location without sufficient locking. Most but not all data races are programming errors and are the cause of subtle and hard-to-find bugs. Also DRD can detect Deadlock that occurs when two or more threads wait for each other indefinitely. DRD supports any combination of multi-threaded programming paradigms as long as the implementation of these paradigms is based on the POSIX threads primitives. However, DRD does not support programs that use e.g. Linux' `futexes` directly. Attempts to analyze such programs with DRD will cause DRD to report many false positives.

2.7 Conclusions

In this chapter, we made a review for some parallel programming methods that are used in our applications. Both the basic fundamentals of OpenMP and the Pthread libraries and their usage have been

presented. OpenMP directives and clauses have been summarized. In addition the pthread-based multi-threaded programming has been detailed. Furthermore, a Linux pseudo-random generator was also described in this chapter. This model can be used to generate a random numbers by integrating `/dev/urandom` in the system. This step will increase the uniformity and the randomness of the generated sequence. In this chapter we pointed out how to protect the sensitive information at the code source level and to eliminate every security threats, sensitive information must be kept in memory for the minimum amount of time possible and should be written over, not just released, when no longer needed. One solution relies on wiping such sensitive data from memory once it is no longer needed in order to prevent any malicious attacks. Finally, dynamic and static software security analysis tools were presented. Static analysis is a software analysis performed without actually executing, or running the system. In this chapter we demonstrate a set of static analysis tools like Clang Static Analyzer, Cppcheck, Coverity. In other hand we also described a set of dynamic analysis tools (performed with system running) used in our thesis to test our code (Valgrind, Callgrind and DRD). In the next chapter, we will present the first contribution of our thesis work.



CONTRIBUTIONS

First contribution: Design and Efficient Implementation of Chaos-based Generators and a Stream Cipher

3.1 Introduction

Cryptography was used in the past to keep military information and diplomatic correspondence secure and to protect national security. In recent times, the range of cryptography applications has been widely expanded, following the development of new communication means. Cryptography is used to ensure that the contents of a message are confidentially transmitted and will not be altered. Chaos is one interesting field of research dealing with nonlinear, deterministic, and dynamic systems. It is applied to many different domains such as physics, robotics, biology, finance and encryption. The most important chaos properties are the high dependency on initial conditions and parameter variation, ergodicity and the random-like behavior. These properties enticed researchers into developing chaotic secure communication systems [13, 39, 124, 125, 126, 127, 128, 129, 130]. Under certain conditions, chaos can be generated by any non-linear dynamic system [131]. For public channels including network communication and for computer communication, most data transactions (valuable information) need to be protected from malicious attacks and threats [132, 133, 134]. A block symmetric cipher is one of the classical encryption techniques which is widely used in the literature. The Advanced Encryption Standard (AES) is one of the most famous symmetric encryption methods for block ciphers. The stream cipher is used to secure useful information that must be transmitted continuously over the network communication, for example. Generally, stream ciphers are more efficient than block ciphers in two situations: 1) in software applications requiring a very high encryption or decryption rate, and 2) in hardware applications where physical resources (e.g., chip area, power, etc) are restricted. Handling a stream cipher encryption with block ciphers is possible by using counter and output feedback modes (CTR, OFB). Because the AES is very secure and widely adopted, its two modes, namely CTR and OFB are used as stream ciphers. However, to benefit from both advantages of stream ciphers compared to block ciphers, several stream cipher designs such as RC4 and eSTREAM algorithms have been produced. RC4 is one of the widely known stream ciphers, and its efficient hardware implementation was performed by Gupta et al. [135]. However, RC4 has now been broken. The eSTREAM project was a multi-year effort, running from 2004 to 2008, to promote the design of efficient and compact stream ciphers suitable for the widespread adoption of eSTREAM [46]. Nevertheless, until now most of the eSTREAM ciphers are still not definitely secure [136]. Chaos-based stream ciphers are used to enhance

the security issue [137].

In this chapter we design and implement in an efficient and secure manner a chaos-based generator. The proposed chaotic system uses two non-linear recursive filters, a technique of disturbance and a chaotic multiplexing. The non-linearity is achieved by using chaotic maps. Then, based on the previous chaotic generator, we implement and test two applications. The first application concerns the generation of a Random Number Generator (RNG) using a Pseudo-Chaotic Number Generator (PCNG). The second application is the realization of a chaos-based stream cipher. The remainder of the chapter is structured as follows. In the next section, the description of the proposed chaotic generator is detailed. In section 3.3 the parallel implementation technique is described. In section 3.4, the first application is presented and analysed. In section 3.5 we present the chaos based stream cipher and we set out its performance in terms of computation performance and security using known cryptographic and statistical attacks. Software security tools are handled in section 3.6. Finally, in section 3.7 the conclusions are presented.

3.2 Proposed chaotic generator

The architecture of the proposed chaotic generator is composed of several black-boxes, as presented in Figure 3.1 [138]. The detailed description of the internal state and the output functions are given in Figure 3.2 and 3.3. The secret key K , the initial vector `Nonce IVg` and `Parameters` are the inputs of the chaotic generator. In case of parallel implementation, from these inputs, the `IV-setup` computes another three IVs values and the `Key-setup` creates another three keys. Then, four IVs and four keys will be used by four threads in the system. Because chaos is sensitive to any small changes in the secret key, the creation of each new key in the `Key-setup` entity is achieved by the circular shift rotation of the three-bit value of $K1_s, K1_p$ parameters (see Equation 3.5). Also, the creation of each new IV in the `IV-setup` entity is achieved by the circular shift rotation of the three bit value of U_s, U_p (see Equation 3.6). Before the execution of the program is completed, a new IV value is generated and stored in the `Non-Volatile Memory` box. The generation of this new value comes from `/dev/urandom` Linux PRNG [114]. The internal state, which contains the main cryptographic complexity of the system, is formed by two recursive filters of order three (delay 1, 2 and 3) [26]. The first recursive cell contains a discrete Skew tent map (STmap) and the second one contains a discrete piecewise linear chaotic map (PWLC). These maps are used as non-linear functions. We give below the outputs of the recursive cell containing the STmap and of

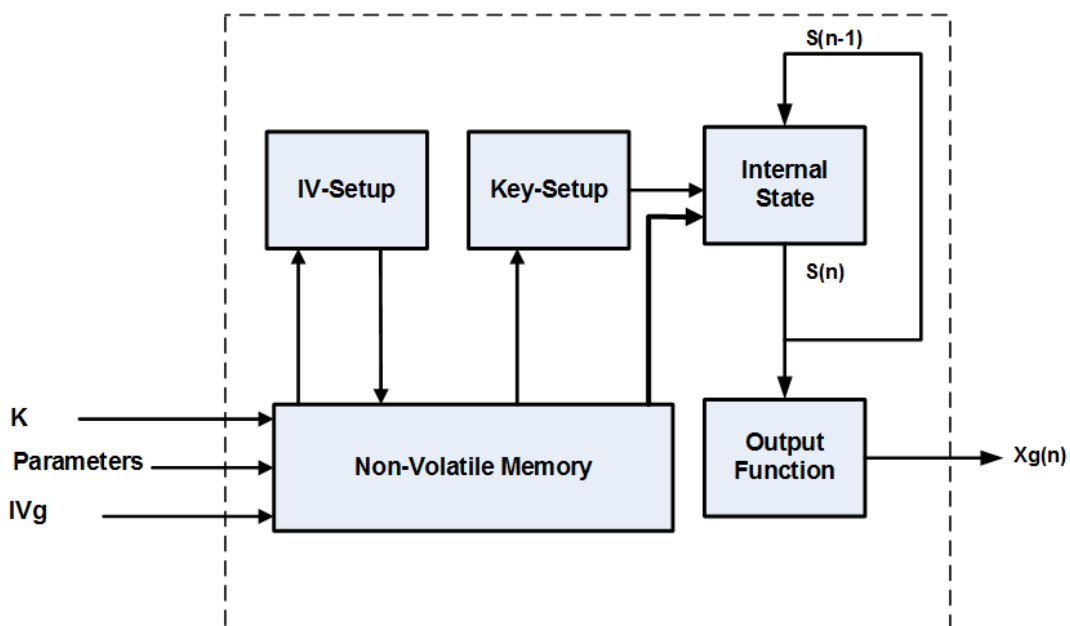


Figure 3.1 – Architecture of the proposed generator with internal feedback mode

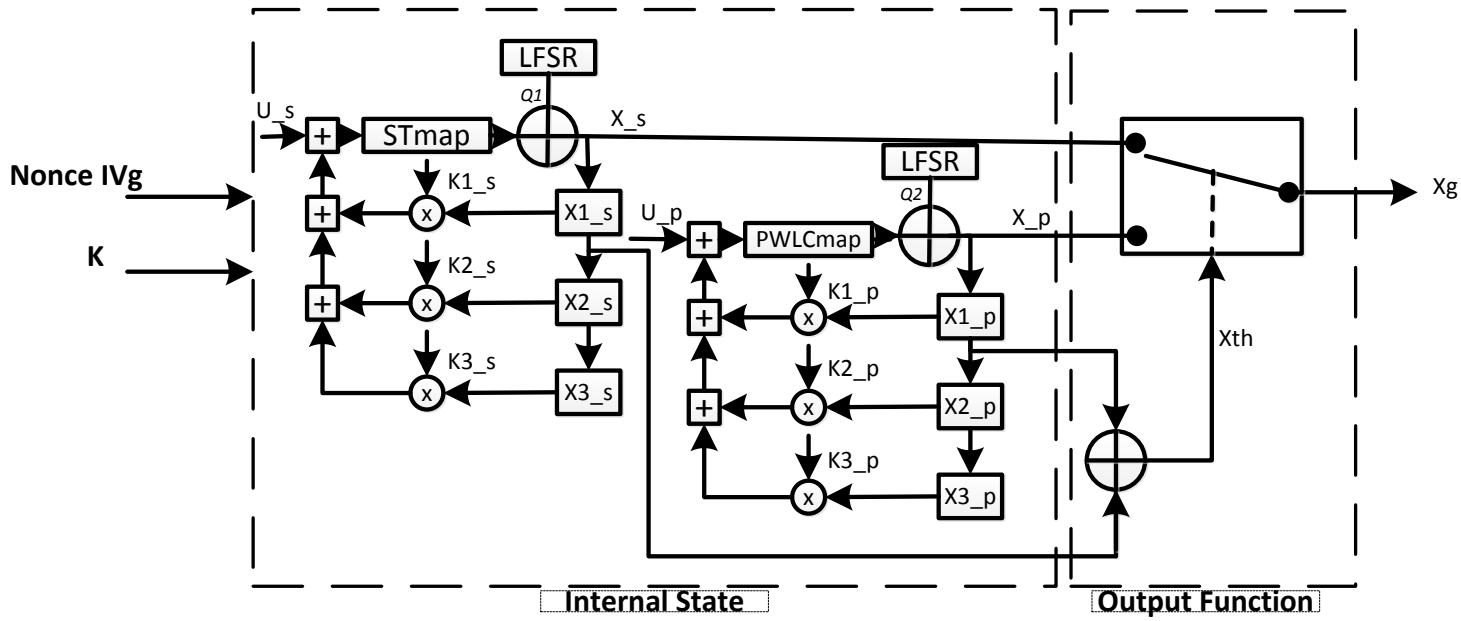


Figure 3.2 – Detailed description of the internal state and the output function

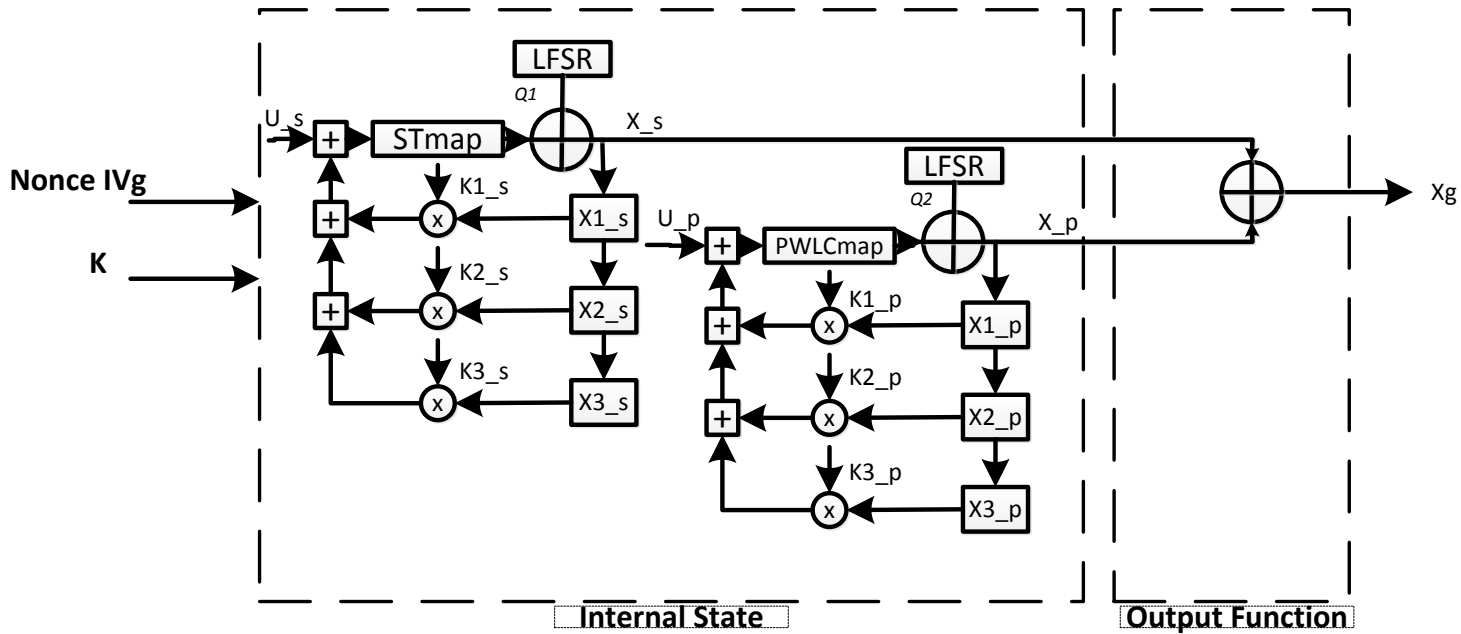


Figure 3.3 – Detailed description of the internal state and the output function using Xor.

the recursive cell containing the PWLC map respectively. Hence the output equation of the recursive cell STmap is:

$$X_{_s} = STmap\{F1[n - 1], P1\} \oplus Q1 \tag{3.1}$$

with

$$F1[n - 1] = mod[U_{_s} + X_{_s}(0) \sum_{i=1}^3 [K(i)_{_s} \times X(n - i)_{_s}], 2^N] \tag{3.2}$$

And the output equation of the recursive cell PWLC is:

$$X_{_p} = PWLCmap\{F2[n - 1], P2\} \oplus Q2 \tag{3.3}$$

with

$$F2[n - 1] = mod[U_{_p} + X_{_p}(0) \sum_{i=1}^3 [K(i)_{_p} \times X(n - i)_{_p}], 2^N] \tag{3.4}$$

In the equations above, $P1$ and $P2$ are control parameters in the range $[1, 2^N - 1]$ and $[1, 2^{N-1} - 1]$ respectively. $Q1$ and $Q2$ are perturbing signals produced by the linear feedback shift registers (LFSRs). $K1_s, K2_s, K3_s, K1_p, K2_p, K3_p$ are the coefficients of the recursive cells in the interval $[1; 2^N - 1]$. U_s and U_p , each of 32 bits are deduced from IVg of 64 bits as shown in Equation 3.6.

In parallel implementation we fix the number of threads to four, and each created thread needs to have its own secret key to generate samples. Thus, we create three others secret keys from the initial one, and then others initial values from original IV as follows:

$$\left\{ \begin{array}{l} K11_s = K1_s \\ K12_s = L_{cir}[K11_s, 3] \\ K13_s = L_{cir}[K12_s, 3] \\ K14_s = L_{cir}[K13_s, 3] \\ K11_p = K1_p \\ K12_p = L_{cir}[K11_p, 3] \\ K13_p = L_{cir}[K12_p, 3] \\ K14_p = L_{cir}[K13_p, 3] \end{array} \right. \quad (3.5)$$

$$\left\{ \begin{array}{l} U_s = lsb(IV) \\ U1_s = U_s \\ U2_s = L_{cir}[U1_s, 3] \\ U3_s = L_{cir}[U2_s, 3] \\ U4_s = L_{cir}[U3_s, 3] \\ U_p = msb(IV) \\ U1_p = U_p \\ U2_p = L_{cir}[U1_p, 3] \\ U3_p = L_{cir}[U2_p, 3] \\ U4_p = L_{cir}[U3_p, 3] \end{array} \right. \quad (3.6)$$

$lsb(IV)$ is the 32 least significant bits of IV , $msb(IV)$ is the 32 most significant bits of IV and $L_{cir}[S, r]$ performs the r -bits left circular shift on the binary sequence S , as follows in C code:

```

for(i=1 ; i < nb_cores ; i++){
// 3 bit circular shift for K_s,K_p,U_s and U_p
  shift= K[i-1][0].k_s[1] >> (32-3);
  K[i][0].k_s[1] = K[i-1][0].k_s[1] <<3;
  K[i][0].k_s[1] = K[i][0].k_s[1] | shift;

  shift= K[i-1][0]._k_p[1] >> (32-3);
  K[i][0].k_p[1] = K[i-1][0].k_p[1] <<3;
  K[i][0].k_p[1] = K[i][0].k_p[1] | shift;

  shift= K[i-1][0].U_s >> (32-3);
  K[i][0].U_s = K[i-1][0].U_s <<3;
  K[i][0].U_s = K[i][0].U_s | shift;

  shift= K[i-1][0].U_p >> (32-3);

```

```

K[i][0].U_p = K[i-1][0].U_p <<3;
K[i][0].U_p = K[i][0].U_p | shift;
}

```

The equations of the *Discrete* Skew Tent and *Discrete* PWLCM maps are respectively given by [23, 90, 139, 140]:

Discrete Skew Tent Map:

$$X_s[n] = \begin{cases} \left\lceil 2^N \times \frac{X_s[n-1]}{P1} \right\rceil & \text{if } 0 < X_s[n-1] < P1 \\ 2^N - 1 & \text{if } X_s[n-1] = P1 \\ \left\lceil 2^N \times \frac{2^N - X_s[n-1]}{2^N - P1} \right\rceil & \text{if } P1 < X_s[n-1] < 2^N \end{cases} \quad (3.7)$$

Discrete PWLCM map:

$$X_p[n] = \begin{cases} \left\lceil 2^N \times \frac{X_p[n-1]}{P2} \right\rceil & \text{if } 0 < X_p[n-1] \leq P2 \\ \left\lceil 2^N \times \frac{X_p[n-1] - P2}{2^{N-1} - P2} \right\rceil & \text{if } P2 < X_p[n-1] \leq 2^{N-1} \\ \left\lceil 2^N \times \frac{2^N - P2 - X_p[n-1]}{2^{N-1} - P2} \right\rceil & \text{if } 2^{N-1} < X_p[n-1] \leq 2^N - P2 \\ \left\lceil 2^N \times \frac{2^N - X_p[n-1]}{P2} \right\rceil & \text{if } 2^N - P2 < X_p[n-1] \leq 2^N - 1 \\ 2^N - 1 - P2 & \text{otherwise} \end{cases} \quad (3.8)$$

The values produced $X_s[n], X_p[n]$ by the recursive cells in the internal state are entered to the output function. Then, the output sequence $Xg(n)$ is obtained using a chaotic multiplexing controlled by the chaotic sequence $Xth = X1_s(n-1) \oplus X1_p(n-1)$ and by a threshold $Th = 2^{N-1}$, as shown in Figure 3.2, and Equation 3.9, or by xoring $X1_s$ and $X1_p$ as clarified in Figure 3.3 and Equation 3.10.

$$Xg(n) = \begin{cases} X_s(n), & \text{if } 0 < Xth \leq Th \\ X_p(n), & \text{otherwise} \end{cases} \quad (3.9)$$

$$Xg(n) = X_s(n) \oplus X_p(n) \quad (3.10)$$

The primitive polynomials used to generate the perturbing signals Q1 and Q2 are:

$$G12(x) = x^{21} + x^{13} + x^5 + x^2 + 1 \quad (3.11)$$

$$G18(x) = x^{23} + x^{12} + x^5 + x^4 + 1 \quad (3.12)$$

Notice that any other primitive polynomials (of the same degree) can be used.

Secret key and parameters components

In Tables 3.1 and 3.2 we give the structure of the secret key and the parameters used in our thesis applications.

Table 3.1 – Structure of the secret key.

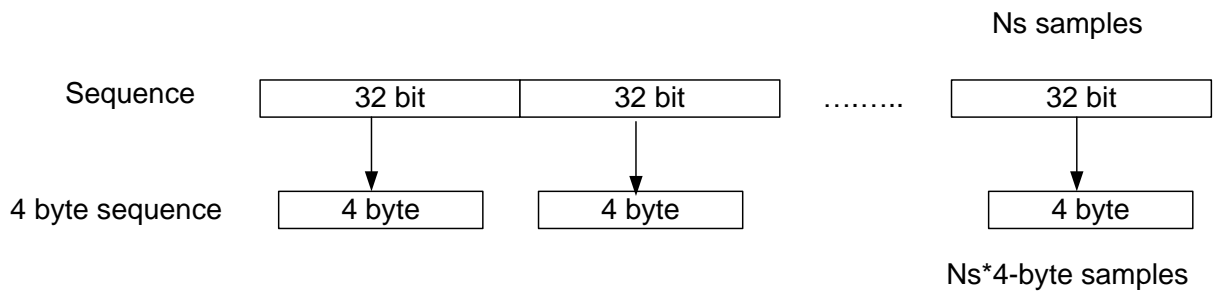
Variable	Secret key Description
reg_s	Initial value of LFSR used to perturb the skew tent map
reg_p	Initial value of LFSR used to perturb the PWLC map
P1	Parameter of the skewtent map
P2	Parameter of the PWLC map
X_s(0)	Initial value of the skewtent map
X_p(0)	Initial value of the PWLC map
X1_s	Initial value for the recursive cell with delay=1 using skew tent map
X1_p	Initial value for the recursive cell with delay=1 using PWLC map
X2_s	Initial value for the recursive cell with delay=2 using skew tent map
X2_p	Initial value for the recursive cell with delay=2 using PWLC map
X3_s	Initial value for the recursive cell with delay=3 using skew tent map
X3_p	Initial value for the recursive cell with delay=3 using PWLC map
K1_s	Parameter for the recursive cell with delay=1 using skew tent map
K1_p	Parameter for the recursive cell with delay=1 using PWLC map
K2_s	Parameter for the recursive cell with delay=2 using skew tent map
K2_p	Parameter for the recursive cell with delay=2 using PWLC map
K3_s	Parameter for the recursive cell with delay=3 using skew tent map
K3_p	Parameter for the recursive cell with delay=3 using PWLC map
tr	Transition value to reach the chaotic region; then the first useful generated sample is from the tr+1

Table 3.2 – Structure of parameters.

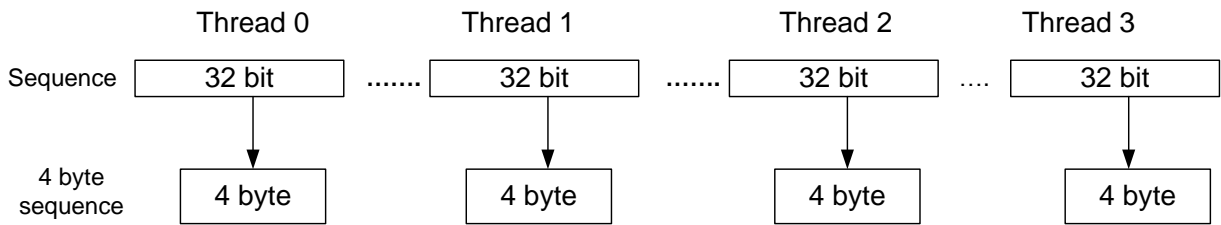
Variable	Parameters Description
N	Is the number of bits used as quantization
Delay(1,2,3)	Is the delay used in each recursive cell
Delta_s	Is the average orbit of the skewtent map without perturbation
Delta_p	Is the average orbit of the PWLC map without perturbation
g_s	Is the primitive polynomial used as perturbation for skewtent map
g_p	Is the primitive polynomial used as perturbation for skewtent map
IV	Is the initial vector

Byte conversion

Each sequence produced by the sequential generator is in 32-bit form. A conversion function is that provided in Algorithm 1 and Figure 3.4a is used to convert this sample to 4-byte. As the parallel chaotic generator produces 4 sequences 32-bit long in each call, the same conversion function is used to reform each produced sample to a 4-byte sequence. Figure 3.4b and 3.5 depict the byte conversion scheme and the storing of samples of different threads in the parallel implemented version.



(a) Sequential conversion to byte.



(b) Parallel conversion to byte.

Figure 3.4 – Byte conversion.

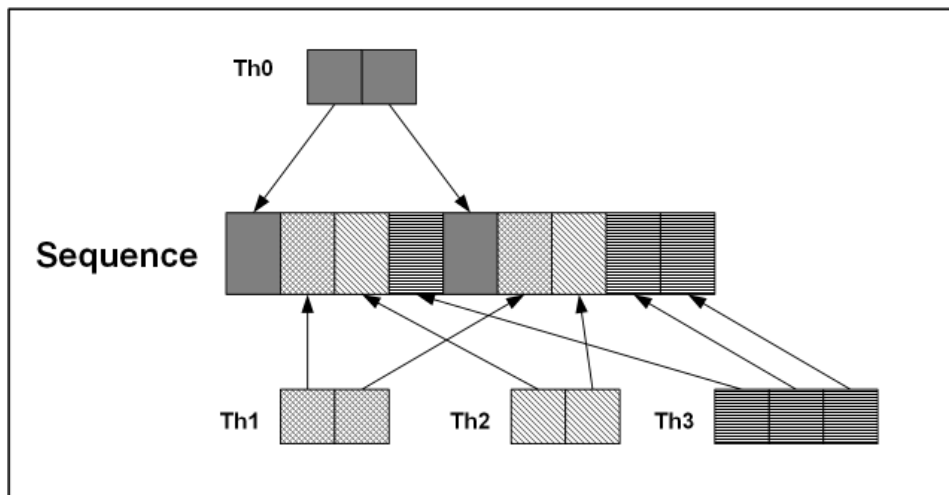


Figure 3.5 – Storing of samples generated by different threads.

Algorithm 1 Conversion to byte.

```

for (  $s = 1; s < NS; s++$  ) do
    sequence[s]= generator(s)
    byte_sequence[0+s×4] = (sequence[s]>>24) &0xFF.
    byte_sequence[1+s×4] = (sequence[s]>>16)&0xFF.
    byte_sequence[2+s×4] = (sequence[s]>>8) &0xFF.
    byte_sequence[3+s×4] = (sequence[s] &0xFF).
end for
    
```

Note: $Y = X \gg 24 \& 0xFF$ is a bitwise right shift by 24 of X and then takes its least significant byte. The generator() is a function that used to produce chaotic samples.

3.3 Parallel implementation

Parallel programming can be implemented using several different software interfaces, or parallel programming models. The programming model used in any application depends on the underlying hardware architecture of the system on which the application is expected to run: `shared memory` architecture or `distributed memory` environment. In Chapter 2 we detailed the parallel programming tools that were used in this thesis mainly, `OpenMP` and `Pthread`. The following subsections handle the usage of these techniques in implementation of the chaos-based generator.

3.3.1 Parallel implementation of the chaotic generator using OpenMP

As we mentioned in Chapter 2, `OpenMP` is an API for writing multithreaded applications. It is designed for multi-processor/core, shared memory machines. It is made of a set of compiler directives, library routines and environment variables for parallel application programmers. `OpenMP` provides capability to incrementally parallelize a serial program. Most major platforms have been implemented including Unix/Linux platforms and Windows. In order to convert the sequential version of the chaos based generator to a parallel one using the `OpenMP`, we accomplish parallelism exclusively through the use of the fork-join model of parallel execution. A master thread creates a team of parallel threads (fork) that simultaneously execute statements in the parallel region. After executing the statements in the parallel region, team threads synchronize and terminate (join) but master continues. We start by dividing the generation of key stream sequences among 4 threads by using the following `OpenMP` directive:

```
#pragma omp parallel num_threads(4)
{
    // Generation of Key_stream sequences
}
```

This implementation doesn't achieves any enhancements according to time performance due to a big overhead incurred by the threads' work. Thus, we decided to use the `pthread` programming approach. With `threads` we can achieve much better parallelism. This need a lot of code rewriting.

3.3.2 Parallel implementation of the chaotic generator using Pthread

Usually, a multi-thread process launches several threads that run concurrently. In our implementation, we parallelized the sequential version of our chaotic generator using the standard API used for implementing multithreaded application, namely `POSIX Threads` or `pthread` [98]. `pthread` is a library of functions that programmers can use to implement parallel programs. Unlike `MPI`, `pthread` is used to implement shared-memory parallelism. It is not a programming language (such as C or Java). It is a library that can be linked with C programs. The source code is compiled with `gcc` and using the `-lpthread` option. In our multithreaded approach, data sequences are partitioned among several threads. Threads execute the same instructions on different data sets. The number of samples to be processed and the starting point of the samples' subset data are different for each thread. The different threads are created and launched via a call to `pthread_create()`.

```
int pthread_create(pthread_t *restrict thread ,
    const pthread_attr_t *restrict attr ,
    void *(*computation)(void*), void *restrict arg);
```

In our case, we create a number of threads equal to the number of cores chosen by the user. The function `pthread_create()` takes the thread as a parameter. Each thread will call the `computation` function. This function ensures the generation of the samples and the conversion to bytes. Then the computed sequences from threads will be stored in a buffer in a systematic manner to gain a maximum performance. Each sequence from each thread is then stored consecutively as illustrated in Figure 3.5. In the

`main()` function, we wait for the termination of all threads by calling the `pthread_join()` function. The `main()` function of Pthread program is executed by a single, main thread. All other threads must be explicitly created by the main thread by calling the, `pthread_create()` function which creates a new thread. Algorithm 2 shows the decomposition of the samples among the threads. `Ns` is the number of samples to generate, `nb_cores` is the number of cores in the system, `nbTh` is the thread number (Th0, Th1, Th2 and Th4) and `max/min` are the sequence indexes.

Algorithm 2 Distribution of the samples among the threads.

```

input imin.
input imax.
input nb_cores.
input Ns.
remainder_seq = Ns mod nb_cores .
imin = nbTh × (Ns/nb_cores).
if nbTh = nb_cores -1 then
    imax = (nbTh + 1) × (Ns/nb_cores) + remainder_seq
else
    imax = (nbTh + 1) × (Ns/nb_cores)
end if

```

To describe the decomposition of the sequences among the threads, we give the following example: consider that 4 cores are available on the platform and that the sequence length is `seq_length=3125000` samples; 4 threads will then be created. The first thread computes samples from index `imin = 0*3125000/4 = 0` to index `imax=(0 + 1) * (3125000/4) - 1 = 781249`. The second thread computes samples from index `imin = 1 * 3125000/4 = 781250` to index `imax=(1 + 1) * (3125000/4) - 1 = 1562499` and so on until the last thread that will compute the rest of samples. The remainder of samples that resulted from the division of number of sequences by number of threads, if it exists, will also be computed by the last thread. Samples from each thread are stored in a shared result array, each thread filling specific index values.

Table 3.3 gives the number of computed samples by each thread in case of four threads and `Ns = 10` samples.

Table 3.3 – Number of samples computed by each thread (`Ns = 10`).

Thread	<code>imin</code>	<code>imax</code>	Actual number of computed samples
Th0	0	2	2
Th1	2	4	2
Th2	4	6	2
Th3	6	10	4

3.3.3 OpenMP vs Pthread

Pthreads and OpenMP represent two totally different multiprocessing paradigms. Pthreads is a very low-level API for working with threads. Thus, you have an extremely fine-grained control over thread management (`create/join/etc`), `mutexes`, and so on. It's fairly bare-bones. On the other hand, OpenMP is much higher level, is more portable and doesn't limit you to using C. It's also much more easily scaled than `pthreads`. One specific example of this is OpenMP's work-sharing constructs, which let you divide work across multiple threads with relative ease. But as usual, we get more flexibility and parallelism with `pthreads`. Finally, the nature of the programming problem will specify which API technique can be used in the application. Basically, the choice of an API technique depends on what the application is, the degree of parallelism required or how much the tasks need to react with each other, and how much

synchronization is required. In our applications we prefer to use `pthread` to gain maximum speed up and parallelism (see Table 3.4).

Table 3.4 – NCpB for Pthread and OpenMP implementation.

Data (Bytes)	NCpB-Pthread(Cycles/B)	NCpB-OpenMP(Cycles/B)
1024	1847	2118
16384	126	512
125000	26	211
196608	23	206

3.3.4 Computing performance of the chaotic generator

To evaluate the computing performance of the proposed chaotic generator using `pthread`, we performed some experiments using a two 32-bit multi-core Intel Core (TM) i5 processors running at 2.60 GHz with 16 G of memory. This hardware platform was used on top of an Ubuntu 14.04 Trusty Linux distribution. Here after, for different sizes of data bytes, we give the average generation time in micro second $GT(\mu s)$, the average bit rate in Mega bit per second $BR(Mbit/s)$, and the average of the required number of cycles to generate one byte, $NCpB(Cycles/B)$. The average is determined by using 100 different secret keys for each data size. For parallel implementation we use 4 threads in parallel running on a 4-core platform. The results obtained for $GT(\mu s)$, $BR(Mbit/s)$ and $NCpB(Cycles/B)$ are given in Tables 3.5, 3.6 and 3.7 with three delays and also are depicted in Figures 3.6, 3.7 and 3.8 for sequential and parallel implementation, only for delay 1. The average bit rate $BR(Mbit/s)$ and the number of cycles required to generate one byte $NCpB$ are given by:

$$BR = \frac{Data_Size_{(Mbit)}}{GT_{(\mu s)}} \quad (3.13)$$

$$NCpB = \frac{CPU\ Speed_{(Hertz)}}{BR_{(Mbit/s)}} \quad (3.14)$$

As we can see from these results, the parallel implementation is only better for data size equal to or bigger than 393216 bytes. This is due to the overhead time caused by the synchronization between threads and the overhead related to run the different threads. In Table 3.8 we compare our obtained results in terms of $NCpB$ with some known chaos-based generators, for data size equal to 786432 bytes that correspond to an image size of $512 * 512 * 3$. As we can see, the obtained performance by our generator is better than the others.

3.3.5 Statistical tests of proposed chaos based generator

In this section we report the results of several statistical tests that were carried out in order to quantify the statistical cryptographic properties of the proposed generator. They concern Mapping, Histogram, Chi-square, Approximated invariant measures, Auto and Cross Correlation, and NIST test.

3.3.5.1 Mapping, Histogram, Chi-square test and Approximated invariant measures.

Mapping

The phase space trajectory is one of the characteristics of the generated sequence that reflects the dynamic

Table 3.5 – Generation time for sequential and parallel implementation with three delays.

Data (Bytes)	GT-Seq/Parl d1 (μ s)	GT-Seq/Parl d2(μ s)	GT-Seq/Parl d3(μ s)
64	6 /705	8/709	11 /725
128	8 /726	9/731	13/733
256	11 / 743	13/755	16/761
512	19 /753	22/766	24/779
1024	32 /763	36/777	36/786
2048	57 /801	62/816	67/831
4096	109 /810	114/823	121/839
16384	332/835	341/849	351/855
32768	520 /847	534/861	551/873
65536	712 /764	728/781	734/792
125000	1282 /1325	1291/1334	1299/1348
196608	1830 /1869	1844/1881	1859/1893
393216	2902 /2436	2929/2451	2949/2467
786432	5502 /4835	5528/4877	5539/4889
3145728	21723 /19539	21739/19570	21761/19587
12582912	85009 /49154	85066/49169	85091/49181

Table 3.6 – Bit rate for sequential and parallel implementation with three delays.

Data (Bytes)	BR-Seq/parl d1 (Mbit/s)	BR-Seq/parl d2 (Mbit/s)	BR-Seq/parl d3 (Mbit/s)
64	85.33/0.73	83.61/0.68	81.31/0.61
128	128/1.41	126/1.21	125/1.01
256	186.18/ 2.76	185.02/ 2.23	184.78/2.04
512	215.58/ 5.44	214.17/4.87	213.14/3.96
1024	256/10.74	255/9.31	254/8.88
2048	287.44/ 20.45	286.14/19.58	285.22/18.91
4096	300.62/40.45	298.51/39.35	297.56/38.18
16384	394.8/156.97	392.12/155.16	391.01/154.38
32768	504.12/ 309.5	503.44/307.01	500.13/302.66
65536	736.36/ 686.24	733.78/684.98	731.12/682.75
125000	780.03/ 754.72	777.89/750.70	771.05/748.84
196608	859.49/ 841.55	857.48/839.68	853.80/837.18
393216	1083.99 /1291.35	1079.47/1288.25	1075.47/1286.74
786432	1143.49/1301.23	1140.03/1295.87	1137.60/1291.37
3145728	1158.49/1287.98	1154.12/1283.77	1151.12/1280.67
12582912	1184.15/2047.92	1178.38/2043.52	1173.77/2040.22

behaviour of the system. The resulting mapping in Figure 3.9 for delay 1,2 and 3 seems to be random compared to the known mapping (a signature) of a Skewtent and a PWLC map. This is due to the recursion of the structure, the perturbation and the chaotic multiplexing technique or xoring operation. In this case, it is impossible from the generated sequences to know which type of map is used. Therefore, the security of the system is improved.

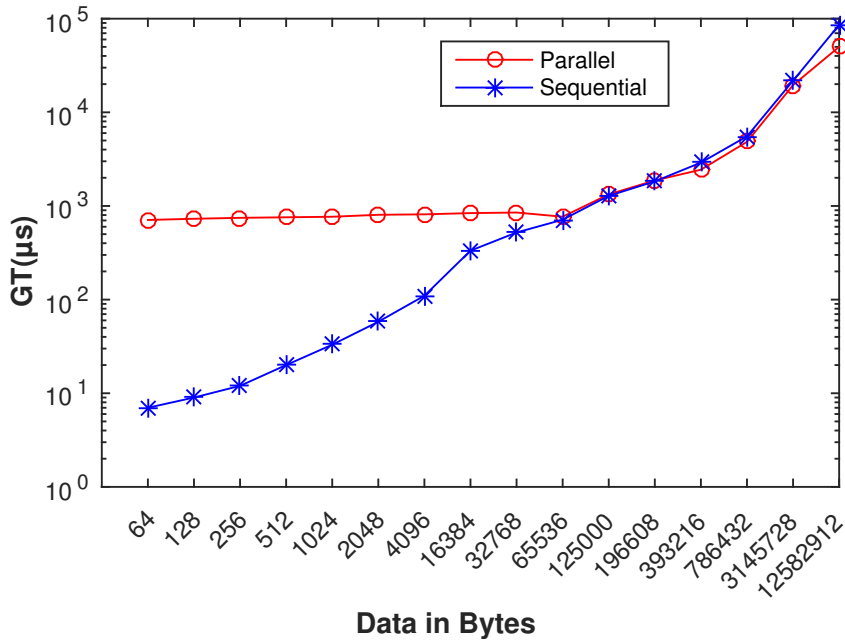


Figure 3.6 – Generation time for parallel and sequential implementation.

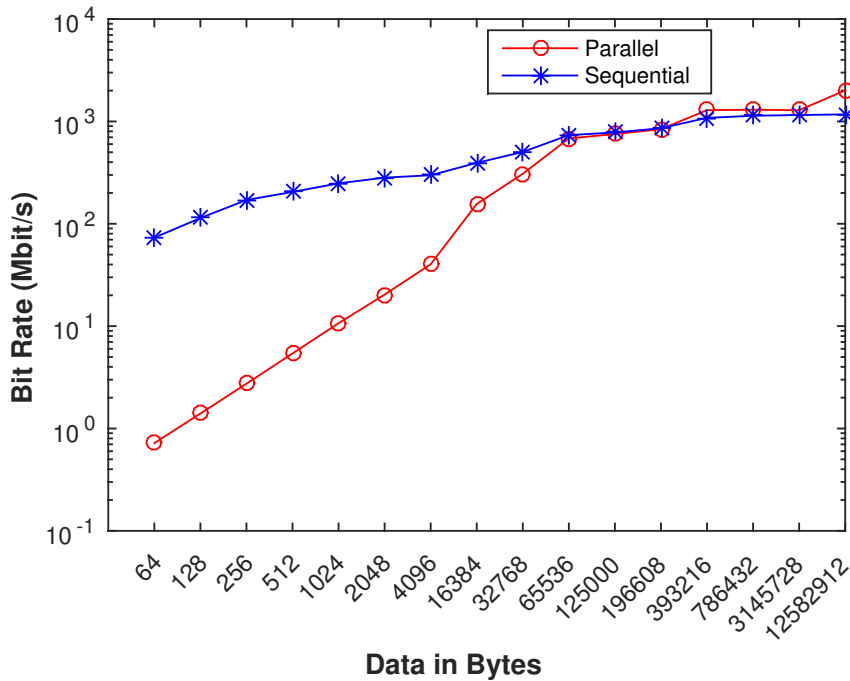


Figure 3.7 – Bit Rate for parallel and sequential implementation.

Histogram

Another key property of any robust pseudo chaotic number generator is to provide a uniform distribution in the whole phase space. We present in Figure 3.10 the histograms for three generated sequences corresponding to delay 1, 2 and 3 respectively. We can visually observed that the histograms have uniform distribution for the three delays.

Table 3.7 – NCpB for sequential and parallel implementation with three delays.

Data (Bytes)	NCpB-S/P(Cycles/B)-d1	NCpB-S/P(Cycles/B)-d2	NCpB-S/P(Cycles/B)-d3
64	232.5/27173.2	234.9/27178.1	236.5/27181.7
128	155/14068.4	159/14071.9	161/14073.4
256	106.5/7187.1	108.3/7189.4	111.7/7194.1
512	92/3646.4	96/3651.3	98/3655.7
1024	77.5/1847	81.3/1851.3	85.7/1856.1
2048	69/970	75/973	77/976
4096	66/490.4	69/494.1	71/498.3
16384	50.2/126.4	53.7/129.4	55.1/133.7
32768	39.3/64.1	34.7/67.6	39.2/69.3
65536	26.9/28.9	28.1/30.4	30.3/31.6
125000	25.4/26.3	27.1/28.1	29.2/29.5
196608	23.1/23.6	25.6/26.1	27.1/27.0
393216	18.3/15.4	20.7/17.7	22.1/19.6
786432	17.3/15.2	19.6/16.3	19.1/17.1
3145728	17.1/15.4	18.7/16.8	18.1/17.3
12582912	16.8/9.7	17.1/10.3	18.1/11.9

Table 3.8 – NCpB performance of some PRNG

PRNG	NCpB (Cycles/B)
Wang et al. [141]	160
Akhshani et al. [142]	45
Ons et al. [143]	24.68
Proposed algorithm	17.3

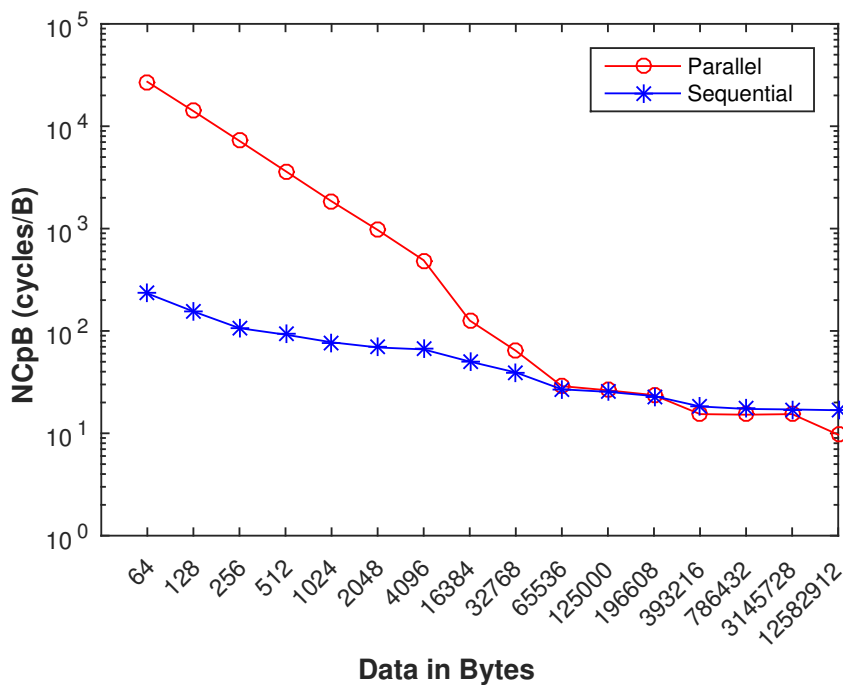


Figure 3.8 – NCpB for parallel and sequential implementation.

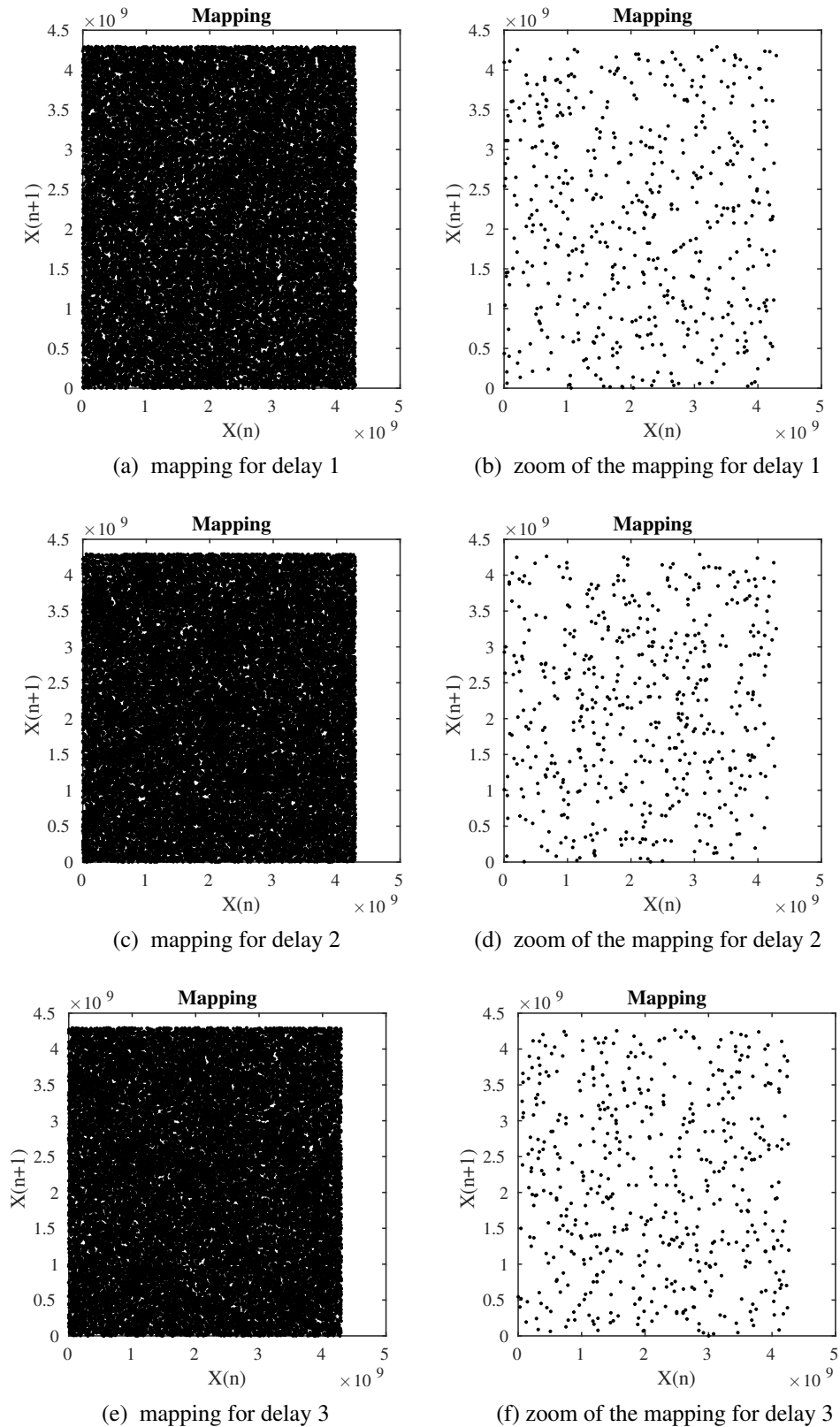


Figure 3.9 – Mapping of three generated sequences for delays 1, 2 and 3.

Chi-square test and Approximated invariant measures

We apply the Chi-Square test in order to assert the uniformity of generated sequences. The statisti-

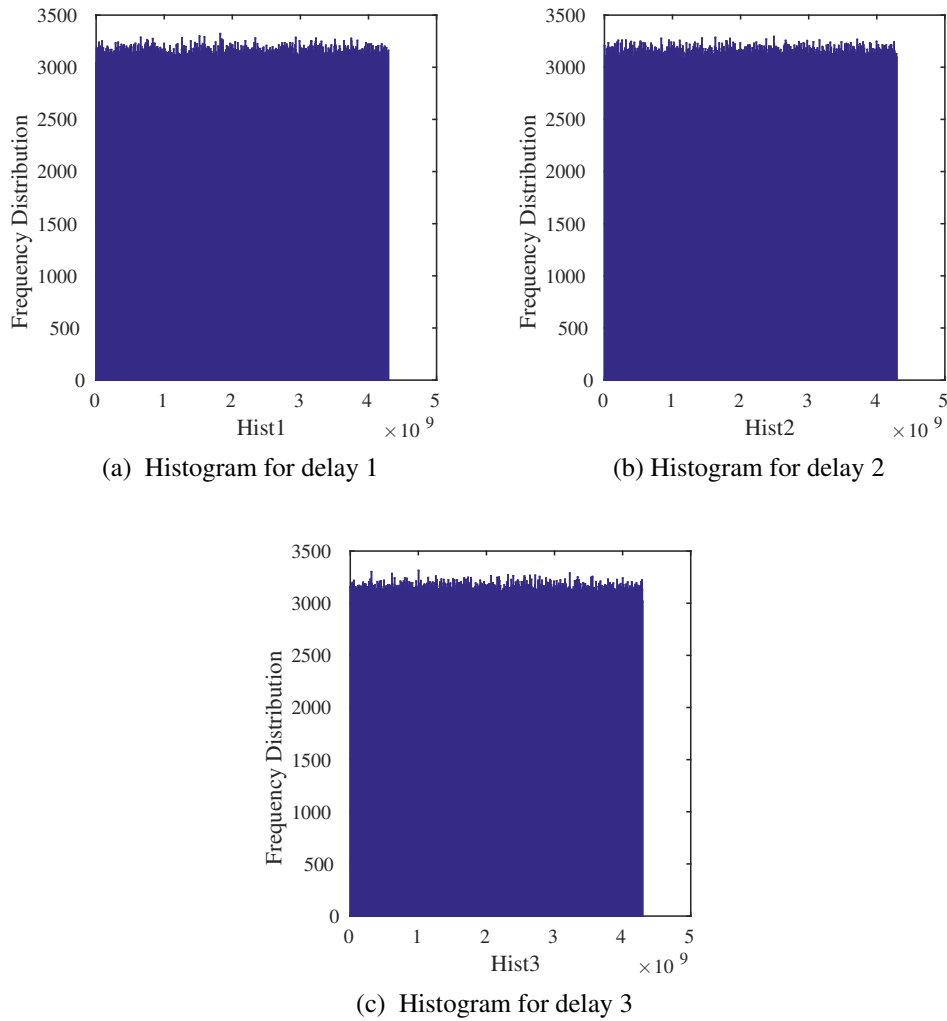


Figure 3.10 – Histograms of three generated sequences for delays 1, 2 and 3.

cal Chi-Square test χ^2 is calculated by the following formula:

$$\chi_{\text{exp}}^2 = \sum_{i=0}^{K-1} \frac{(O_i - E_i)^2}{E_i}. \tag{3.15}$$

with K being the number of classes (sub-intervals) equal to 1000, O_i being the number of observed (calculated) samples in the i -th class and E_i being the expected number of samples of a uniform distribution, $E_i = 10^7/K$. We compare the experimental value calculated above with a theoretical value obtained for a threshold $\alpha=0.05$ and a degree of freedom $K-1=999$. To prove the uniformity of a generated sequence, the experimental value of chi2 must be lower than the theoretical one $\chi_{\text{exp}}^2 < \chi_{\text{th}}^2$. More the experimental value of chi2 is smaller than the theoretical one, better is the uniformity of the generated sequence. Experimental and theoretical values of the Chi-Square test for sequences are presented in Table 3.9.

Table 3.9 – Experimental and theoretical values of the Chi-Square test for the proposed generator.

Test	delay=1	delay=2	delay=3
theoretical	1073.642651	1073.642651	1073.64265
experimental	1044.52000	1017.450800	980.710400

As we can see from Table 3.9, all histograms are uniform.

To prove the uniformity of the histogram, Lozi [28] uses the "approximated invariant measures". This function was computed with floating numbers and based on the partition of the mapping space to M^2 small squares (boxes). In finite precision N , the approximated invariant measures $Pd_N(s_i, t_j)$ are defined in the same manner, as follows [143]. First, the space mapping is divided into M^2 boxes r_{ij} as shown in Figure 3.11 with:

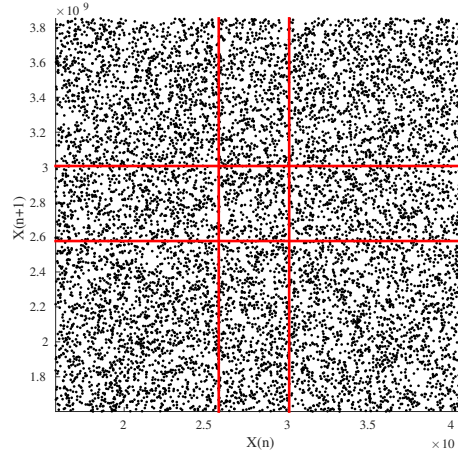


Figure 3.11 – Space mapping division.

$$s_i = X_{min} + i \times l, i = 0, \dots, M. \quad (3.16)$$

$$t_j = X_{min} + j \times l, j = 0, \dots, M. \quad (3.17)$$

Where

$$l = \frac{X_{max} - X_{min}}{M}. \quad (3.18)$$

with $X_{min} = \min(X_i(N_s))$, $X_{max} = \max(X_i(N_s))$ and N_s is the number of samples under test.

The box r_{ij} is given by :

$$r_{ij} = [s_i, s_{i+1}[\times [t_j, t_{j+1}[, i, j = 0, \dots, M - 1. \quad (3.19)$$

So, the approximated probability distribution function $Pd_N(s_i, t_j)$ is :

$$Pd_N(s_i, t_j) = \frac{\#r_{ij}}{N_s/M^2}. \quad (3.20)$$

with $\#r_{ij}$ being the number of samples inside the box r_{ij} .

The cumulative relative error (CRE) is calculated as follows:

$$CRE = \sum_{i,j=1}^M \left| \frac{N_s/M^2 - \#r_{ij}}{N_s/M^2} \right|. \quad (3.21)$$

Values of CRE for sequence X are given in Table 3.10. For this experiment, we take two different values for N_s : $N_s = 31250$, $N_s = 31250 \times 100$. And for each N_s , we consider four values of M : $M = 10, 32, 50$ and 100 .

Table 3.10 – CRE with delay=1.

Samples	M=10	M=32	M=50	M=100
31250	4.8265	146.1248	562.9231	4.4708e+03
31250 × 100	0.4485	14.8226	58.8994	453.9785

The *CRE* decreases when N_s increases. Also, the *CRE* increases when M increases for a given size of N_s . We notice that, for each M , the *CRE* of the proposed generator decreases by a factor approximately equal to $\sqrt{N_s}$.

Auto and cross correlation

The properties of a random sequence are that the values in the sequences are not repeated or correlated, and the cross-correlation of two sequences x and y (generated with slightly different keys) is close to zero. The correlation coefficient ρ_{xy} of the two sequences x and y is calculated by the following mathematical equations [144]:

$$\rho_{xy} = \frac{\text{cov}(x, y)}{\sqrt{D(x)}\sqrt{D(y)}} \quad (3.22)$$

where

$$\text{cov}(x, y) = \frac{1}{N} \sum_{i=1}^N ([x_i - E(x)][y_i - E(y)]) \quad (3.23)$$

$$D(x) = \frac{1}{N} \sum_{i=1}^N (x_i - E(x))^2 \quad (3.24)$$

$$E(x) = \frac{1}{N} \sum_{i=1}^N (x_i) \quad (3.25)$$

In the previous equations, x_i and y_i are the values of x and y respectively.

In Figure 3.12, we give the obtained results of the auto cross-correlation of 320 sequences; each contains 31250 samples; also, a zooming of these results is presented. As we can see, the correlation between the generated sequences is near zero, and the auto correlation is at its maximum.

3.3.5.2 NIST

To evaluate the statistical performances of the keystream produced, we also use one of the most popular test for investigating the randomness of binary data, namely the NIST statistical test [145]. This test is a statistical package that consists of 188 tests and sub-tests that were proposed to assess the randomness of arbitrarily long binary sequences. These tests focus on a variety of different types of non-randomness that could exist in a sequence. We generated 100 different binary sequences, each with a different secret key, and 31250 samples (corresponding to 1 million bits); we used the NIST test on all of these entities. For each test, a set of 100 *P_value* is produced and a sequence passes a test whenever the *P_value* $\geq \alpha = 0.01$, where α is the level of significance of the test. A value of $\alpha = 0.01$ means that 1% of the 100 sequences are expected to fail. The proportion of sequences passing a test is equal to the number of *P_value* $\geq \alpha$ divided by 100. In Figure 3.13 we present the obtained proportion versus test for delay 1, 2 and 3. As we can see, all the 188 tests and sub-tests pass the Nist, except one sub-test with delay 1. Notice that the minimum pass rate for each statistical test with the exception of the random excursion variant test is approximately= 0.960150 for 100 binary sequences. The minimum pass rate for the random excursion variant test is approximately 0.952091 for a sample size =62 binary sequences. In Tables 3.11, 3.12, 3.13, we give the *P_value* and the proportion with delays 1, 2 and 3 respectively for the 15 NIST tests.

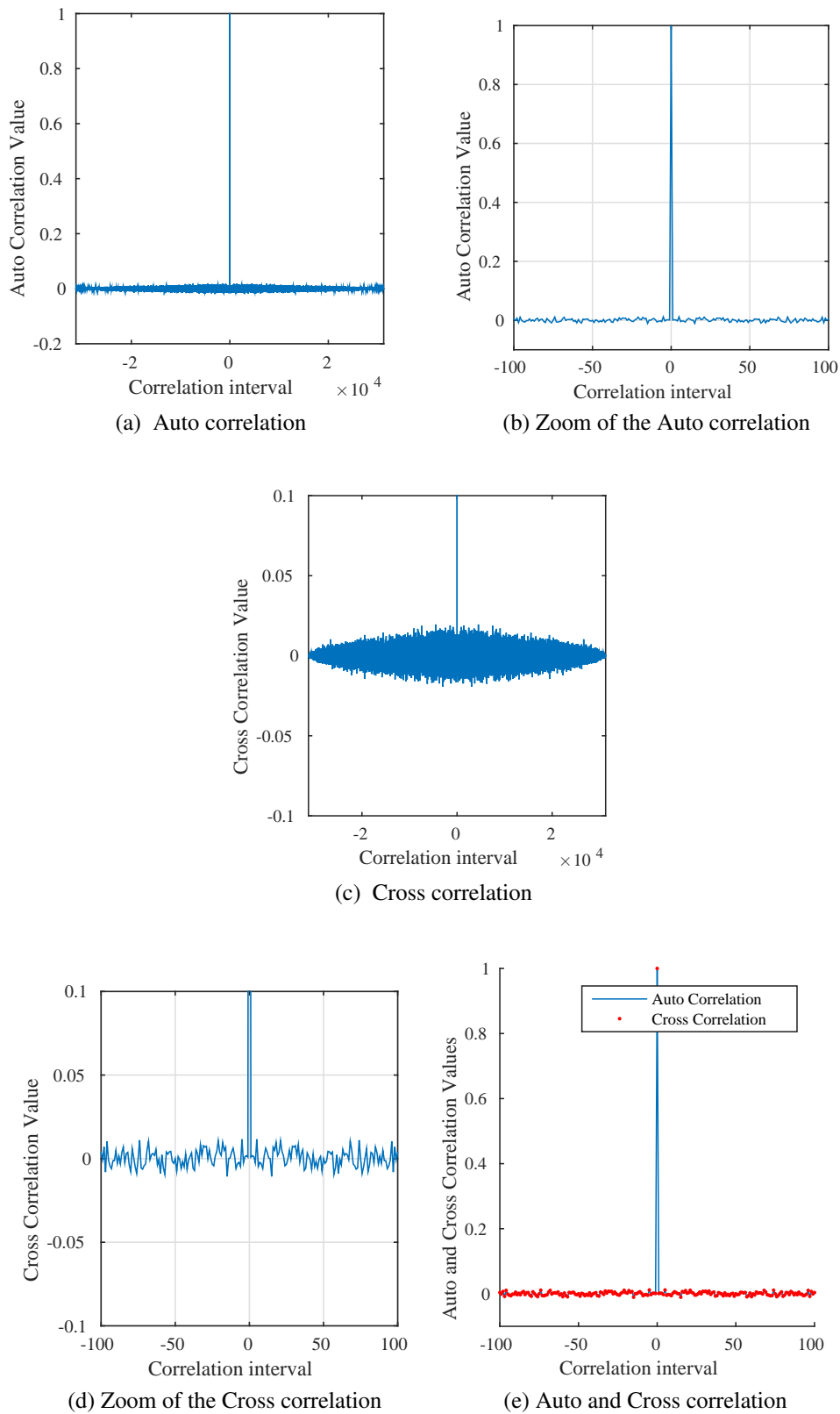


Figure 3.12 – Auto and cross correlation for generated sequences, with delay 1.

All statistical results indicate the strength of the generated keystream.

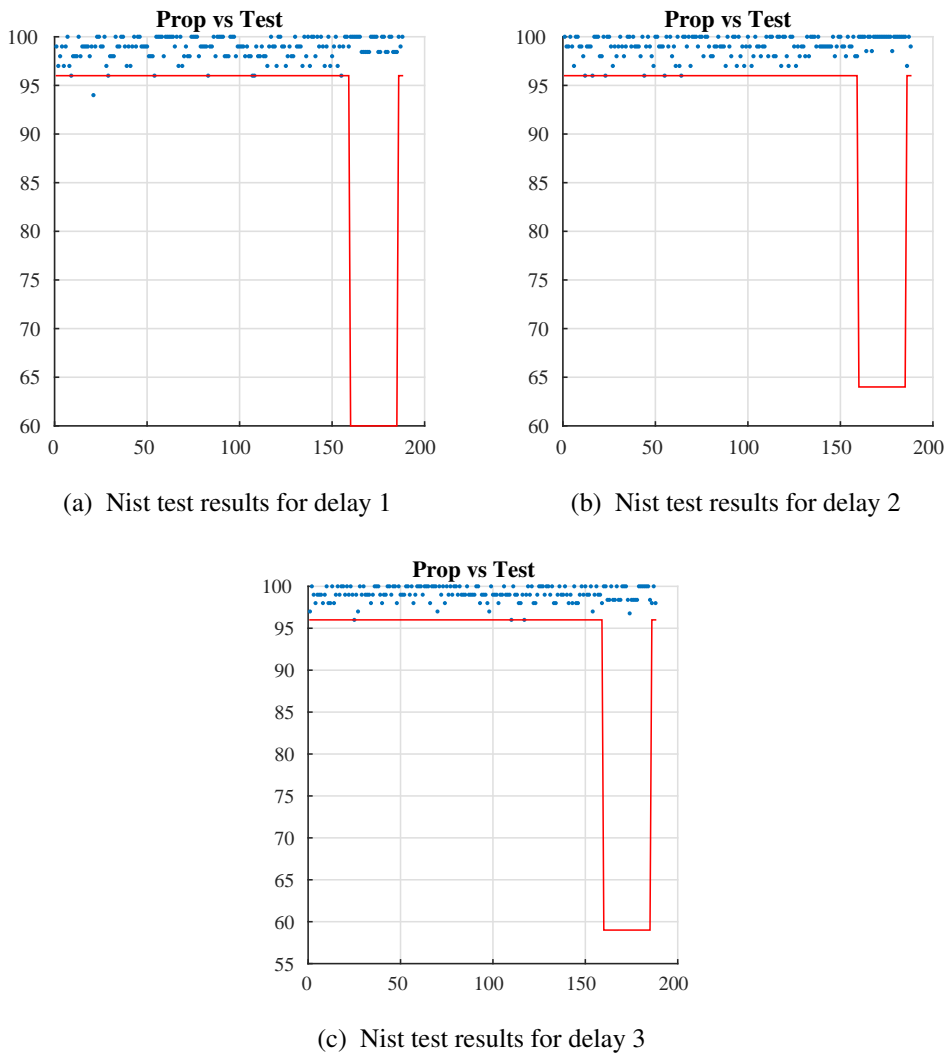


Figure 3.13 – Nist test results for generated sequences.

Table 3.11 – Nist Test values with delay 1

Test	P_value	proportion
frequency test	0.637	100.000
Block-frequency test	0.956	98.000
Cumulative-sums test	0.715	99.500
Runs test	0.720	98.000
Longest-run test	0.055	98.000
Rank test	0.554	99.000
FFT test	0.109	100.000
nonperiodic-templates	0.546	98.973
overlapping-templates	0.2256	99.000
universal	0.994	99.000
approximty entropie	0.575	99.000
random-excursions	0.428	97.581
random-excursions-variant	0.428	98.925
serial test	0.519	99.000
linear-complexity	0.740	98.000

Table 3.12 – Nist Test values with delay 2

Test	P_value	proportion
frequency test	0.494	97.000
Block-frequency test	0.760	100.000
Cumulative-sums test	0.797	97.000
Runs test	0.596	99.000
Longest-run test	0.699	98.000
Rank test	0.029	100.000
FFT test	0.834	98.0000
nonperiodic-templates	0.479	99.000
overlapping-templates	0.237	96.000
universal	0.494	98.000
approximty entropie	0.740	99.000
random-excursions	0.223	99.375
random-excursions-variant	0.428	98.925
serial test	0.828	99.500
linear-complexity	0.834	100.000

Table 3.13 – Nist Test values with delay 3

Test	P_value	proportion
frequency testS	0.851	100.000
Block-frequency test	0.172	99.000
Cumulative-sums test	0.382	99.500
Runs test	0.679	99.000
Longest-run test	0.0.883	97.000
Rank test	0.367	100.000
FFT test	0.367	100.000
nonperiodic-templates	0.482	98.905
overlapping-templates	0.964	100.000
universal	0.437	98.000
approximty entropie	0.679	98.000
random-excursions	0.336	99.632
random-excursions-variant	0.339	99.918
serial test	0.557	98.500
linear-complexity	0.475	99.000

3.4 First Application: Design of a pseudo-chaotic number generator as a random number generator

Random numbers are best obtained using physical True Random Number Generators (TRNG), which operate by measuring a well controlled and specially prepared physical process [146]. However, TRNGs based on physical processes are inefficient (slow and costly). Alternatively, random number generation can be realized by using a structure based on deterministic random number generators (DRNGs), usually called pseudo-random number generators (PRNGs). A random number generator (RNG) is a computer program intended to behave like a random variable. More specifically, PRNGs have

always attracted attention from both computer science and mathematics communities. A PRNG takes as input a fixed value, called the seed, and produces a sequence of output numbers or bits using a deterministic algorithm [43]. The seed value is often generated by a TRNG or an entropy source. Algorithms based on RNGs such as Blum Blum Shub or Blum-Micali [147] have a security proof, but they are very inefficient (slow) and therefore impractical unless extreme security is needed. There are a number of practical schemes for PRNGs, based on either number theoretical designs, hash function, or block cipher algorithms such as AES-CTR [145], that can be used in practical applications for both software and VLSI chips testing. The security provided by RNG that uses a Deterministic PRNG mechanism is a system implementation issue; both the Deterministic PRNG mechanism and its source of entropy input must be considered when determining whether the RNG is appropriate for use the consuming applications [145]. Generating random numbers is an important task in cryptography. RNGs are necessary not only for generating cryptographic keys, but are also needed in the steps of cryptographic algorithms or protocols, like password generation, nonce generation, the initialization vectors for symmetric encryption [148]. Also, random number generators have applications in statistical sampling, computer simulation, Monte Carlo-method simulations, cryptography, gambling, completely randomized design, and other areas where producing an unpredictable result is desirable [114].

In this contribution, we propose a new RNG based on a pseudo-chaotic number generator (PCNG) that uses the Linux random number generator `/dev/urandom`.

3.4.1 Scheme of the proposed Pseudo-chaotic Number Generator as a RNG

The architecture of the proposed Pseudo Chaotic Number Generator (PCNG) is presented in Figure 3.14. It is formed by two recursive filters of order one. Figure 3.14 describes the whole structure of the generator with xoring operation mode and a finite computing precision of $N = 32$. The first recursive cell contains a discrete Skew tent map and the second recursive cell contains a discrete piecewise linear chaotic (PWLC) map [149]. These maps are used as non-linear functions [23]. In order to produce the final Xg random sequence, this version of the generator implements an xoring operation on STmap and PWLC map outputs (see Equation 3.26). The Equations of the recursive cells are based on the previous Equations 3.1, 3.2, 3.3, 3.4, 3.7, 3.8.

$$F[n - 1] = F1[n - 1] \oplus F2[n - 1] \quad (3.26)$$

As detailed before in Chapter 2, the entropy source of the proposed RNG comes from Linux RNG [114]. The process of entropy extraction includes three steps: 1) updating the pool's contents, 2) extracting random bits as output, and 3) decrementing the entropy counter of the pool. This process involves hashing the pool contents using SHA-1, and adding the results to the pool [114]. Within the kernel, the interface for receiving random values from the RNG is the function `get_random_bytes(*buf, nbytes)` which relies on two device drivers named `/dev/random` and `/dev/urandom`. `/dev/random` will block after the entropy pool is exhausted. It will remain blocked until additional data has been collected from the sources of entropy that are available. This can slow down random data generation. `/dev/urandom` will not block. Instead it will reuse the internal pool to produce more pseudo-random bits. We fed all the necessary initial conditions and needed parameters for the generator shown in Figure 3.14. This step will increase the uniformity and the randomness of the generated sequence. As an illustration, we gave in Chapter 2 the structure of Linux RNG illustrated in Figure 2.8 and a sample of code that allows to exploit the `/dev/urandom` entropy source. We seeded our generator several times to produce a random number from the proposed deterministic PCNG. The obtained results in terms of statistical properties in Figure 3.15 and Table 3.14 indicate that the proposed PCNG can be used reliably for applications that need random numbers.

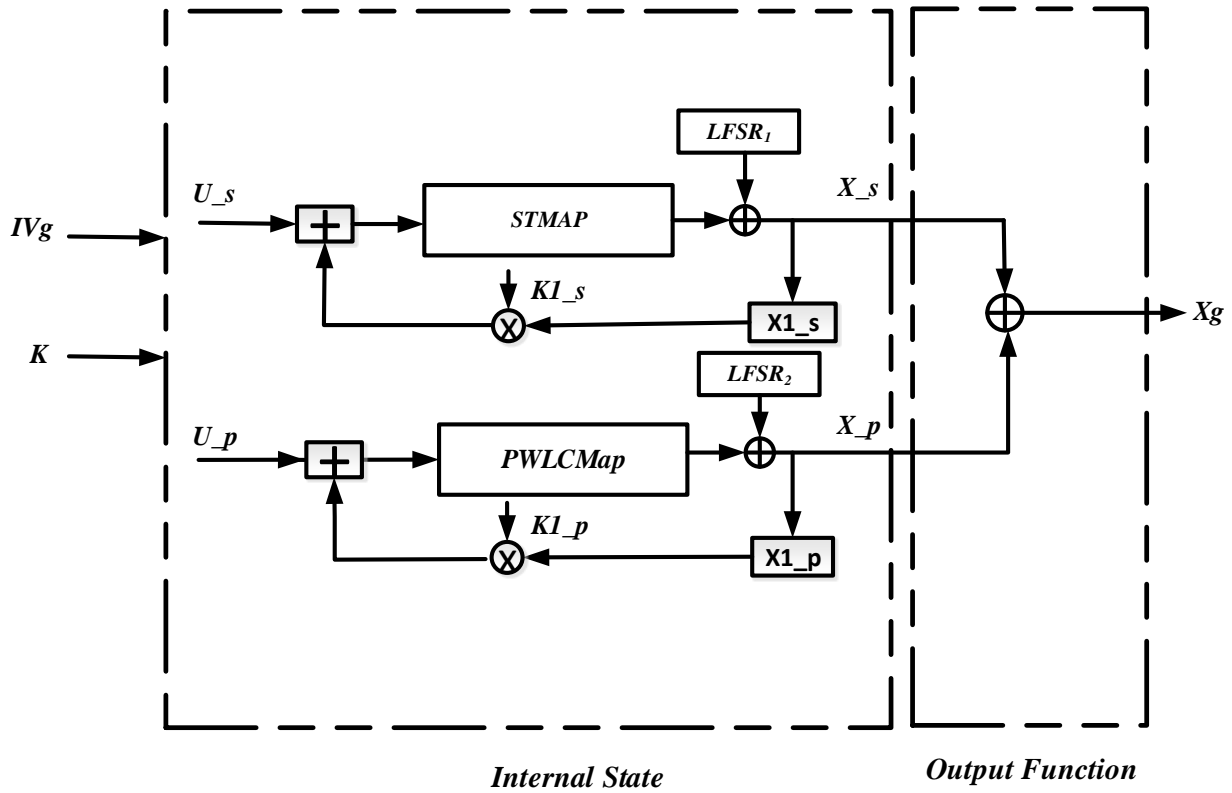


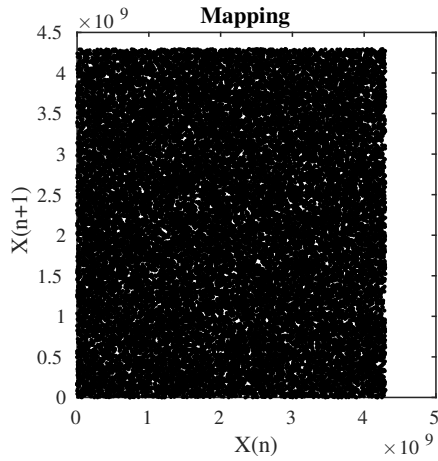
Figure 3.14 – The architecture of the proposed PCNG.

Table 3.14 – Nist Test values for proposed PCNG.

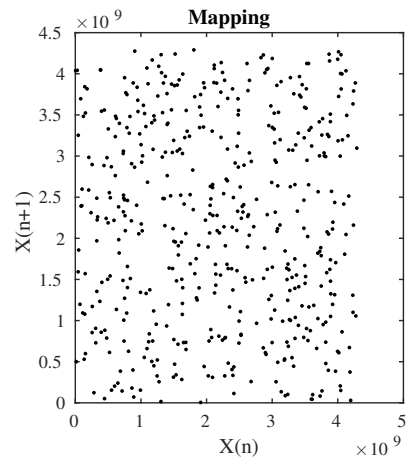
Test	P_value	proportion
frequency test	0.237	100.000
Block-frequency test	0.367	100.000
Cumulative-sums test	0.321	100.000
Runs test	0.384	100.000
Longest-run test	0.679	100.000
Rank test	0.514	98.000
FFT test	0.335	97.000
nonperiodic-templates	0.498	98.919
overlapping-templates	0.554	98.000
universal	0.994	99.000
approximty entropie	0.679	99.000
random-excursions	0.392	97.645
random-excursions-variant	0.435	98.712
serial test	0.564	98.500
linear-complexity	0.052	100.000

3.4.1.1 Computing Performance of the proposed PCNG

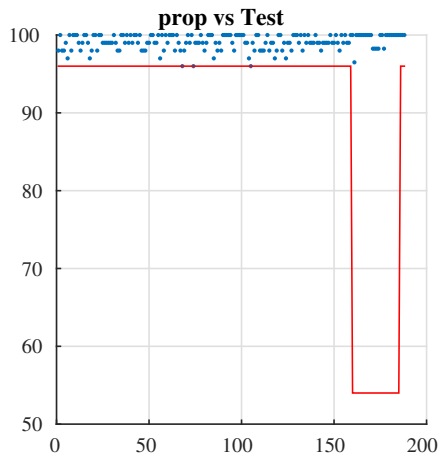
The experiment is conducted using a two 32-bit multicore Intel Core(TM) i5 processors running at 2.60 GHz with 16 Gb of main memory. This hardware platform was used on top of an Ubuntu 14.04 Trusty



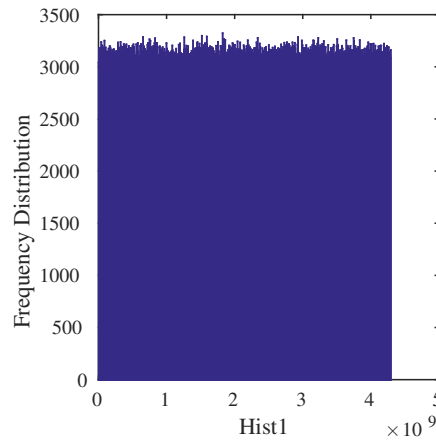
(a) mapping of the proposed PCNG



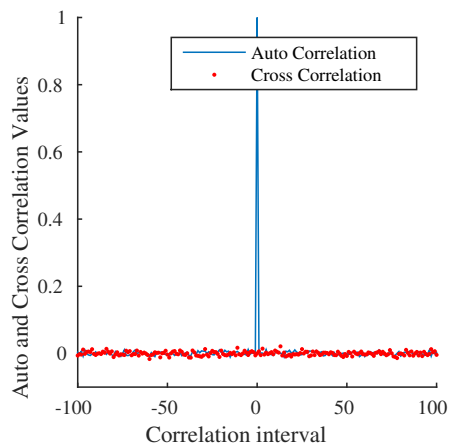
(b) zoom of the mapping



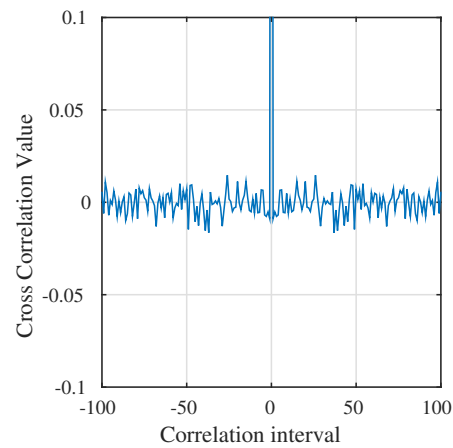
(c) NIST test of the proposed PCNG



(d) Histogram of the proposed PCNG



(e) Auto and Cross correlation of the proposed PCNG



(f) Zoom of cross correlation of the proposed PCNG

Figure 3.15 – Statistical tests of the proposed PCNG.

Linux distribution, and the programming is performed in code C. We provide below, for different sizes of data bytes, the average generation time in micro second $GT (\mu s)$, the average bit rate in Mega bit per second $BR (Mbit/s)$, and the average of the needed number of cycles to generate one byte, $NCpB (Cycles/B)$. The average is calculated by using 100 different secret keys. The obtained results are listed in Table 3.15. The computing performance is very close to Jallouli et al. [150] PCNG that was detailed previously Chapter 1.

Table 3.15 – Computing Performance of the proposed PCNG

Data (Bytes)	PCNG GT(μs)	PCNG BR(Mbit/s)	PCNG NCpB(Cycles/B)
64	6	101.33	132.2
128	8	190	110
256	11	201.17	98.5
512	19	255.18	78
1024	32	287	65.3
2048	57	301.42	62
4096	70	500.62	45
16384	232	584.32	40.2
32768	420	604.12	33.2
65536	712	736.36	26.9
125000	1182	790.03	25.4
196608	1730	959.49	23.1
393216	2902	1083.99	18.6
786432	4502	1243.15	16.3
3145728	17723	1357.49	14.1
12582912	75009	1481.15	14.8

3.4.1.2 Security analysis and statistical attacks

We report below first the security analysis in terms of key size, `keystream` attack and key sensitivity attack. Then, we give the obtained results of several statistical tests that were carried out in order to quantify the good statistical properties of the proposed PCNG.

Key size, Keystream attack and Key Sensitivity of the PCNG

The key size $|K1|$ of the proposed PCNG consists of all initial conditions and parameters of the proposed system and they are large enough to resist the brute force attack. Indeed, $|K1| = 4 \times 32 + 23 + 21 + 32 + 31 + 32 + 32 = 299$ bits. Also, as for each new execution, the produced keystream is totally different from the others due to the IVg value; so, the system can resist a keystream attack. Besides, key sensitivity is an essential property. This means, a small change in the secret key must cause a very big change in the output keystream. In order to verify this characteristic, we calculated the Hamming Distance (HD) of two sequences generated with only one bit change (lsb bit) in the parameter X_p . We calculate the average Hamming Distance HD between two sequences S_1 and S_2 , over 100 random secret keys. The $HD(S_1, S_2)$ is defined by the following equation:

$$HD(S_1, S_2) = \frac{1}{Nb} \times \sum_{K=1}^{Nb} (S_1(K) \oplus S_2(K)) \quad (3.27)$$

Where Nb is the number of bits in a sequence. The obtained average value of Hamming distance is equal to 0.499999 of the proposed PCNG. This value is close to the optimal value of 50%, which indicates its

criticality to the secret key.

Mapping and Histogram

The Mapping ($X(n + 1) = f(X(n))$) reflects the dynamic behavior of the system. As we can see in Figure 3.15, the resulting mapping of a given produced sequence by the PCNG seems to be random in comparison with a mapping (a signature) of a given known map. A good PCNG must produce sequences that have uniform distribution in the whole phase space. Visually, the obtained histogram in Figure 3.15 for a given generated sequence is uniform. To confirm this result we applied the Chi-Square and we obtained 991.962210 as an experimental value, which is smaller than the theoretical value of 1073.642651; then the histogram is uniform. Indeed, more the experimental value of Chi-Square is smaller than the theoretical one, better is the uniformity of the generated sequence.

Auto and Cross-correlation

Another good property of a PCNG is that, the generated sequences must be uncorrelated. Thus, the cross-correlation of two sequences x and y (generated with slightly different keys) must be close to zero. Figure 3.15 points out that the sequences from the PCNG are not correlated or repeated.

NIST Test

This test is a statistical package that consists of 188 tests and sub-tests that were proposed by NIST in order to assess the randomness of an arbitrarily long binary sequence. Figure 3.15 gives the results for sequences $X1$ generated by PCNG. We observe that, sequence $X1$ has successfully passed all the NIST tests. Therefore, the proposed PCNG is robust against statistical attacks. The security performance is better than Jallouli et al. [150] PCNG that was detailed previously Chapter 1.

3.5 Second Application: Proposed chaos-based stream cipher

In this section we present two versions (namely V2 and V3) of a synchronous stream cipher based on the previous proposed chaotic generator. Each version is implemented in sequential and parallel:

- V2¹. sequential implementation: After storing the plaintext data (in bytes), we call the generator r times (with $r = \text{Size_of_the_plaintext_data}/4$) and the generated samples are first converted to bytes and then xored with the plaintext data bytes for producing the ciphertext.
- V2 parallel implementation: In this version, after storing the plaintext data (in bytes), we create 4 threads (in parallel), each of the first 3 threads calls the generator r times and the 4th-thread calls the generator rm times (with rm is the remainder samples to produce). Then, after the worker threads waited each other for achieving thread synchronization by using the `pthread_join()` function, the generated samples are converted to bytes and then xored with the plaintext data bytes for producing the ciphertext (see Table 3.16).
- V3 sequential implementation: After storing the plaintext data (in bytes), we take each time 16 bytes and then we call the generator 4 times for producing 4 samples (each of 32 bits) that are converted immediately to 16 bytes, in order to be xored with the 16 bytes of the plaintext data, and so on.
- V3 parallel implementation: Here on each 16 plaintext data bytes, we create 4 threads that call (in parallel) the generator to produce 4 samples (one sample by a thread). Then, after the waiting process (synchronization between the 4 threads), the 4 samples are converted to 16 bytes and then xored with the 16 plaintext data bytes, and so on. In this version, the generator is called approximately r times,

1. Version 1 is the generator version

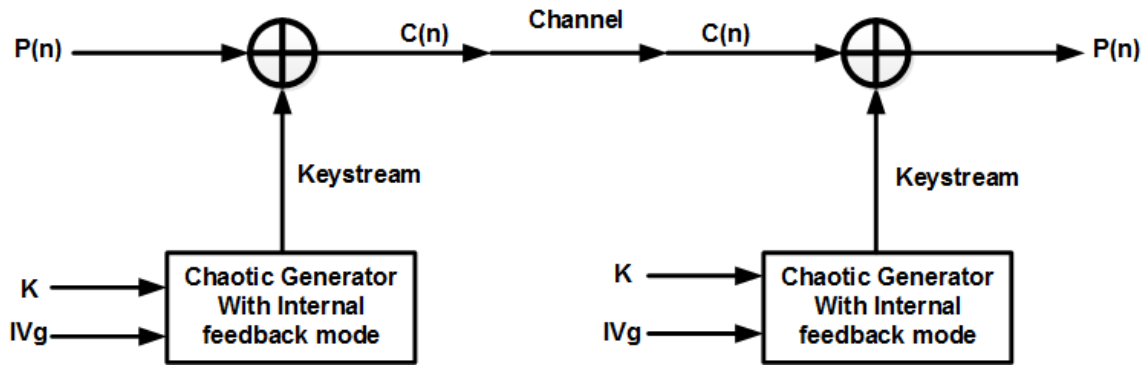


Figure 3.16 – Stream cipher encryption/decryption structure

each including one `pthread_join()` process. Consequently, the computing performance is very bad compared to the other versions, as indicated in Table 3.17.

As depicted in Algorithm 3, the encryption process starts by reading the secret key and IV value. Subsequently a creation of another three keys and IVs is achieved by a circular shift operation. After that, four threads are created using `pthread` in order to perform keystream generation in a parallel manner. The decryption process is performed by using the same secret key and IV in order to generate the same keystream that used before in encryption. A xoring operation is done between the `ciphertext` and the `keystream` to obtain the original `plaintext`. The encryption/decryption processes ensure the synchronization between sender and receiver because we use the same secret key and IV values in generator initialization. The general structure of the stream encryption and decryption processes is shown in Figure 3.16.

As with any encryption system, the secret key `K` and the initial IV vector must be shared between the sender and the receiver. The key must be kept secret while the IV vector is not necessarily kept secret but must be a nonce. The common method to share the secret Key between the two parties is a symmetric key distribution based on either symmetric encryption using a key distribution center (KDC) or asymmetric encryption using the RSA (Rivest, Adi Shamir and Leonard Adleman) algorithm [43]. The IVg is changed every new session as a key session.

3.5.1 Encryption computation performance and security analysis of the proposed stream cipher

3.5.1.1 Computation performance

The computation performance is determined by: the average encryption time $Enc_T(\mu s)$, the average encryption throughput $ET(Mbit/s)$ defined in Equation 3.28, and the average number of cycles to encrypt one byte $NCpB(Cycles/B)$ defined previously in Equation 3.14.

$$ET = \frac{Image_Size(Mbit)}{Encryption_Time(mus)} \quad (3.28)$$

We report in Table 3.16 and in Figures 3.17, 3.18, 3.19 the obtained results of the computation performance for sequential and parallel implementation of the proposed stream cipher (V2). The decryption time is approximately equal to the encryption time.

For data size, from 196608 bytes upwards, the parallel implementation becomes better than the sequential one and on average the `NCpB` of the stream cipher takes approximately 8 `cycles` more compared to the `NCpB` of the chaotic generator.

In Table 3.18, we report a comparison of computation performance for the proposed algorithm (for different data size images of Lena) with three chaos-based algorithms and the most Known stream ciphers of `eStream` project [151]. For big data, the proposed algorithm has better results than [55, 56]. We also observed that the computation performance of `eStream`'s algorithms is better than the proposed system

Algorithm 3 Chaos based stream cipher, main steps.

Data: Key,I**Data:** IV**Data:** Plain_text**Result:** X result**Result:** Cipher_textKey \leftarrow Compute recursive cell **for** nb_cores **do**| Key1 \leftarrow circular-shift(Key,3-bit)| Key2 \leftarrow circular-shift(Key1,3-bit)| Key3 \leftarrow circular-shift(Key2,3-bit)| IV1 \leftarrow circular-shift(IV,3)| IV2 \leftarrow circular-shift(IV1,3)| IV3 \leftarrow circular-shift(IV2,3)**end** $U_S \leftarrow$ 32bit-IV $U_P \leftarrow$ 32bit-IV**Function Generation**| $X_{skewtent} \leftarrow$ SkewTent function with $X_{-1skewtent}$ **if** $I\% \Delta_{skewtent} \text{ AND } \neq 0$ **then**| | $perturb_{skewtent} \leftarrow$ LFSR **if** $X_{-1skewtent} \oplus perturb_{skewtent} \neq 0$ **then**| | | $X_{-1skewtent} \leftarrow X_{-1skewtent} \oplus perturb_{skewtent}$ | | **end**| **end**| $X_{PWLC} \leftarrow$ PWLC function with X_{-1PWLC} **if** $I\% \Delta_{PWLC} \text{ AND } \neq 0$ **then**| | $perturb_{PWLC} \leftarrow$ LFSR **if** $X_{-1PWLC} \oplus perturb_{PWLC} \neq 0$ **then**| | | $X_{-1PWLC} \leftarrow X_{-1PWLC} \oplus perturb_{PWLC}$ | | **end**| **end**| $X \leftarrow X_{skewtent} \oplus X_{PWLC}$ $X_4byte \leftarrow X$ Shift Recursive cell $IV \leftarrow$ (Div/Urandom)| **End** Generation**Function Threading****for** nb_cores **do**| pthread_create($th[i]$, $NULL$, Generation, $(void^*)(intptr_t)i$)| **end****for** nb_cores **do**| pthread_join($th[i]$, $NULL$)| **end**| Cipher_text = Plain_text \oplus X_4byte

| return Cipher_text

End Threading

until we reach the big data size, for which, our system will be faster. For very big data size (201326592) such as videos, the obtained NCpB is around 9. In addition, the proposed chaotic system has a strong non-linearity compared to the other systems; thus, its robustness against cryptographic attacks is higher.

Table 3.16 – Performance results of proposed Stream Cipher (V2) with different data sizes

Data in Bytes	Enc-T (μ s) Seq/Parl	ET (Mbit/s) Seq/Parl	NCpB (Cycles/B) Seq/ Parl
512	21/ 778	213.01/ 5.31	92.9/ 3650.7
1024	33/ 792	259.1/ 11.1	78.2/ 1889
2048	60/ 806	286.5/ 19.9	70.2/ 973
4096	116/ 822	299.3/ 39.3	67.0/ 491.3
49152	659/ 1619	569.0/ 231.6	34.8/ 85.6
196608	2455/ 2419	610.9/ 620.0	31.9/ 31.2
786432	9088/ 8099	660.2/ 740.8	30.0/ 26.7
3145728	35560/ 24190	674.9/ 978.8	29.3/ 20.2
12582912	121899/ 88597	787.5/ 1083.5	25.1/ 18.3
50331648	398089/ 319785	964.6/ 1200.8	20.5/ 16.5

Table 3.17 – Performance results of proposed Stream Cipher (V3) with different data sizes

Data in Bytes	Enc-T (μ s) Seq/Parl	ET (Mbit/s) Seq/Parl	NCpB (Cycles/B) Seq/ Parl
512	18/ 1477.6	218.17/ 2.64	86.3/ 7503.5
1024	34.06/ 2830.4	229.37/ 2.76	86.5/ 7186.7
2048	41.27/ 5595.03	399.68/ 2.79	62.7/ 7103.
4096	120.78/ 11398.94	627.76/ 2.74	37.6/ 7235.7
49152	365.4/ 128031.5	826.13/ 2.93	24.3/ 6772.5
196608	1436.9/ 519951.48	1043.88/ 2.88	23.3/ 6876.0
786432	5339.4/ 2061207.47	1123.72/ 2.91	17.7/ 6814.5
3145728	21351.6/ 8366680.62	1124.03/ 2.87	17.6/ 6915.2

3.5.2 Security analysis of proposed chaos-based stream cipher

In this section we evaluated the security of the proposed chaotic system against cryptanalytic and statistic attacks.

3.5.2.1 Cryptanalytic attacks

The proposed chaotic system has the ability to resist common attacks such as ciphertext only [152], chosen plaintext attack and key sensitivity attack. Indeed, first, encrypting an image several times using the same secret key produces totally different ciphered images. This is due to the IV-setup block. Second, the chaotic generator is in one way a hash function.

Key Space

The size of the secret key, formed by all the initial conditions and by all the parameters of the system, varies from 299 bits, with delay = 1, to 555 bits, with delay =3. This means that the brute force

Table 3.18 – Performance results comparison of some stream ciphers

Stream cipher-Alg	Image size(B)	Enc-Time(μ s)	ET(Mbit/s)	NCpB(cycles/B)
Abderrahim et al.	-	-	10	2800
Hauping et al.	-	-	914	17
Ping et al.	-	-	700	20
Rappit	256x256x3	811.3	1848.8	9.5
	512x512x3	3256	1842.6	9.5
	1024x1024x3	12950	1853.9	9.5
HC-128	256x256x3	1221	1228.1	14.4
	512x512x3	4895	1225.6	14.4
	1024x1024x3	19647	1221.5	14.4
Salsa20/12	256x256x3	836.4	1793.4	9.8
	512x512x3	3389	1770	9.9
	1024x1024x3	13483	1779.9	9.9
SOSEMANUK	256x256x3	880.3	1704	10.3
	512x512x3	3570	1680	10.5
	1024x1024x3	14134	1698	10.4
AES-CTR	-	-	-	21.2
Proposed chaos stream cipher (Seq)	256x256x3	2455	610.9	31.9
	512x512x3	9088	660.2	30.0
	1024x1024x3	35560	674.9	29.3
Proposed chaos stream cipher (Parl)	256x256x3	2419	620	31.2
	512x512x3	8099	740.8	26.7
	1024x1024x3	24190	978.8	20.2
	201326592	1200178	1881	8.8

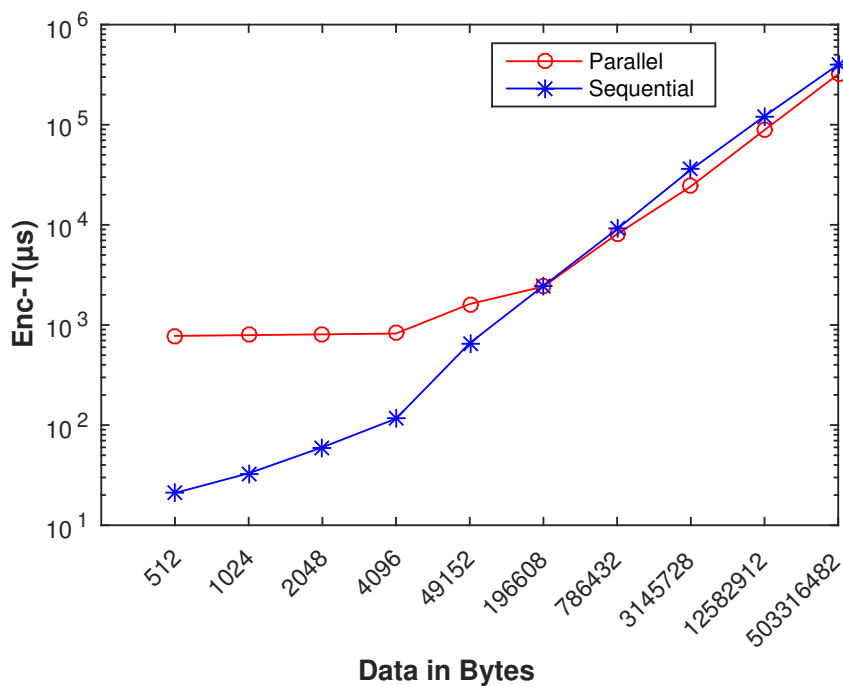


Figure 3.17 – Encryption time for parallel and sequential stream cipher.

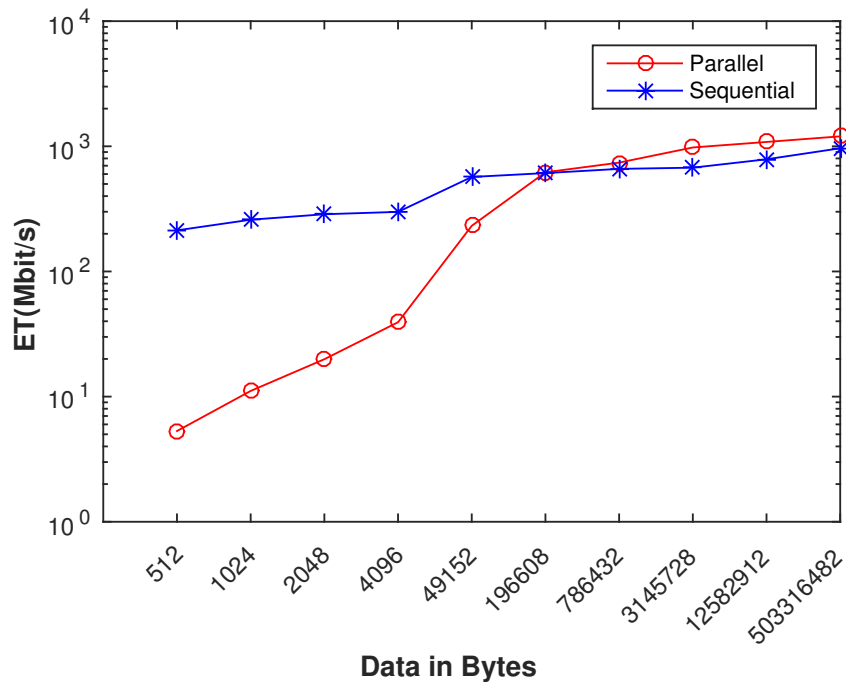


Figure 3.18 – Encryption throughput for parallel and sequential stream cipher.

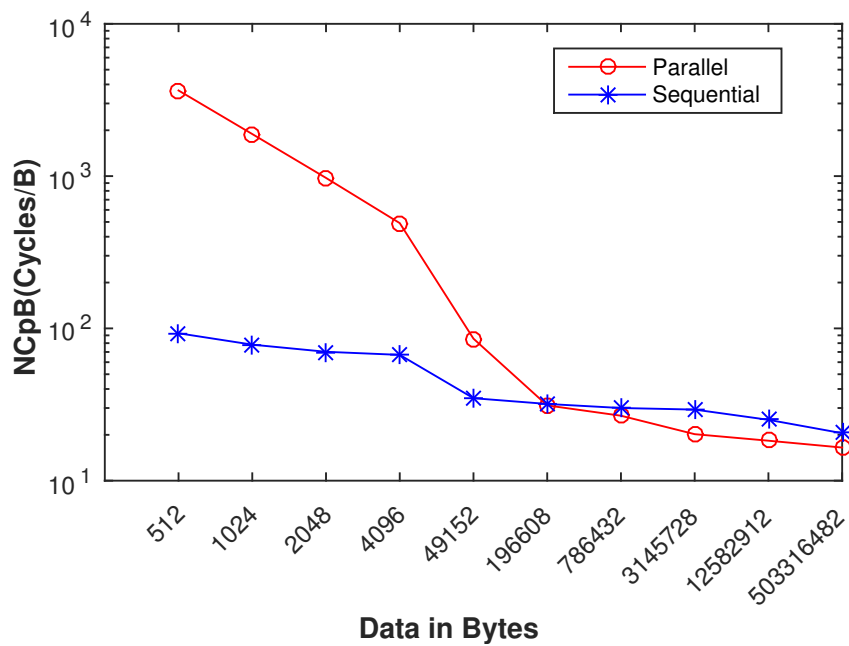


Figure 3.19 – NCpB for parallel and sequential stream cipher.

Table 3.19 – The NPCR, UACI and HD

Cryptosystem	NPCR	UACI	HD
Proposed Cipher Cryptosystem	99.665	33.459	0.499999

attack is impracticable.

Key security and sensitivity attack

From the generated sequences, it is impossible to find the secret key; this is because of the structure of the chaotic generator which also includes a chaotic switching. The knowledge of part of the secret key is not very useful for an attacker because of the intrinsic property of the chaotic signal, which is extremely sensitive to the secret key. Besides, we computed the average Hamming distance (of 100 secret keys) of two Keystreams generated each time with two secret keys that differ only by one bit and the result obtained is equal to 0.499993, therefore very close to 50%. In conclusion, the produced keystreams are highly secure.

A cryptosystem must be sensitive to one bit change per key used. This property is important in order to resist many attacks [24]. To test the key sensitivity of the proposed chaos stream cipher, we encrypted "Lena" image 100 times using 100 secret keys that differ only by the LSB bit. Then we computed the following parameters: the Number of Pixel Change Rate (NPCR), the Unified Average Changing Intensity (UACI) and the Hamming Distance (HD). The parameters (NPCR, UACI) are necessary but not sufficient to ensure that the proposed cryptosystem is resistant against the key sensitivity attack. For this reason, we added the Hamming Distance measurement [153].

The NPCR and UACI, introduced by Eli Biham and Adi Shamir [6] are given by the following equations:

$$NPCR = \frac{1}{L \times C \times P} \times \sum_{p=1}^P \sum_{i=1}^L \sum_{j=1}^C D(i, j, p) \times 100\% \quad (3.29)$$

where

$$D(i, j, p) = \begin{cases} 0, & \text{if } C_1(i, j, p) = C_2(i, j, p) \\ 1, & \text{if } C_1(i, j, p) \neq C_2(i, j, p) \end{cases} \quad (3.30)$$

$$UACI = \frac{1}{L \times C \times P \times 255} \times \sum_{p=1}^P \sum_{i=1}^L \sum_{j=1}^C |C_1(i, j, p) - C_2(i, j, p)| \times 100\% \quad (3.31)$$

In the previous equations, i , j and p are the row, column, and plane indexes of the image, respectively. L , C and P are, the length, width, and plane sizes of the image respectively. The optimal NPCR and UACI values are 99.61% and 33.46% respectively [154].

The HD is defined in Equation 3.27, in which N_b represents the size of the image in bits that equal to $L \times C \times P \times 8$. The optimum HD value is 50%. A good stream cipher should produce an HD close to 50% [155]. Table 3.19 indicates that the NPCR, UACI and HD values of the proposed stream cipher are very close to the optimal values.

3.5.3 Statistical analysis of the proposed chaos stream cipher

3.5.3.1 NIST Test

To evaluate the statistical performances of the cipher-image produced, the NIST statistical test was also used [145]. We applied the NIST test to many ciphered texts; all the NIST results obtained, are as expected (good NIST values). In Figure 3.20 we present one of the NIST results obtained. This means that the ciphered texts have a high randomness.

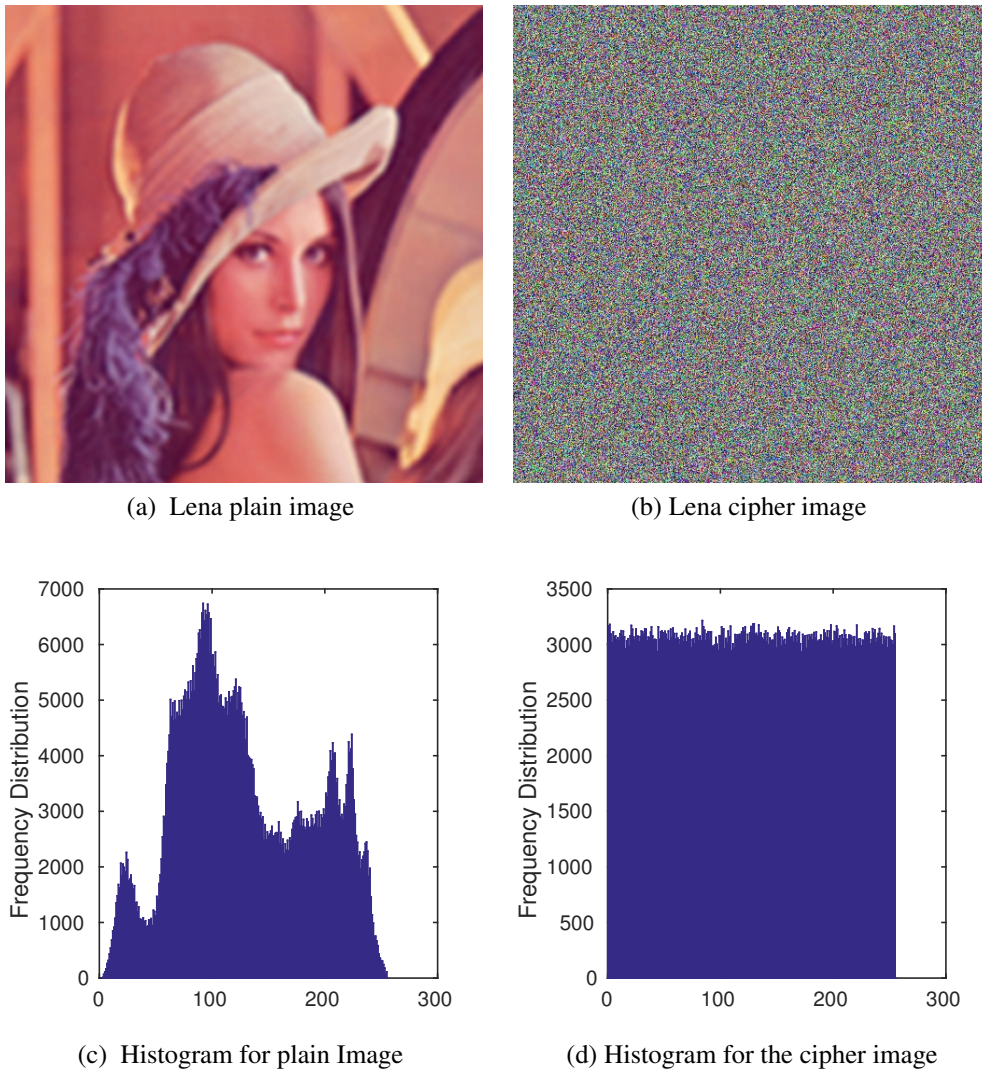


Figure 3.21 – Histogram of the lena plain image and its ciphered image

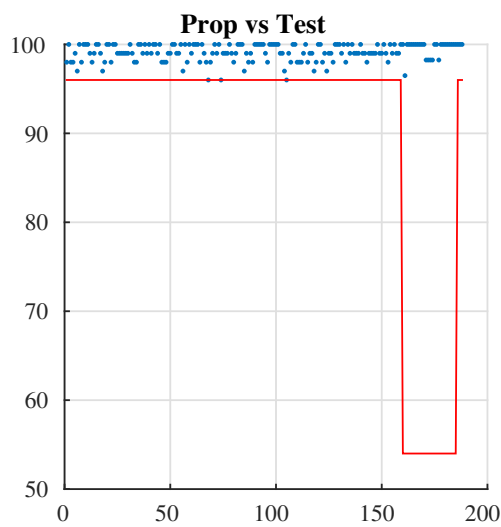


Figure 3.20 – NIST test cipher-image results.

3.5.3.2 Histogram and Chi-square test

A cryptosystem is considered to be strong against statistical attacks, if the histogram of the ciphered text is uniformly distributed. Visually, the uniformity test is necessary, but it is not sufficient. The chi-square test is applied to statistically confirm the uniformity of the histogram:

Table 3.20 – Chi-square value of histograms for different ciphered/plain images with different sizes

Image	Experimental value	Theoretical value
Lena 256x256x3	261.085938	293.247835
Lena 512x512x3	263.013852	293.247835
Lena 1024x1024x3	270.300127	293.247835
Boat 256x256x3	260.186354	293.247835
Boat 512x512x3	266.465369	293.247835
Boat 1024x1024x3	272.669811	293.247835
C-man 256x256x3	259.339680	293.247835
C-man 512x512x3	267.317852	293.247835
C-man 1024x1024x3	274.397541	293.247835
Peppers 256x256x3	259.963257	293.247835
Peppers 512x512x3	266.357961	293.247835
Peppers 1024x1024x3	273.386931	293.247835

The Equation 3.15 is used to perform the Chi test with the following parameters: K is the number of levels (here $K = 256$), O_i is the observed occurrence frequency of each color level (0-255) on the histogram of the ciphered image, and E_i is the expected occurrence frequency of the uniform distribution, given here by $E_i = \frac{L \times C \times P}{K}$. For a secure cryptosystem, the experimental chi-square value must be less than the theoretical chi-square one, which is 293 in case of $\alpha = 0.05$ and $K = 256$. In Figures 3.21, 3.22, 3.23 and 3.24 we give the histograms for the plain/cipher images for Lena, Boat, Camera man and Peppers in size 512*512*3. As we can see, the histogram of the ciphered image seems to be uniform. To assess the uniformity, we performed the chi square test. Experimental value obtained is less than the theoretical one at 293. This means that the histograms are uniform (see Table 3.20).

3.5.3.3 Correlation analysis

Correlation analysis is also one of the statistical attacks that are used to cryptanalyze the cryptosystem. The attacker should not have any information of the used secret key or any partial information about the original plain image. This means that the encrypted image should be extremely different from its original version. Correlation analysis is one of the common and standard methods to measure this property. Indeed, it is well-known that adjacent pixels in the plain images are very redundant and correlated. Thus, in the encrypted images, adjacent pixels should have a redundancy and a correlation as low as possible. The Equations 3.22, 3.23, 3.24 and 3.25 are used to calculate image correlation.

To test the security of our proposed stream cipher algorithm, in relation to this type of attack, first $N = 10000$ pairs of adjacent pixels in vertical, horizontal, and diagonal directions are selected from the plain image and its ciphered version. Figure 3.25 shows the correlation curves of the adjacent pixels in the horizontal, vertical and diagonal direction for the Boat plain image and its ciphered one. The correlation coefficient values for all previous tested plain/cipher images are given in Table 3.21. As we can expected, these results conform to those found in the literature.

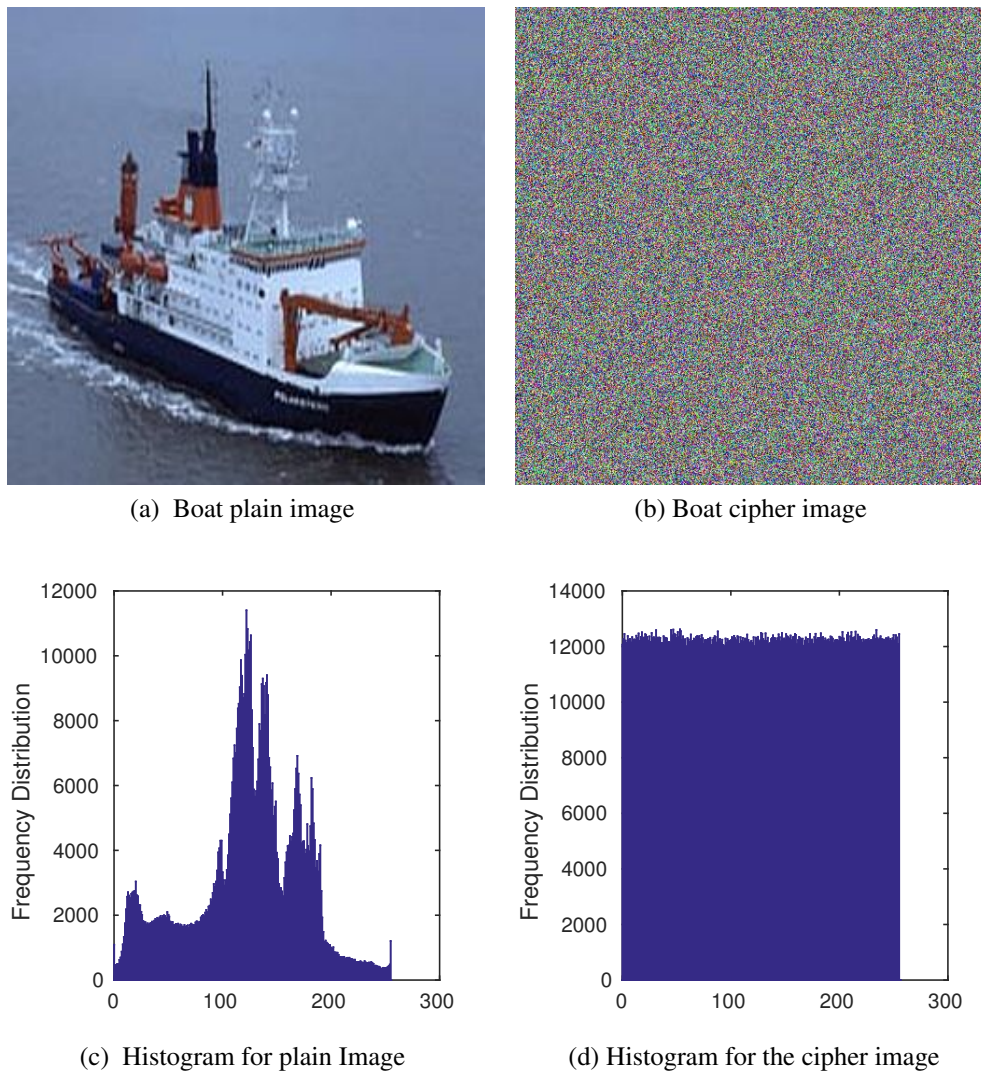


Figure 3.22 – Histogram of the Boat plain image and its ciphered image

Table 3.21 – Correlation coefficient values for the previous plain/cipher images.

Plain/cipher image	Horizontal	Vertical	Diagonal
Lena	0.96606/0.035	0.96613/0.026	0.96619/0.027
Boat	0.99605/0.022	0.99703/0.019	0.99671/0.020
Camera man	0.96618/0.036	0.96771/0.028	0.96767/0.022
Peppers	0.96608/0.019	0.96612/0.031	0.96647/0.011

3.6 Software security implementation

In cryptographic applications, sensitive data (e.g., secret keys) must be stored in memory for the minimum amount of time possible and should be written over/deleted, not just released, when no longer needed. In order to assess the security of our implemented code a Software security analysis is used, to attempt to evaluate the security of the system by safely trying to exploit vulnerabilities. These vulnerabilities may exist in memory buffer storage, threads calling and joining or all of the code instructions. Such assessments are also useful in validating the efficacy of defensive mechanisms, as well as, end-user adherence to security

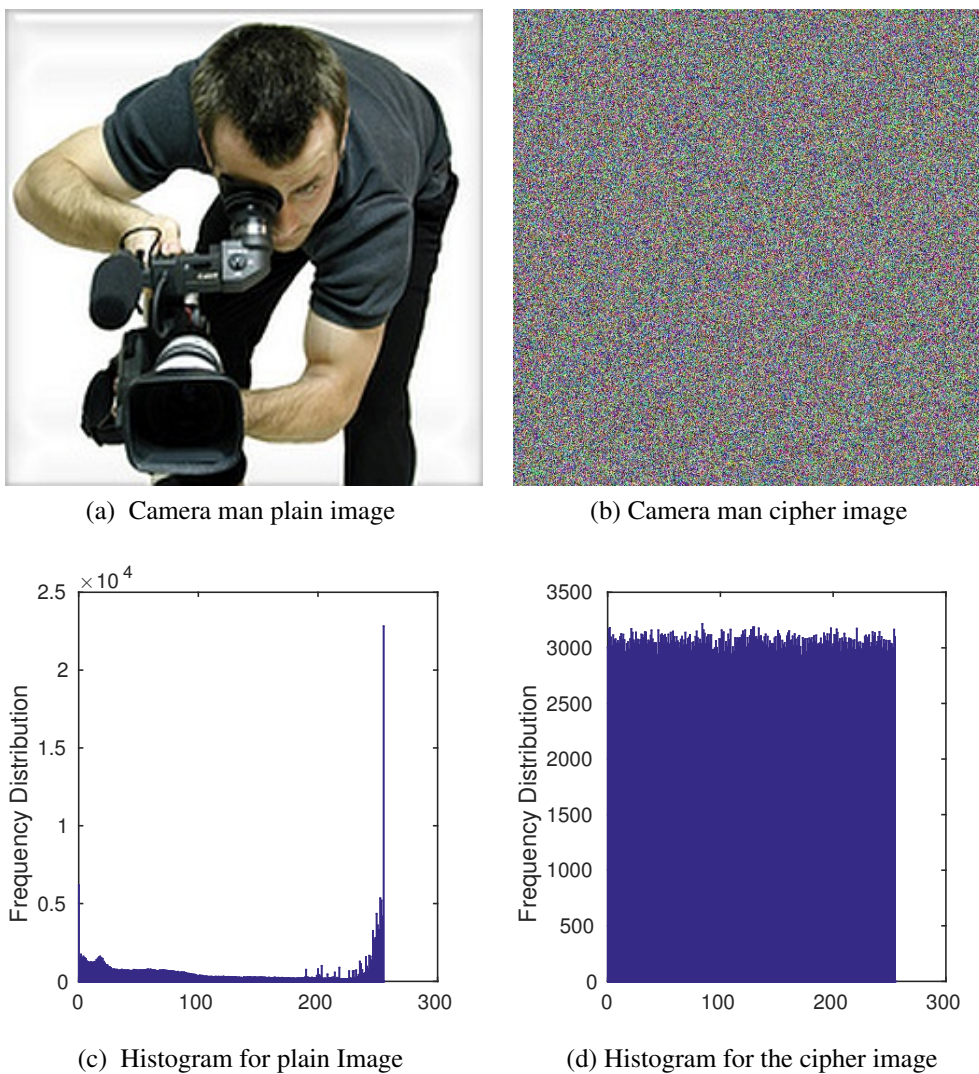


Figure 3.23 – Histogram of the Camera man plain image and its ciphered image

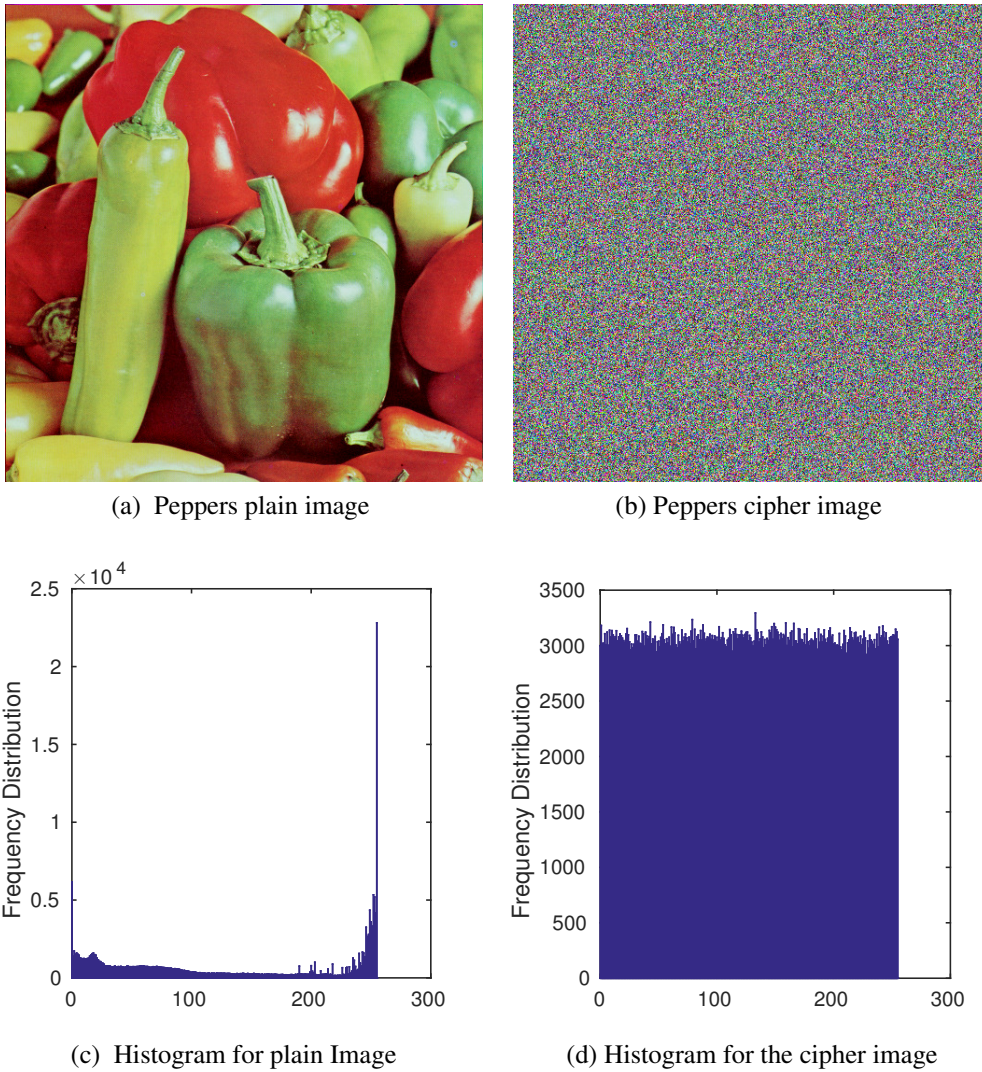


Figure 3.24 – Histogram of the Peppers plain image and its ciphered image

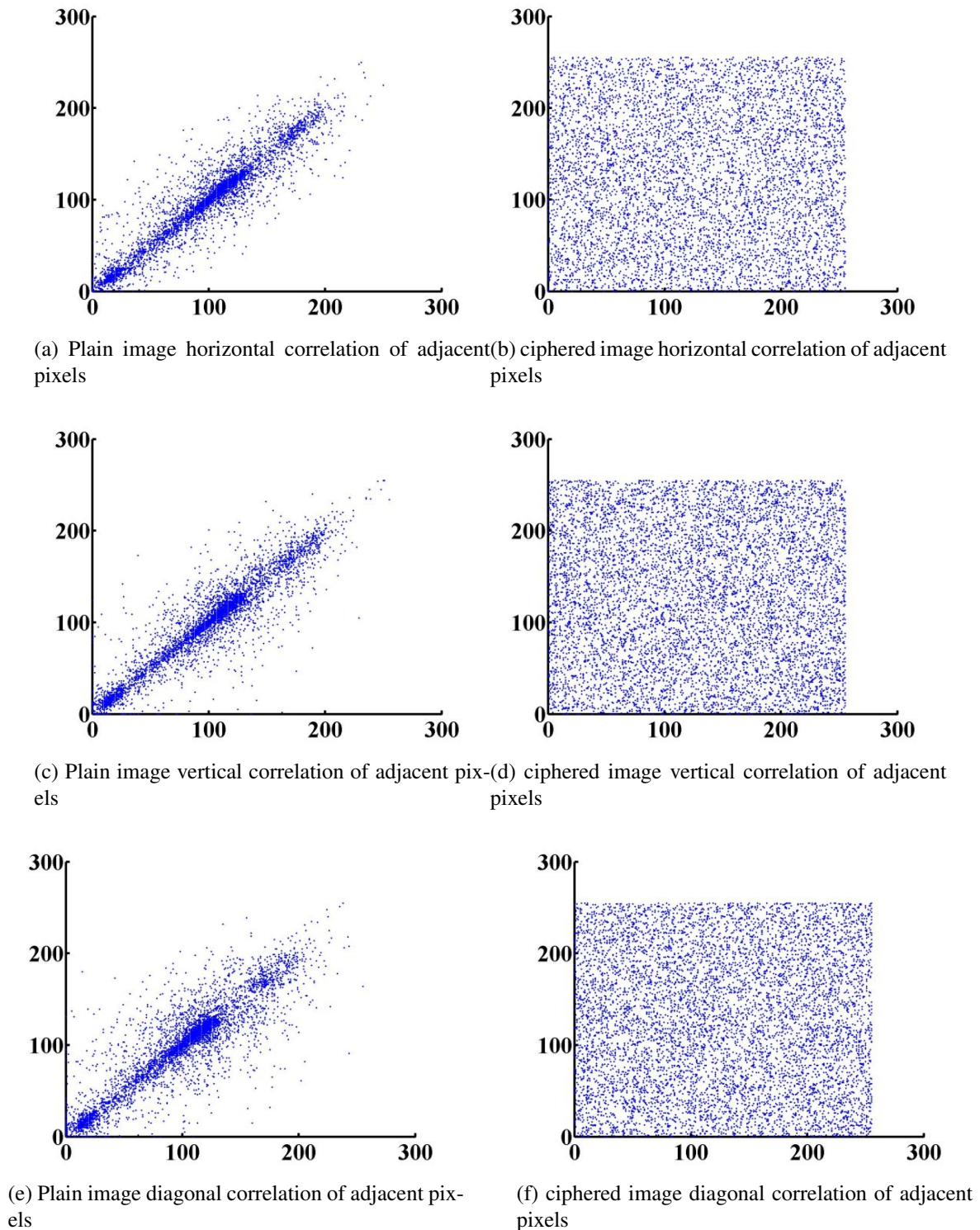


Figure 3.25 – Horizontal, Vertical and Diagonal correlation of the Boat plain image and its ciphered image

policies. In order to guarantee the validity of our solution, we carried out a security code review using several static and dynamic techniques: Clang, Gdb, Valgrind, DRD, Callgrind and Leak-analysis tools. Results match up well with the security level requested by our chaos-based stream cipher [149]. Because it is still possible to access data out of memory even if the application no longer has pointers to it, it is important to deploy data security within the source code. In practice, we used the following functions to decontaminate (i.e., zero) a buffer and guarantee that the compiler will not optimize it away: The `secure_memzero()` function depends on a function pointer `memset_ptr` that itself points to the `memset()` function. It

uses the key and the key size and will put zero value on the allocated memory related to the key by call `memset()`. The function `memset()` is invoked to write a specific value in a buffer that was allocated previously. We used this function to write a zero value in the buffer. While some compilers optimize away the call to `memset()` function, to overcome this, we declared `memset_ptr` as a *volatile* pointer. Since a *volatile* pointer can be manipulated outside the scope of the application, the code is not optimized by the compiler, thus keeping the program unchanged. Furthermore, the data in main memory may leak to the disk through virtual memory, thus representing another source of the most serious leaks (leaks to physical mediums). One solution, which is sufficient to include, is to deactivate the swap space altogether, thus preventing data from being written to the page file by locking it in memory. In our code, we used the `mlock()` function that locks pages in the address range starting at the address and continuing for byte lengths. All pages that contain a part of the specified address range are secured in the main memory when the call returns successfully. This way, the pages are secured in the main memory until they are unlocked later.

In Chapter 2, mainly in section 2.6.1 we elaborated the software security tools used to review the code in our applications. The usage of this tools to check our C code are described in Appendix B.

3.7 Conclusion

We designed and implemented, in an efficient and secure way a chaos-based generator. Its structure is modular, generic, and allows the production of highly secure sequences. Based on the chaotic generator (PCNG), two application were designed, implemented and analysed. The first application deals with the realization of a RNG based PCNG, and the obtained results are very satisfactory. The second application concerns the realization of a chaos-based stream cipher. The computation performance for the proposed generator is better than other known PRNG. Also, for very big data size, the obtained performance results are better than other known stream ciphers. The proposed chaotic system is robust against known cryptographic attacks. Furthermore, it has strong non-linearity compared to the other systems. Indeed, the results obtained from the cryptographic analysis and of common statistical tests indicate the robustness of the proposed stream cipher.

In the next chapter we focus on the design of real-time chaos-based joint crypto-compression systems to secure videos: an HEVC bitstream.

Second Contribution: Real Time Selective Encryption in the HEVC Standard

4.1 Introduction

HEVC is currently the newest video coding standard issued by the ITU-T Video Coding Experts Group and the ISO/IEC Moving Picture Experts Group. The main important object of the HEVC standardization effort is to permit appreciably improved compression performance relative to existing standards. Nowadays Video encryption is a hot research topic [2]. In the up-coming years, HEVC standard is expected to be increasingly adopted with the perspective to replace the previous video compression standard. Security and confidentiality of multimedia contents become a challenging research topic, which was widely investigated in the last decade. [19, 20, 21, 75, 76, 77, 78, 79, 80, 81].

The most straightforward method for content security is to encrypt the whole bit-stream or the most informative part, called *Region of Interest (ROI)*. This approach [75], treats the video bit-stream as a simple text data without taking account the structure of the compressed video. Bit-streams encrypted with this method are decodable only after a correct decryption event, when only parts of the video are encrypted. This process limits the usage of the content to only users who have the right permission on the encrypted parts. Moreover, these algorithms are time and energy consuming and not suitable for real-time video applications. Consequently, Selective Encryption (SE) has emerged as an effective solution to overcome these full encryption drawbacks [156, 157].

The aim of SE is to reduce the amount of data to encrypt while preserving a sufficient level of security. Thus, only the most sensitive information in the bitstream is encrypted. In this chapter also, we focus on SE that hides only the ROI in the video (human faces, personal data, etc.) and keeps the rest of the video (background) clear. In our approach, the HEVC video is first split into independent rectangular regions called tiles [79] and then only the tiles belonging to the ROI are encrypted.

The proposed solution encrypt a set of HEVC parameters including *Motion Vector (MV)* differences, MV-signs, *Transform coefficients (TCs)*, TC-signs, as given in [80]. Besides, we propose format compliant encryption solution of the luma and chroma Intra Prediction Modes (IPMs). The selective encryption is performed using the chaos-based stream cipher, introduced in Chapter 3 [138, 149]. The proposed solution of ROI allows to prevent the encryption propagation outside the ROI in Intra and Inter coding configurations. Finally, for real-time character, the encryption and decryption processes are implemented in the real-time Kvazaar HEVC [158] encoder and the openHEVC decoder [159], respectively.

The rest of this chapter is organized as follows. The proposed selective encryption of IPM and ROI

encryption in HEVC are investigated in Sections 4.3 and 4.4, respectively. Performance evaluations and associated results analysis are given in Section 4.5. Finally, Section 5 concludes the chapter and gives some perspectives about future works.

4.2 Real time encoder: Kvazaar

Kvazaar is an video encoder for the newest video coding standard the(HEVC/H.265) standard. It gives the users a free, cross-platform HEVC encoder for x86, x64, PowerPC, and ARM processors on Windows, Linux, and Mac. Kvazaar is being construct from scratch in C and optimized in Assembly under the LGPLv2.1 license. The development is being organized by Ultra Video Group at Tampere University of Technology (TUT) and the implementation work is performed by an active community on GitHub. Developer friendly source code of Kvazaar makes simple joining for new developers. Indeed, Kvazaar includes all fundamental coding tools of HEVC and its modular source code facilitates parallelization on multi and manycore processors as well as algorithm acceleration on hardware. Kvazaar is able to accomplish real-time HEVC coding speed up to 4K video on an Intel 14-core Xeon processor. Kvazaar is also supported by FFmpeg and Libav. These standard of multimedia frameworks boost Kvazaar popularity and enable its joint usage with other well-known multimedia processing tools. Kvazaar has got a key role in three Eureka Celtic-Plus projects in the fields of 4K TV broadcasting, virtual advertising, Video on Demand, and video surveillance [158].

4.3 Proposed video encryption system

4.3.1 Encryption of intra prediction parameters

To the best of our knowledge, this is the first work encrypting luma and chroma IPMs of HEVC prediction parameters. In HEVC, there are three scanning orders of the quantized TCs and the scanning order is derived for Intra coded blocks from the IPM. The proposed algorithm encrypts the IPM without changing the original scanning order of the modes (the order before encryption). This enables the IPM encryption to be format compliant with HEVC and can be decoded with any standard HEVC decoder. The proposed encryption solution of IPMs is performed as shown by Algorithm 4. First, the IPM elements of HEVC are classified into three sets of modes: $Set_VER \in \{6, 7, 8, 9, 11, 12, 13, 14\}$, $Set_HOR \in \{22, 23, 24, 25, 27, 28, 29, 30\}$ and $Set_DIA \in \{0, 1, 2, 3, 4, 5, 15, 16, 17, 18, 19, 20, 21, 31, 32, 33, 34\}$. Each set contains the prediction modes that share the same scanning direction (*horizontal, vertical or diagonal*). The encryption process is carried out using a circular shift operation. Each IPM, in a particular set, is shifted according to a key stream bits. The stream values, required to the encryption process, are produced by a chaos based generator. Then, a new IPM position is deduced inside the same set. The derivation process of the chroma mode must rely on the encrypted luma mode to ensure format compliant encryption. The luma and chroma IPMs are encrypted in the same manner. Table 4.1 shows the encryptable bits of chroma IPMs (the red color bins are context coded in CABAC). The encryption process is fully format compliant, since we keep the scanning directions unchanged. Unlike the encryption of other syntax elements, the encryption of the IPMs is performed before the entropy coding and, thus, may decrease the coding rate-distortion (RD) performance. Figure 4.1 shows the IPMs in HEVC standard.

4.3.2 CABAC level encryption

A fast and secure selective chaos-based crypto-compression system is realized to encrypt the most sensitive information in the video contents. Selectively encrypted HEVC bitstream will fulfill real-time constraints (format compliant, fast, secure and constant bit rate). A group of sensitive HEVC parameters is selected to be used as input for this selective encryption solution including: MVs, MV signs, (TCs), TC

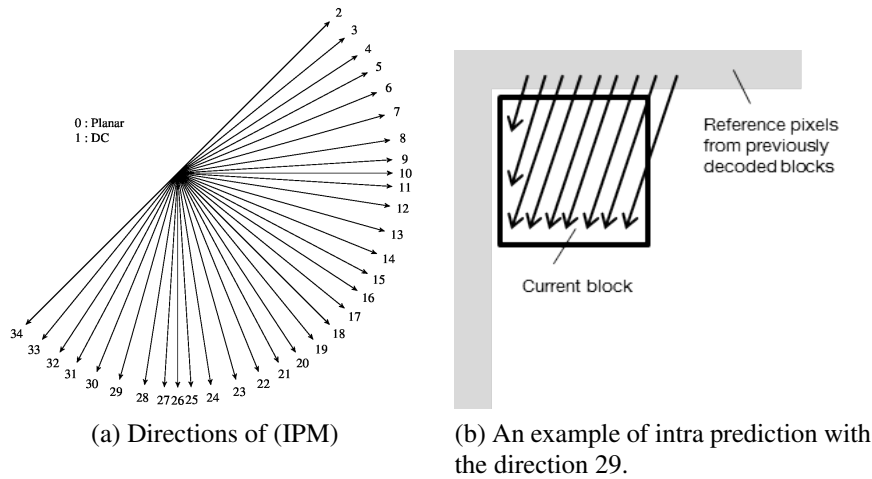


Figure 4.1 – Intra Prediction Modes (IPMs) in the HEVC standard.

Algorithm 4 IPMs encryption.

Input: *Intra Prediction Mode IPM*

Output: *Encrypted Intra Prediction Mode E_IPM*

- 1: $Set_VER \in \{6, 7, 8, 9, 10, 11, 12, 13, 14\}$
 - 2: $Set_HOR \in \{22, 23, 24, 25, 26, 27, 28, 29, 30\}$
 - 3: $Set_DIA \in \{0, 1, 2, 3, 4, 5, 15, 6, 17, 18, 9, 20, 21, 31, 32, 33, 34\}$
 - 4: Call chaotic generator to produce bit steam **K**
 - 5: **if** $IPM > 5$ **And** $IPM < 15$ **then**
 - 6: **E_IPM**=Circular shift(*Set_VER*, **IPM**, **K**)
 - 7: **else if** $M > 21$ **And** $IPM < 31$ **then**
 - 8: **E_IPM**=Circular shift(*Set_HOR*, **IPM**, **K**)
 - 9: **else**
 - 10: **E_IPM**=Circular shift(*Set_DIA*, **IPM**, **K**)
 - 11: **end if**
-

Table 4.1 – Encrypted bit of the chroma intra prediction mode.

Intra_chroma_pred_mode	Encrypted Bin value
<i>Intra_Derived</i>	0
<i>Planar</i>	100
<i>Angular(mode-26)</i>	101
<i>Angular(mode-10)</i>	110
<i>DC</i>	111

signs. As illustrated in Figure 4.2, the format compliant selective encryption is performed for the particular syntax elements between binarization and arithmetic coding. The image-based encryption/decryption algorithms (all frame) are integrated in the HEVC reference software (HM) version 16.7 [160] encoder/decoder respectively. Table 4.2 shows the encrypted syntax elements that performed in this work.

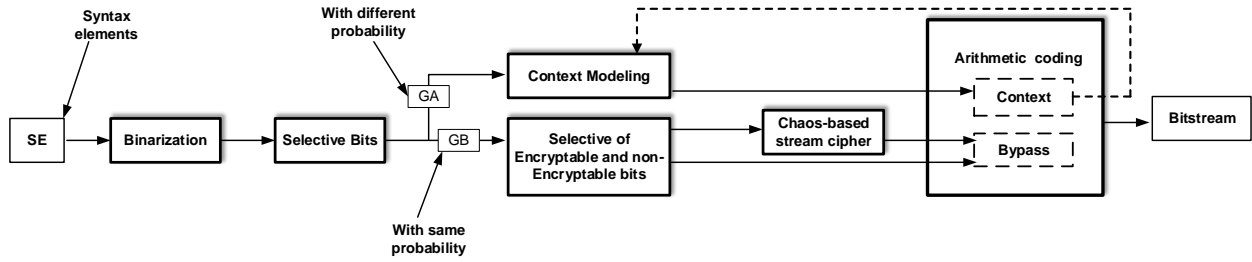


Figure 4.2 – Selective encryption in HEVC at CABAC level.

Table 4.2 – Encrypted syntax elements.

Syntax elements	Encrypted Part
<i>MV differences</i>	Suffix
<i>MV-sign</i>	1 bin
<i>TCS</i>	Algorithm 3.1 in [75]
<i>TC-sign</i>	suffix
<i>Luma</i>	Algorithm 4
<i>Chroma</i>	Algorithm 4

4.3.3 Chaos-based encryption system

For a given syntax element, the *chaotic generator* produces the necessary key-streams to obtain the ciphering data. The applied key-stream generator proposed in our previous work [138, 149] is used. The internal state, which contains the main cryptographic complexity of the system, is formed by two recursive filters of three-order. The first recursive cell contains a discrete Skew tent map and the second one contains a discrete piecewise linear chaotic map. These maps are used as non-linear functions. A new *IV* value is generated in each generator call, this value enables to produce different bits key-stream sequence on each generator call. The cryptographic security analysis of the key stream generator is detailed in Chapter 3 [138].

The encryption of syntax elements at the CABAC level, including *MV differences*, *MV-signs*, *TCs*, *TC-signs*, is given by the the following formula:

$$C_i = P_i \oplus X_i \quad (4.1)$$

where $P(i)$ denotes the syntax elements, $C(i)$ the ciphered syntax elements and $X(i)$ the key stream bits. Furthermore, the luma and chroma IPMs encryption is performed as follows:

Let N be the number of IPMs modes elements in the set vector $V = [1, 2, \dots, N]$, $V \in \mathbb{R}^N$, n_b the number of bits produced by chaotic generator and i the IPM index. The new value, $V_s[i]$, produced at the i^{th} position of IPM is given by Equation 4.2.

$$V_s[i] = V[(i + n_b) \bmod N] \quad (4.2)$$

The decryption algorithm is performed by inverse operations of Equations 4.1 and 4.2. Finally, the encoder and the decoder must share the same secret key K , used to initialize the chaotic generator.

4.4 ROI encryption in HEVC

In this section we propose a new encryption solution of ROI based on the tile concept in the HEVC video to protect privacy.

4.4.1 Tile-based encryption system

The proposed ROI encryption is based on the tile concept introduced in HEVC. This mechanism splits the video frame into different rectangles with integer number of blocks, where Intra prediction and entropy coding dependencies are broken at the tile boundaries. Tiles are used also to get more parallel processing or to prevent from error transmissions. The proposed solution performs a selective encryption of ROI tiles at the CABAC bin-string level. The most sensitive HEVC syntax elements are encrypted in order to reduce the visual quality of the involved ROI. The selective process encrypts only the tiles containing the ROI, whereas the non ROI tiles remain clear (not encrypted). A set of HEVC parameters, including MVs, MV signs, (TCs), and TC signs, are then encrypted as stated above. This is done in HEVC format compliant without increasing the bitrate of the encrypted video. In addition to these four parameters, we integrated the HEVC compliant encryption of IPMs, which may introduce a slight increase in bit rate, as clarified in section 4.3.

4.4.2 Encryption propagation in inter video coding

The merge mode in HEVC derives the *MVs* information from a list of spatial neighbouring and temporal candidates. Therefore, these two decoding operations can propagate the encryption from the encrypted tiles to the background, when the ROI is not correctly decrypted. Thus, we restrict the temporal candidates of the background tiles to be inside the background zone in the reference frame. In this case, in order to prevent the propagation of encryption outside the ROI tile, two non-normative encoding constraints are enforced by the Kvazaar encoder (as shown in Figure 4.3):

1. The *MVs* in the reference frame are restricted to point only to the co-located tile of the predicted block.
2. The in-loop filters are disabled across the tile boundaries.

These constraints tend to have a negative impact on the rate distortion performance, depending on the resolution, tiling configuration and the video content. In the contrary, they enable to perform a safe interpolation process at the tile boundaries.

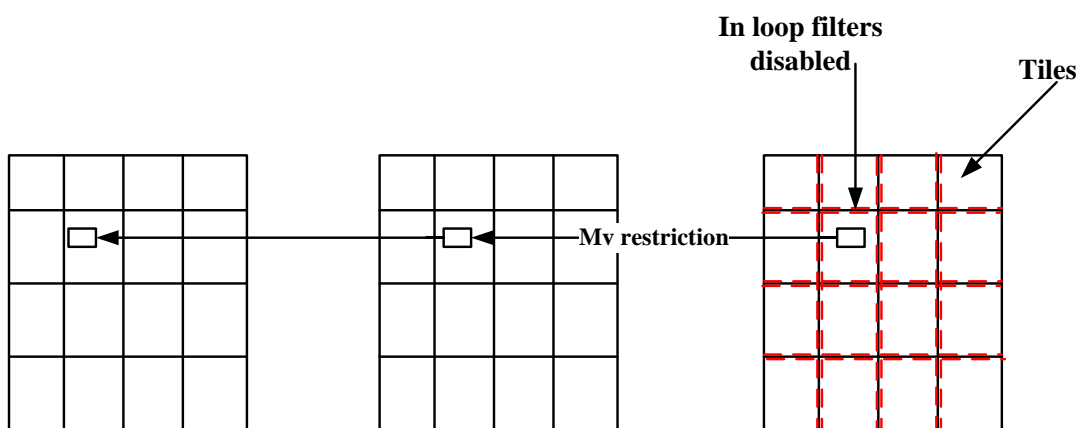


Figure 4.3 – MVs and in-loop filter restrictions.

Table 4.3 – The set of benchmark video sequences used in the experiment.

Sequence	Class	Resolution	Frame Rate
<i>PeopleOnStreet</i>	A	2560 × 1600	30
<i>Kimono</i>	B	1920 × 1080	24
<i>BasketballDrive</i>	B	1920 × 1080	50
<i>ParkScene</i>	B	1920 × 1080	24
<i>Cactus</i>	B	1920 × 1080	50
<i>BQSquare</i>	B	1920 × 1080	60
<i>Vidyo1</i>	E	1280 × 720	60
<i>Vidyo3</i>	E	1280 × 720	60
<i>Vidyo4</i>	E	1280 × 720	60
<i>FourPeople</i>	E	1280x720	60
<i>BQSquare</i>	D	416x240	60

4.5 Results and discussions

4.5.1 Experimental setup

The frame-based selective encryption scheme is implemented with HM version 16.7 [160], in `Main Intra` and `Random Access` configurations. In other hand, the ROI-based encryption and decryption algorithms are implemented in the real time Kvazaar HEVC encoder and OpenHEVC decoder, respectively. Eleven video sequences, from different classes and categories are used in this experiment; as given in Table 4.3. These videos, of 10 seconds duration each, are mainly taken from HEVC common test conditions [83]. They are simultaneously encoded and encrypted, in both `Intra` and `Inter` coding configurations, at four Quantization Parameter (*QP*) values $\in \{22, 27, 32, 37\}$. The encrypted videos are encoded with two uniform tiling configurations: 4×3 (i.e. four horizontal by three vertical repartition) and 4×4 . The same encoder configuration, without tiles and encryption, is used as an anchor. The processor used in these evaluations has 32-bit multi-core Intel Core (TM) i5 processor, running at 2.60 GHz with 16GB of main memory. The operating system is Ubuntu 14.04 Trusty Linux distribution.

It is important to note here that two HEVC plat-forms are used in this study (HM and Kvazaar/OpenHEVC). Firstly, the selective encryption (all frame) and subjective experiment are performed under the HM encoder/decoder. Several measures have then used (PSNR, SSIM, IPMs BD-rate evaluations, Edge Detection Ratio and Encryption Quality). Secondly, the ROI-based encryption is implemented with Kvazaar encoder/OpenHevc decoder and other encryption metrics have been used (BD-rate, complexity evaluations) with PSNR and SSIM.

In the following sections, we describe in detail the proposed solution performance based on two main criteria: objective measurements and subjective evaluations.

Table 4.4 – PSNR and SSIM values between original and encrypted videos (QP = 22).

Sequence	Original		Encrypted ROI	
	PSNR	SSIM	PSNR	SSIM
<i>PeopleOnStreet</i>	42.8	0.93	11.2	0.23
<i>Kimono</i>	42.2	0.96	9.9	0.22
<i>ParkScene</i>	43.3	0.91	10.7	0.20
<i>Cactus</i>	42.5	0.94	10.4	0.23
<i>BQTerrace</i>	41.8	0.90	10.8	0.24
<i>BasketballDrive</i>	41.5	0.96	10.1	0.23
<i>Vidyo1</i>	45.2	0.92	11.3	0.21
<i>Vidyo3</i>	44.6	0.94	10.9	0.20
<i>Vidyo4</i>	44.7	0.90	11.1	0.22

Table 4.5 – PSNR and SSIM values for three video sequences with different QP.

Sequence	QP	Original-PSNR			SE-PSNR			Original-SSIM			SE-SSIM		
		Y	U	V	Y	U	V	Y	U	V	Y	U	V
<i>BasketballDrive (B)</i>	22	42.1	43.5	44.9	10.2	10.8	11.1	0.92	0.94	0.94	0.21	0.22	0.24
	27	41.3	42.2	43.6	10.1	10.7	11.0	0.89	0.91	0.93	0.2	0.21	0.21
	32	37.5	38.8	39.1	9.8	10.5	10.9	0.76	0.72	0.88	0.16	0.18	0.22
	37	36.7	37.9	38.1	8.1	8.9	10.1	0.74	0.78	0.81	0.12	0.16	0.18
<i>Kimono (B)</i>	22	43.7	44.1	45.1	9.5	10.1	10.3	0.96	0.96	0.99	0.17	0.18	0.19
	27	42.3	42.9	43.1	9.1	10.0	10.1	0.95	0.98	0.98	0.17	0.17	0.19
	32	38.8	38.9	39.9	8.5	9.9	10.3	0.82	0.83	0.88	0.14	0.16	0.18
	37	37.8	38.6	38.9	7.5	8.4	9.9	0.77	0.78	0.80	0.12	0.14	0.17
<i>PeopleOnStreet (A)</i>	22	38.6	41.4	43.4	10.2	10.6	11.3	0.95	0.95	0.97	0.19	0.19	0.22
	27	38.3	39.8	41.2	9.5	9.9	10.3	0.91	0.93	0.94	0.17	0.20	0.22
	32	37.0	38.1	40.6	8.9	9.3	10.1	0.88	0.90	0.92	0.18	0.19	0.21
	37	35.4	37.6	38.9	8.1	9.0	9.8	0.78	0.83	0.88	0.15	0.15	0.19

4.5.2 Objective measurements

4.5.2.1 Video quality metrics

Peak Signal to Noise Ratio (PSNR) and the Structural Similarity (SSIM) are used to assess the quality of the encrypted videos. In other words, the quality of the encrypted video reflects the degree of the visual content and, thus, the encryption solution consistency. Results of these two metrics, using original and encrypted ROI schemes are given in Tables 4.4 and 4.5. The average PSNR inside the ROI, for all encrypted sequences, remains below 11.4 dB and the SSIM values are below 0.24. In terms of quality, these results indicate that the quality of video content is very degraded. Moreover, at different bit-rates, video qualities are very poor whatever the used QP . The proposed solutions reduce considerably the visual content quality and, thus, making the known plain-text attack inapplicable.

Table 4.6 presents a brief comparison, in terms of PSNR and SSIM objective metrics, between the proposed encryption method and a state-of-the-art encryption solution examples. The proposed SE solution enables lower PSNR value compared to [157] with less SSIM values than ones given in [157] and [76]. In addition, we performed PSNR and SSIM measures of two different encryption stages: (TC, TC signs, MV, MV signs) and (TC, TC signs, MV, MV signs, IPMs), with Random access and Main Intra configuration. The obtained results indicate the robustness of all encryption stages together, especially the quality degradation that IPMs encryption has impacted on the video sequences (Tables 4.7, 4.8).

Table 4.6 – Comparative evaluation, using weighted PSNR and SSIM for three sequences encoded by HM at ($QP = 32$).

Sequence	Wallendael et al. [157]		Boyadjis et al. [76]		Proposed SE	
	PSNR	SSIM	PSNR	SSIM	PSNR	SSIM
<i>BasketballDrive</i>	11.4	0.40	10.4	0.43	9.9	0.17
<i>KimonoI</i>	10.1	0.32	6.6	0.27	8.9	0.14
<i>VidyoI</i>	12.9	0.61	11.2	0.55	10.1	0.18

Table 4.7 – Mean PSNR (Y) values in dB of the three video classes encoded by HM ($QP = 22$).

Class	Main Intra		Random Access	
	TC, TCs, MV and MVs	All	TC, TCs, MV and MVs	All
<i>B</i>	11.1	10.2	10.4	10.2
<i>D</i>	9.3	8.9	8.7	8.4
<i>E</i>	10.2	9.6	9.7	9.1

Table 4.8 – Mean SSIM values of the three video classes encoded by HM used ($QP = 22$).

Class	Main Intra		Random Access	
	TC, TCs, MV and MVs	All	TC, TCs, MV and MVs	All
<i>B</i>	0.33	0.25	0.30	0.21
<i>D</i>	0.26	0.20	0.23	0.18
<i>E</i>	0.22	0.19	0.20	0.17

4.5.2.2 BD rate evaluation

We consider the Bjøntegaard-Delta Bit-Rate (BD-BR) metric [161], which refer to the average bit-rate differences between two bit-rate-PSNR curves.

The encoding process is performed using `Inter` and `Intra` coding for the 4×4 and 4×3 tile repartitions, with MVs limitations and disabling the in-loop filters across the tile edges. The RD losses with `Intra` and `Inter` coding configurations of the two tiles configurations are provided in Table 4.9 and Table 4.10, respectively. The bit-rate overhead caused by the MVs restriction varies between 2%–18.23% depending on the coding configuration (`Inter` and `Intra`), video content and number of tiles within the frame.

The BD-BR loss for 4×4 tiles repartition in `Inter` coding is more than the loss in `Intra` coding configuration and reaches 12.33% and 5.36%, respectively. The BD-BR loss for 4×3 tiles repartition is less than the loss for 4×4 tiles in both coding configurations. For example, the loss in BD-BR of *KimonoI* (1920×1080) video sequence with 4×3 and 4×4 tiles using `Inter` coding configuration is around 11.65% and 13.19%, respectively. This difference in loss is mainly caused by the mores restrictions related to tile coding, disabling the in-loop filtering across tiles and MVs restriction in the higher number of tiles configuration (4×4). However, in `Intra` coding it remains low and does not exceed 4.13% and 5.16%, respectively. The RD loss for *PeopleOnStreet* (2560×1600) video sequence is 5.13% and 3.42% in `inter` coding and 3.67%, 2.14% in `intra` coding configuration. In general, the proposed encryption solution decreases the RD performances and this is depending slightly of the video sequence content and resolution.

Table 4.11 shows the increase, in terms of BD-BR, introduced by IPMs encryption in two coding configurations: `Main Intra` and `Random Access`. In all `Intra` configuration, where all blocks are `Intra` coded, the encryption decreases the coding efficiency by +2% to +4%. In `Random Access` configuration that uses both `Intra` and `Inter` predictions, the bit-rate increase remains below +2.6%. Therefore, the encryption of the IPMs comes at the expense of slight bit-rate increase, especially in `Inter` coding configurations. Figure 4.4 shows the RD-performance using the average bit-rate difference between two bit-rate-wPSNR (weighted PSNR) curves for *BasketballDrive* video sequence with and without encryption. As depicted the IPMs encryption conduct a diminutive BD-BR loss.

Table 4.9 – BD-rate and complexity increase of the proposed encryption scheme in `Intra` and `Inter` coding (4×4 tile configuration).

Sequence	Intra coding (4×4 tiles)			Inter coding (4×4 tiles)		
	Bit rate loss (%)	Complexity increase (%)		Bit rate loss (%)	Complexity increase (%)	
	BD-rate	Encoding	Decoding	BD-rate	Encoding	Decoding
<i>PeopleOnStreet</i>	3.67	3.05	1.87	5.13	3.27	2.88
<i>Kimono</i>	5.16	3.16	1.21	13.19	3.87	1.96
<i>ParkScene</i>	4.09	2.34	1.13	9.81	3.08	1.89
<i>Cactus</i>	5.43	2.82	2.02	7.65	3.96	2.19
<i>BQTerrace</i>	7.18	2.19	1.67	18.23	3.54	1.93
<i>BasketballDrive</i>	6.34	3.16	2.15	17.11	3.78	2.44
<i>Vidyo1</i>	4.21	2.13	1.32	13.87	2.60	1.91
<i>Vidyo3</i>	6.17	2.31	1.41	10.08	2.98	2.07
<i>Vidyo4</i>	6.01	2.25	1.48	15.91	2.71	1.88
Average	5.36	2.60	1.58	12.33	3.31	2.12

4.5.2.3 Encryption quality

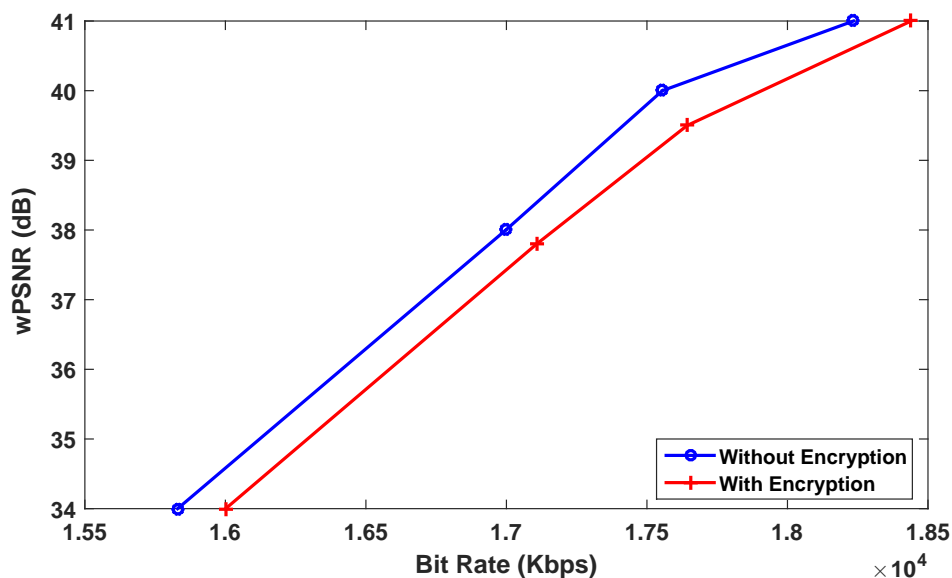
Encryption Quality (EQ) measure is the difference between the frequency of occurrence for each byte with and without encryption. The maximum EQ value is calculated using the following two equations, as

Table 4.10 – BD-rate and complexity increase of the proposed encryption scheme in Intra and Inter coding(4x3 tile configuration).

Sequence	Intra coding (4 × 3 tiles)			Inter coding (4 × 3 tiles)		
	Bit rate loss (%)	Complexity increase (%)		Bit rate loss (%)	Complexity increase (%)	
	BD-rate	Encoding	Decoding	BD-rate	Encoding	Decoding
<i>PeopleOnStreet</i>	2.14	2.11	1.12	3.42	2.16	1.71
<i>Kimono</i>	4.13	2.13	1.01	11.65	2.48	1.63
<i>ParkScene</i>	3.68	1.98	1.06	8.55	2.18	1.12
<i>Cactus</i>	3.14	1.68	1.22	5.12	2.56	1.67
<i>BQTerrace</i>	4.32	1.67	1.10	12.56	2.14	1.73
<i>BasketballDrive</i>	4.74	1.36	1.17	13.49	2.68	1.41
<i>Vidyo1</i>	2.08	1.43	1.15	9.19	1.93	1.43
<i>Vidyo3</i>	4.65	1.21	1.08	7.81	1.68	1.47
<i>Vidyo4</i>	4.79	1.64	1.33	11.02	1.98	1.39
Average	3.74	1.69	1.13	9.20	2.19	1.50

Table 4.11 – Bjontegaard's difference for three video sequences with IPM Encryption.

Schemes	Main Intra	Random Access
<i>Kimono</i>	+4.21%	+2.58%
<i>BasketballDrive</i>	+3.3%	+2.51%
<i>BQSquare</i>	+2.56%	+1.96%

Figure 4.4 – Rate distortion for proposed IPMs encryption for *BasketballDrive* video sequence.

given in [162]:

$$EQ = \frac{\sum_{i=0}^{255} |o_i(P) - o_i(C)|}{256} \quad (4.3)$$

where $o_i(C)$ are the observed occurrence for the byte level i in the encrypted frame C , and $o_i(P)$ are the observed occurrences of the same byte level i in the plain frame P .

$$EQ_{max} = \frac{510 \times L \times C}{256^2} \quad (4.4)$$

where L and C are the height and the width of the gray frame.

The larger the EQ value, the better the encryption security is. The maximum EQ value of a given video frame of *Kimono1*, *PeopleOnStreet* and *Vidyo1* sequences are equal to 16136 31875 and 7171 respectively [1]. Table 4.12 indicates that the EQ values of our proposal with two video sequences (*Kimono1* and *PeopleOnStreet*) are higher compared to results given by [1].

Table 4.12 – The EQ for proposed SE and the state of the art [1] (QP = 22) encoded by HM.

Sequence	EQ in [1]	EQ of proposed SE
<i>Kimono1</i>	8996	10192
<i>PeopleOnStreet</i>	14884	18965
<i>Vidyo1</i>	–	4288

4.5.2.4 Visual analysis

Visual security analysis is used to measure the unidentifiable degree of encrypted videos. Encrypted video is regarded as of high visual security if the distortion of encrypted video is too chaotic to be understood. The Edge Differential Ratio (EDR) that evaluates the edges differences between the original and the encrypted frame has been applied, with Random Access encoding configuration (Intra and Inter predictions) [163], using the Laplacian of Gaussian method [164]. The proposed solution is more efficient when the edges of the encrypted frames are not noticeable. The EDR is calculated as:

$$EDR = \frac{\sum_{i=0}^{h-1} \sum_{j=0}^{w-1} |P_E(i, j) - C_E(i, j)|}{\sum_{i=0}^{h-1} \sum_{j=0}^{w-1} |P_E(i, j) + C_E(i, j)|} \quad (4.5)$$

Where P_E and C_E are the edge detected binary matrix for the plain and cipher frame, respectively. Figures 4.5, 4.6 and 4.7 clarify the visual impact of the proposed scheme on the frame content. Figures 4.5b, 4.6b and 4.7b show the distortion on visual content quality of the frame. Edges in the encrypted frames (Figures 4.5d, 4.6d and 4.7d) are completely affected compared to edges in the original frames (Figures 4.5c, 4.6c and 4.5c).

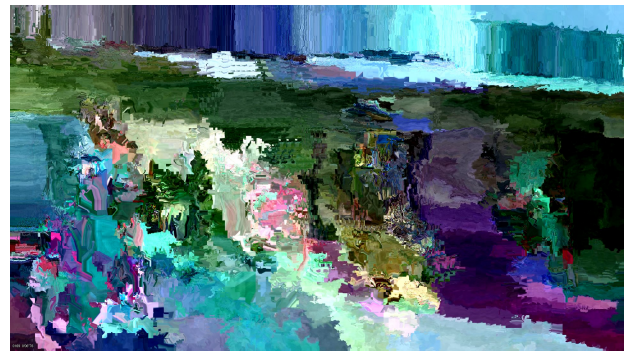
The common step to identify and track the ROI in the video is to split the HEVC frame into tiles where all ROI are included in ROI tiles and the background in separated non ROI tiles [165]. In Figure 4.8 the tiles that including human face represent the ROI tiles and the other tiles represent the background tiles. The proposed encryption solution, performs a selective encryption of ROI tiles at the CABAC bin-string level encrypting only the most sensitive HEVC syntax elements to decrease the visual quality of the ROI as described in Section 4.3 and 4.4. Based on this figure we can observe that, the proposed encryption method conceals the objective quality of the ROI zone while the background remains clean even in inter coding configuration. Videos decoded and decrypted with the correct key on the left side and decoded without decryption (or decryption with incorrect key) on the right side.

4.5.2.5 Histogram analysis

Another objective measurement that widely used in encryption evaluations is the Histogram Analysis (HA). Figure 4.9 and 4.10 depicted an examples of histogram frame with and without encryption, for the *BasketballDrive Kimono* video sequences. Obtained results show that, for the encrypted frame, the



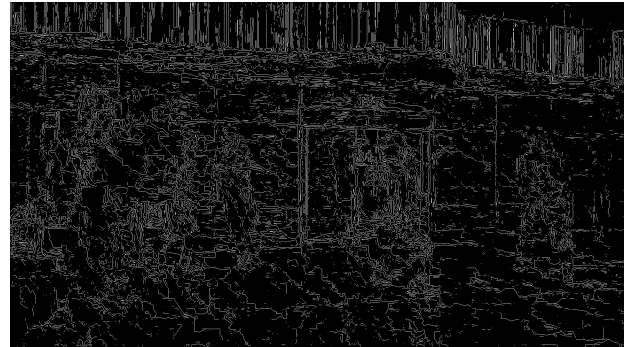
(a) Original frame without encryption



(b) Encrypted frame



(c) EDR of original frame



(d) EDR for encrypted frame

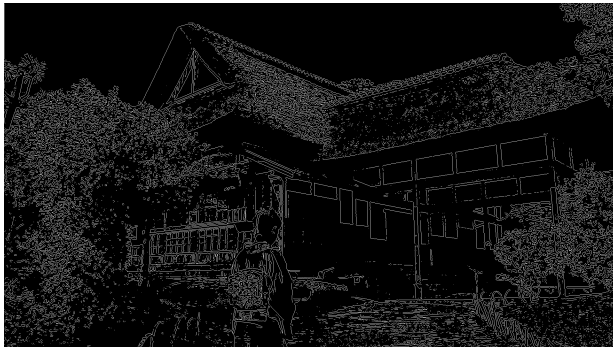
Figure 4.5 – The EDR for frame # 169 in BasketballDrive video sequence.



(a) Original frame without encryption



(b) Encrypted frame



(c) EDR of original frame



(d) EDR for encrypted frame

Figure 4.6 – The EDR for frame # 184 in Kimono video sequence.

histogram is close to the pseudo-random distribution (uniformity) and completely different to that for the

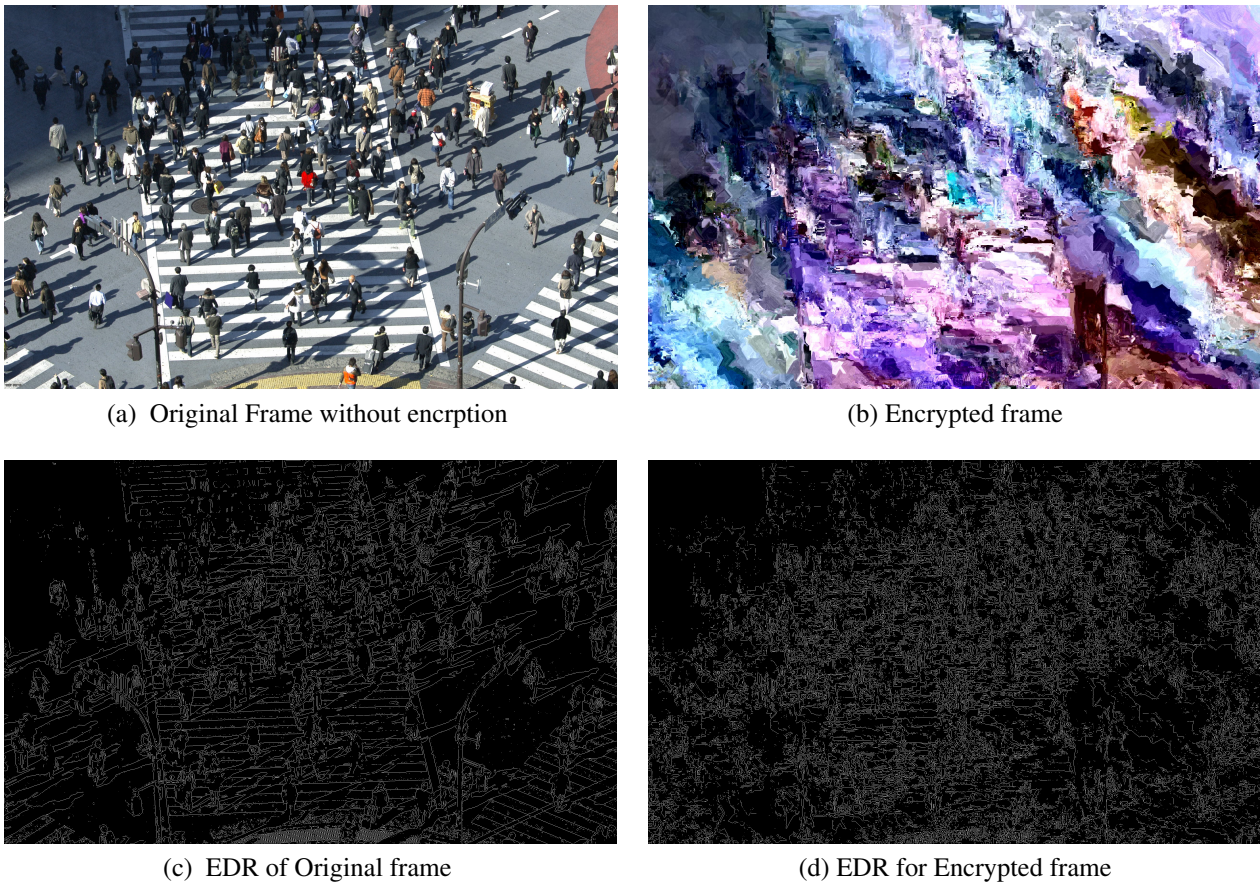


Figure 4.7 – The EDR for frame # 37 in PeopleOnstreet video sequence .

original frame. These results, together with previous objective results, lead to the robustness of the proposed solution against visual attacks.

4.5.2.6 NIST test

To evaluate the statistical performance of the Keystream produced, we also use one of the most popular standards for investigating the randomness of binary data, namely the (NIST) statistical test [145] that detailed in chapter 3. This test is a statistical package that consists of 188 tests that are proposed to assess the randomness of arbitrarily long binary sequences. In Table 4.13, we give the P_value and the proportion for 15 NIST tests. These results indicate the strength of the generated keystream.

4.5.2.7 Security attacks

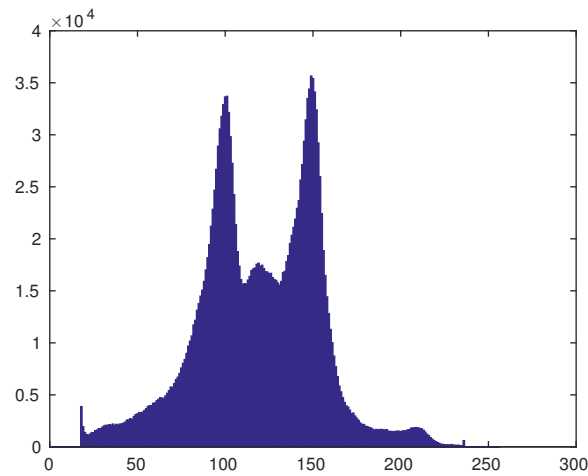
In this section we provide analysis of resistance of our proposed encryption solutions against some known attacks. Mainly Brute force attack, Key sensitivity attack, known-plaintext attack and chosen-plaintext attack.

Brute force attack

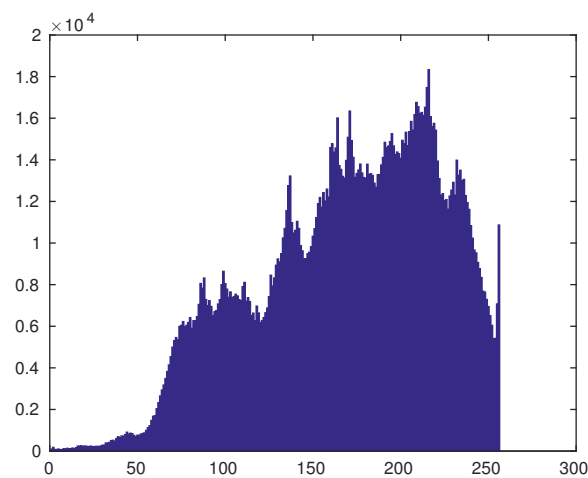
It breaks the cryptosystem by trying a large number of possible keys until the correct one is found. In the worst case, all possible keys in key space are tested [166]. In addition, The size of the secret key, formed by one recursive cell delay is 222 bits. Thus the brute-force attack will be inapplicable. Encryption of motion vector differences and residual signs was classified by [167] to be secure selective encryption



Figure 4.8 – Frame #9 of HEVC videos encrypted with the proposed ROI encryption: (a) Correctly decrypted videos. (b) Encrypted videos.



(a) Histogram of original frame



(b) Histogram encrypted frame

Figure 4.9 – Histogram for frame # 300 BasketballDrive video sequence.

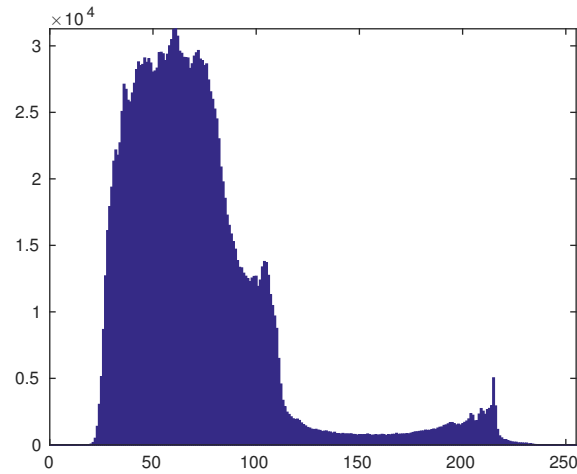
algorithms. In our proposed encryption solution extra parameters such as the MVs, TCs, IPMs together increase the complexity of brute force attack.

Key sensitivity attack

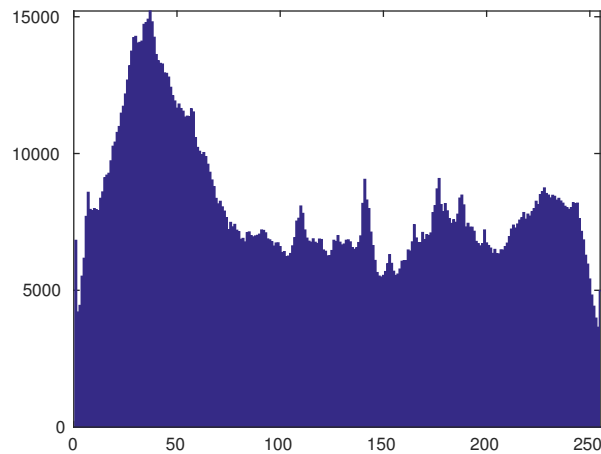
A key stream must be sensitive to one bit change in the secret key used. This property is important in order to resist many attacks [24]. To test the key sensitivity of the key stream cipher, we applied selective encryption on BasketballDrive, Kimono, Four people and BQSquare video sequences two times with a secret key that differ only by the LSB bit. Then we computed the following parameters: the Number of Pixel Change Rate (NPCR), the Unified Average Changing Intensity (UACI) and the Hamming Distance (HD). The parameters (NPCR, UACI) are necessary but not sufficient to guarantee that the key stream cipher is resistant against the key sensitivity attack. For this cause, Hamming Distance measurement has been added [153].

The optimal NPCR, UACI values is 99.61%, 33.46% respectively [154]. The size of the secret key, formed by one recursive cell delay is 222 bits. Thus the brute-force attack will be inapplicable.

The optimal HD value is 50% (Avalanche criterion) [155]. Table 4.14 indicates that the NPCR, UACI and



(a) Histogram of original frame



(b) Histogram encrypted frame

Figure 4.10 – Histogram for frame # 8 Kimono video sequence.

HD values are close to optimal values.

Known-plaintext attack and Chosen-plaintext attack

We did the same experiment as done in section 3.5.2.1 on encrypt-able bits of frames.

4.5.3 Subjective evaluations

The subjective evaluations were carried out, in IETR laboratory psychovisual room complying with the ITU-R BT.500-13 Recommendation [168]. A display screen Full HD 32 inches Samsung UN32J5003 was used to visualise the video sequences. Fifteen observers, 10 men and 5 women their age between 20 to 40 years, have participated in this experiment. All the subjects were screened for color blindness and visual acuity using Ishihara and Snellen charts, respectively, and have a visual acuity of 10/10 in both eyes with or without correction, as detailed in [81]. We consider five video sequences from Table 4.3 (*FourPeople*, *Kimono*, *BasketballDrive*, *BQSquare*, *Cactus*). Both the encoding and selective encryption of two encryp-

Table 4.13 – NIST test of chaos based Key stream sequences.

Test	P_value	Proportion %
<i>Frequency test</i>	0.851	100.000
<i>Block-frequency test</i>	0.172	99.000
<i>Cumulative-sums test</i>	0.382	99.500
<i>Runs test</i>	0.679	99.000
<i>Longest-run test</i>	0.883	97.000
<i>Rank test</i>	0.367	100.000
<i>FFT test</i>	0.367	100.000
<i>Nonperiodic-templates</i>	0.482	98.905
<i>Overlapping-templates</i>	0.964	100.000
<i>Universal</i>	0.437	98.000
<i>Approximty entropie</i>	0.679	98.000
<i>Random-excursions</i>	0.336	99.632
<i>Random-excursions-variant</i>	0.339	99.918
<i>Serial test</i>	0.557	98.500
<i>Linear-complexity</i>	0.475	99.000

Table 4.14 – The NPCR, UACI and HD

Sequence	NPCR	UACI	HD
<i>Kimono</i>	97.9	32.4	0.4999
<i>BasketballDrive</i>	97.1	32.7	0.4997
<i>Four people</i>	97.2	32.9	0.4997
<i>BQSquare</i>	98.2	32.8	0.4999

tion schemes: (TC, TCs, MV, MVs) and All (TC,TCs, MV, MVs, IPMs) with Random Access encoding, performed by the HM(16.7) encoder. Finally, these coding configurations, results in 40 encrypted video sequences, with different QP and resolutions.

4.5.3.1 Design and procedure

In our subjective quality experiment, we used the Double Stimulus Continuous Quality Scale (DSCQS) method [168]. Each encrypted video was presented twice to observer along with its original version. Participants were asked to judge on the degree of content visibility of the encrypted videos numerically. Thus, each participant must assign a visibility score to each of the 40 test videos, according to a rating scale, Varies from 1: video content is *Completely Invisible* to 5:video content is *Clearly Visible* as shown in Table 4.15. At the end of each test condition, a dedicated Graphical User Interface (GUI) is displayed on the screen for about 10 seconds during which the observer gives and then confirms its judgement. video sequences were jumble in such a way that two consecutive sequences must be from different categories, configuration and quality levels, this will eliminate participants memory effects.

4.5.3.2 Data processing

The first step in the results analysis is to calculate the average score of Mean Opinion Score (MOS) for each video used in the experience. This average is given by Equation (4.6).

Table 4.15 – Ranking scale used in our subjective evaluation experiment.

Visibility Degree	Score
<i>Clearly Visible</i>	5
<i>Visible</i>	4
<i>Slightly Visible</i>	3
<i>Barely Visible</i>	2
<i>Completely Invisible</i>	1

$$MOS_{jk} = \frac{1}{N} \sum_{i=1}^N s_{ijk} \quad (4.6)$$

where s_{ijk} is the score of participant i for degree of visibility j of the sequence k and N is the number of observers.

In order to better evaluate the reliability of the obtained results, it is advisable to associate for each MOS score a confidence interval, usually at 95%. This is given by Equation (4.7). Scores respecting the experiment conditions must be contained in the interval $[MOS_{jk} - IC_{jk}, MOS_{jk} + IC_{jk}]$.

$$IC_{jk} = 1.95 \frac{\delta_{jk}}{\sqrt{N}}, \quad \delta_{jk} = \sqrt{\sum_{i=1}^N \frac{(s_{ijk} - MOS_{jk})^2}{N}} \quad (4.7)$$

4.5.3.3 Subjective scores

The subjective results scores of all participants, collected through the dedicated GUI, have been used for the perceptual encryption measurement. Subjects scores range generally between (*barely visible*) and (*completely invisible*) for first encryption scheme. This implies that the human visibility is very significantly reduced by using the proposed SE solution. Indeed, the results imply that the video content is invisible. Otherwise, subjects can only try to guess the context type without seeing any detail of the shown video. A slight MOS variation, depending on the video content and the used QP , can be noticed. In fact, the main subjects scores directed to '*Completely Invisible*' when we added the IPM encryption to the first encryption scheme. The subjects can hardly see a little things of the video (without being able to know the global context of the presented video). Results depend strongly of the video classes and video contents. *BQSquare* (Classe D) and *Cactus* are completely invisible by the whole subjects, with $MOS \simeq 1$, and very few variations depending on the used QP . In addition, *BasketballDrive* shows low visibility scores due to its strong movement character. Curves of this video are dramatically decreased when encrypting the IPMs (Figure 4.11).

4.5.3.4 ANOVA statistical test

A statistical study was performed using the Analyse of Variance (ANOVA) [169]. Indeed, ANOVA allows studying whether the variation in visibility scores is a result of the intended variation of experimental variables (i.e. QP , Class, Encrypted Scheme and Content), or simply due to chance. Table 4.16 indicates that only 'Encryption Scheme' parameter has a significant influence on the subjects scores with $p\text{-value} < 0.0001$ ¹.

1. a factor is considered influencing if $p\text{-value} < 0.05$

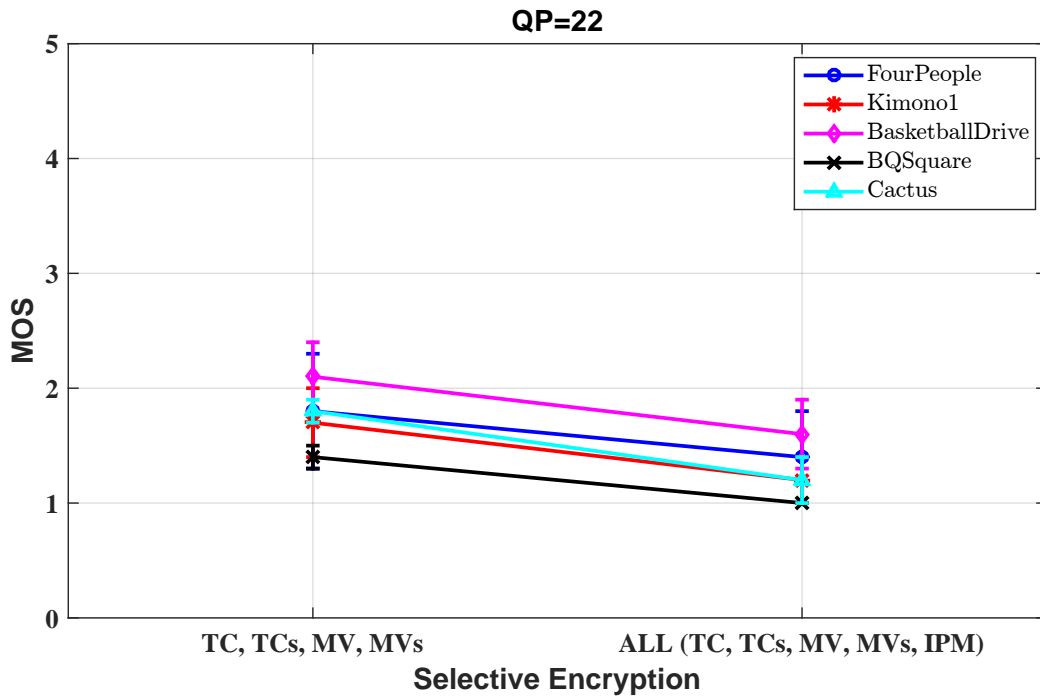


Figure 4.11 – Subjects visibility scores including 95% confidence intervals for (QP = 22).

Table 4.16 – ANOVA on the whole dataset, Df: number of degree-of-freedom and F-value: Fisher test

Source	DF	F-value	P-value
<i>Class</i>	2	1.0133	0.4312
<i>Content</i>	4	0.9981	0.5631
<i>QP</i>	3	0.1381	0.133
<i>SE Scheme</i>	1	97.855	< 0.0001

4.5.4 Complexity evaluations

The small computational overhead is crucial especially for embedded/mobile devices having restricted processing power. It can be calculated through analysis of the additional time required for encoding and decoding encrypted bitstreams. The characteristics of the computer used for this simulation are described in section 4.5. The encoding and decoding complexities of 4 x 4 and 4 x 3 tile configurations are reported over the anchor configuration in Table 4.9 and Table 4.10, respectively.

With considered video sequences, the additional time for encoding and decoding is negligible. For 4 x 4 tile configuration, the encoding time increases by 2.6% in Intra coding and 3.3% in Inter coding. The respective decoding times are 1.6% and 2.1% higher. Changing the tile configuration to 4 x 3 narrows complexity overhead further so that the respective complexity increases are 2.2% and 1.6% for encoding and 1.5% and 1.1% for decoding. This low complexity overhead at the encoder/decoder sides is mainly introduced by the encryption/decryption processes as well as the specific processing and bitrate increase related to the tiling repartitions.

These results confirm that the proposed selective encryption model can be performed without noticeable performance compromises.

4.6 Conclusion

This chapter proposed a new selective encryption solution to protect privacy in the HEVC video content. A robust scheme to encrypts the ROI in the video is proposed. The selective encryption is based on chaos-based generator. The ROI is extracted through independent HEVC tile concept. The encryption is performed at the CABAC bin-string level so that the encrypted bit-stream is decodable with a standard HEVC decoder and a privacy key is only needed in decryption. However, some bit rate overhead is introduced in the HEVC encoding process in order to prevent the propagation of the encryption outside the ROI. Selective encryption of luma and chroma IPMs is proposed in addition to the MV, TC syntax elements encryption in order to significantly improve the structural deterioration of the video content. The proposed end-to-end encryption/decryption is integrated into three open-source software projects: HEVC Kvazaar, HM encoders and OpenHEVC decoder. Subjective evaluation and experimental tests showed that the proposed solution performs a secure protection of privacy in the HEVC video content with a small overhead in bit rate and coding complexity. It also prevents unexpected behaviour of the decoder.

IV

Conclusions and Future work Perspectives

Conclusions and Future work Perspectives

5.1 Conclusions and future work

In this thesis we designed and implemented chaos-based crypto and crypto-compression systems for protecting image and video contents. For this purpose, chaos-based pseudo-random number generators have been designed, implemented and analyzed. Based on these chaotic generators, two applications were proposed. The first one consisting of realizing a random numbers generator based on a pseudo chaotic numbers generator. The second application relies on the realization of a chaos based stream cipher used for the multimedia contents protecting. Then, a selective video encryption system has been proposed to secure video bitstream in the HEVC standard. In what follows we briefly summarize the state-of-the-arts given in Chapters 1 and 2. Then we highlight our contributions established in Chapters 3 and 4.

Chapter 1 and 2 introduced the thesis state-of-the-art around the classical and chaos-based cryptography, a review of the parallel programming tools and the description of some static and dynamic tools used to validate the security of the codes' implementation.

In Chapter 3, we designed and implemented in an efficient and secure manner pseudo chaotic number generators (PCNGs). These PCNGs are based on a modular structure containing an IV-setup, a Key-setup, a non-volatile memory, an output function and a strong cryptographic internal state with internal feedback mode. Based on the previous PCNGs, two applications were designed, implemented and analysed. The former application deals with the realization of a random number generator (RNG) based to the PCNG. The entropy source of the RNG comes from Linux RNG `/dev/urandom`, and the obtained results are promising. The latter application concerns the realization of a chaos-based stream cipher. The proposed chaotic stream encryption system is very robust against known cryptographic and statistical attacks. Indeed, the results obtained from both the cryptographic analysis and of the common statistical tests demonstrated the robustness of the proposed chaotic system. This is also due to its strong non-linearity compared to the other classical stream cipher systems. The computation performance for the proposed chaos-based stream cipher is comparable to classical stream ciphers and is even better when a very big data size is treated.

In Chapter 4, we have proposed a new selective encryption solution to protect privacy in the HEVC video content. A robust scheme to encrypt the ROI in the video has been presented. The selective encryption is based on the chaos-based generator realized in Chapter 3. The ROI is extracted through independent HEVC tile concept. The encryption is done at the CABAC bin-string level so that the encrypted bit-stream is decodable with a standard HEVC decoder and the secret key is only needed in decryption. However, some bit rate overhead is introduced in the HEVC encoding process in order to prevent the propagation of the encryption outside the ROI. Selective encryption of luma and chroma IPMs is proposed in addition to

the MV, and TC syntax elements. This allows to significantly improve the structural retro-gradations of the video content. The proposed end-to-end encryption/decryption is implemented into three open-source software projects: HEVC Kvazaar, HM encoders and OpenHEVC decoder. Subjective evaluation and experimental tests showed that the proposed solution ensures a protection of privacy in the HEVC video content with a diminutive overhead in bit rate and coding complexity.

In future work we will address the following issues:

1. A parallel implementation of our chaotic system using MPI.
2. A parallel encryption system using parallel chaos-based generator with parallel multi-threading tool in HEVC.
3. A selective encryption of IPMs HEVC syntax element using a lookup table to save the number of coded coefficients to deduce the scanning mode.

Bibliography

- [1] M. Farajallah, “Chaos based crypto and joint crypto-compression systems for images and videos,” Ph.D. dissertation, University of Nantes, 2015. [8](#), [25](#), [26](#), [115](#)
- [2] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand, “Overview of the High Efficiency Video Coding (HEVC) Standard,” *IEEE Transactions on circuits and systems for video technology*, vol. 22, no. 12, pp. 1649–1668, 2012. [9](#), [19](#), [34](#), [35](#), [36](#), [37](#), [39](#), [105](#)
- [3] Y.-C. Tseng, Y.-Y. Chen, and H.-K. Pan, “A secure data hiding scheme for binary images,” *IEEE transactions on communications*, vol. 50, no. 8, pp. 1227–1231, 2002. [17](#)
- [4] K. M. M. de Leeuw and J. Bergstra, *The history of information security: a comprehensive handbook*. Elsevier, 2007. [17](#)
- [5] S. Williams, “Cryptography and network security: Principles and practices,” ed: *Pearson Education*, 2006. [17](#)
- [6] E. Biham and A. Shamir, “Differential cryptanalysis of des-like cryptosystems,” *Journal of CRYPTOLOGY*, vol. 4, no. 1, pp. 3–72, 1991. [18](#), [97](#)
- [7] G. Chen, Y. Mao, and C. K. Chui, “A symmetric image encryption scheme based on 3d chaotic cat maps,” *Chaos, Solitons & Fractals*, vol. 21, no. 3, pp. 749–761, 2004. [18](#), [25](#)
- [8] D. SHARMA, A. KULSHRESHTHA, and S. RAM, “Network security challenges and cryptography in network security,” *Journal of Computer and Mathematical Sciences Vol*, vol. 2, no. 1, pp. 1–169, 2011. [18](#)
- [9] H. Kopetz, *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011. [18](#)
- [10] P. Koopman, “Embedded system security,” *Computer*, vol. 37, no. 7, pp. 95–97, 2004. [18](#)
- [11] C. E. Shannon, “Communication theory of secrecy systems*,” *Bell system technical journal*, vol. 28, no. 4, pp. 656–715, 1949. [18](#), [25](#)
- [12] S. Babar, P. Mahalle, A. Stango, N. Prasad, and R. Prasad, “Proposed security model and threat taxonomy for the internet of things (iot),” in *International Conference on Network Security and Applications*. Springer, 2010, pp. 420–429. [19](#)
- [13] S. El Assad and M. Farajallah, “A new chaos-based image encryption system,” *Signal Processing: Image Communication*, vol. 41, pp. 144–157, 2016. [19](#), [67](#)
- [14] Y. Zhou, L. Bao, and C. P. Chen, “Image encryption using a new parametric switching chaotic system,” *Signal processing*, vol. 93, no. 11, pp. 3039–3052, 2013. [23](#)
- [15] L. Kocarev, P. Amato, and G. Rizzotto, “Method of generating a chaos-based pseudo-random sequence and a hardware generator of chaos-based pseudo random bit sequences,” Aug. 17 2010, uS Patent 7,779,060. [23](#)
- [16] G. Alvarez and S. Li, “Some basic cryptographic requirements for chaos-based cryptosystems,” *International Journal of Bifurcation and Chaos*, vol. 16, no. 08, pp. 2129–2151, 2006. [23](#)
- [17] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, “Cryptanalytic attacks on pseudorandom number generators,” in *Fast Software Encryption*. Springer, 1998, pp. 168–188. [23](#)

- [18] M. E. Hellman, "An overview of public key cryptography," *IEEE Communications Magazine*, vol. 40, no. 5, pp. 42–49, 2002. [23](#)
- [19] Z. Shahid, M. Chaumont, and W. Puech, "Fast Protection of H. 264/AVC by Selective Encryption of CAVLC and CABAC for I and P Frames," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 21, no. 5, pp. 565–576, 2011. [24](#), [34](#), [40](#), [105](#)
- [20] F. Dufaux and T. Ebrahimi, "Scrambling for Privacy Protection in Video Surveillance Systems," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 18, no. 8, pp. 1168–1174, 2008. [24](#), [34](#), [41](#), [105](#)
- [21] E. M. Newton, L. Sweeney, and B. Malin, "Preserving Privacy by De-identifying Face Images," *IEEE transactions on Knowledge and Data Engineering*, vol. 17, no. 2, pp. 232–243, 2005. [24](#), [34](#), [41](#), [105](#)
- [22] L. Tang, "Methods for Encrypting and Decrypting MPEG Video Data Efficiently," in *Proceedings of the fourth ACM international conference on Multimedia*. ACM, 1997, pp. 219–229. [24](#), [41](#)
- [23] K. Desnos, S. El Assad, A. Arlicot, M. Pelcat, and D. Menard, "Efficient multicore implementation of an advanced generator of discrete chaotic sequences," in *Chaos-Information Hiding and Security (CIHS), International Workshop on*, 2014. [24](#), [25](#), [71](#), [87](#)
- [24] S. Lian, J. Sun, and Z. Wang, "Security analysis of a chaos-based image encryption algorithm," *Physica A: Statistical Mechanics and its Applications*, vol. 351, no. 2, pp. 645–661, 2005. [24](#), [97](#), [119](#)
- [25] S. El Assad, H. Noura, and I. Taralova, "Design and analyses of efficient chaotic generators for crypto-systems," in *World Congress on Engineering and Computer Science 2008, WCECS'08. Advances in Electrical and Electronics Engineering-IAENG Special Edition of the*. IEEE, 2008, pp. 3–12. [24](#)
- [26] S. El Assad and H. Noura, "Generator of chaotic sequences and corresponding generating system," WO2011121218 A1 Extension to : Europe EP-2553567 A1, February 2013 ; China : CN-103124955 A, May 2013 ; Japan : JP-2013524271 A, June 2013 ; United states : US-20130170641, July 2013., 10 2011. [24](#), [25](#), [68](#)
- [27] M. Hasler and Y. L. Maistrenko, "An introduction to the synchronization of chaotic systems: Coupled skew tent maps," *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, vol. 44, no. 10, pp. 856–866, 1997. [24](#)
- [28] R. Lozi, "Emergence of randomness from chaos," *International Journal of Bifurcation and Chaos*, vol. 22, no. 02, pp. 1 250 021–1 –1 250 021–15, 2012. [24](#), [82](#)
- [29] S. Li, G. Chen, and X. Mou, "On the dynamical degradation of digital piecewise linear chaotic maps," *International Journal of Bifurcation and Chaos*, vol. 15, no. 10, pp. 3119–3151, 2005. [24](#)
- [30] S. Li, Q. Li, W. Li, X. Mou, and Y. Cai, "Statistical properties of digital piecewise linear chaotic maps and their roles in cryptography and pseudo-random coding," in *Cryptography and Coding*. Springer, 2001, pp. 205–221. [24](#)
- [31] S. Li, X. Mou, and Y.-L. Cai, "Pseudo-random bit generator based on couple chaotic systems and its application in stream-ciphers cryptography," in *Progress in Cryptology–INDOCRYPT 2001: Second International Conference on Cryptology in India Chennai, India, December 16 C20, 2001 Proceedings*, 2001, pp. 316–329. [24](#)
- [32] O. Kurganskyy, I. Potapov, and F. Sancho-Caparrini, "Reachability problems in low-dimensional iterative maps," *International Journal of Foundations of Computer Science*, vol. 19, no. 04, pp. 935–951, 2008. [24](#)
- [33] T. E. Tkacik, "A hardware random number generator," in *Cryptographic Hardware and Embedded Systems-CHES 2002*. Springer, 2003, pp. 450–453. [24](#)
- [34] J. Fridrich, "Symmetric ciphers based on two-dimensional chaotic maps," *International journal of bifurcation and chaos*, vol. 8, no. 06, pp. 1259–1284, 1998. [24](#), [25](#)

- [35] R. Lozi, “Giga-periodic orbits for weakly coupled tent and logistic discretized maps,” *Proc. Conf. Intern. On Industrial and Appl. Math., New Delhi, India, Invited conference*, pp. 1–45, 2007. 24
- [36] A. Senouci, I. Benkhaddra, A. Boukabou, A. Bouridane, and A. Ouslimani, “Implementation and evaluation of a new unified hyperchaos-based prng,” in *Microelectronics (ICM), 2014 26th International Conference on*. IEEE, 2014, pp. 1–4. 24
- [37] S. Li, X. Mou, and Y. Cai, “Improving security of a chaotic encryption approach,” *Physics Letters A*, vol. 290, no. 3, pp. 127–133, 2001. 25
- [38] H. Yang, K.-W. Wong, X. Liao, W. Zhang, and P. Wei, “A fast image encryption and authentication scheme based on chaotic maps,” *Communications in Nonlinear Science and Numerical Simulation*, vol. 15, no. 11, pp. 3507–3517, 2010. 25, 26
- [39] M. Farajallah, S. El Assad, and O. Deforges, “Fast and secure chaos-based cryptosystem for images,” *International Journal of Bifurcation and Chaos*, vol. 26, no. 2, pp. 1 650 021–1–1 650 021–21, 2016. 25, 26, 67
- [40] Y. Wang, K.-W. Wong, X. Liao, and G. Chen, “A new chaos-based fast image encryption algorithm,” *Applied soft computing*, vol. 11, no. 1, pp. 514–522, 2011. 25, 26
- [41] W. Zhang, K.-w. Wong, H. Yu, and Z.-l. Zhu, “An image encryption scheme using reverse 2-dimensional chaotic map and dependent diffusion,” *Communications in Nonlinear Science and Numerical Simulation*, vol. 18, no. 8, pp. 2066–2080, 2013. 25, 26
- [42] S. El Assad, M. Farajallah, and C. Vladeanu, “Chaos-based block ciphers: An overview,” in *Communications (COMM), 2014 10th International Conference on*. IEEE, 2014, pp. 1–4. 25
- [43] W. Stallings, *Cryptography and network security: principles and practices*. Pearson Education India, 2006. 27, 87, 92
- [44] H. Lipmaa, D. Wagner, and P. Rogaway, “Comments to nist concerning aes modes of operation: Ctr-mode encryption,” vol. 1, pp. 1– 4, 2000. 27
- [45] M. Boesgaard, M. Vesterager, T. Christensen, and E. Zenner, “The stream cipher rabbit,” *ECRYPT Stream Cipher Project Report*, vol. 6, 2005. 28
- [46] eSTREAM, *eSTREAM: the ECRYPT Stream Cipher Project*, 2008. [Online]. Available: <http://www.ecrypt.eu.org/stream/> 28, 67
- [47] A. Kircanski and A. M. Youssef, “Differential fault analysis of rabbit,” in *International Workshop on Selected Areas in Cryptography*. Springer, 2009, pp. 197–214. 28
- [48] D. J. Bernstein, “The salsa20 family of stream ciphers,” in *New stream cipher designs*. Springer, 2008, pp. 84–97. 28
- [49] Y. Tsunoo, T. Saito, H. Kubo, T. Suzaki, and H. Nakashima, “Differential cryptanalysis of salsa20/8,” in *Workshop Record of SASC*, 2007. 28
- [50] H. Wu, “The stream cipher hc-128,” in *New stream cipher designs*. Springer, 2008, pp. 39–47. 28
- [51] ———, “A new stream cipher hc-256,” in *International Workshop on Fast Software Encryption*. Springer, 2004, pp. 226–244. 28
- [52] A. Kircanski and A. M. Youssef, “Differential fault analysis of hc-128,” in *International Conference on Cryptology in Africa*. Springer, 2010, pp. 261–278. 28
- [53] C. Berbain, O. Billet, A. Canteaut, N. Courtois, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier *et al.*, “Sosemanuk, a fast software-oriented stream cipher,” in *New stream cipher designs*. Springer, 2008, pp. 98–118. 28
- [54] Y. E. Salehani, A. Kircanski, and A. Youssef, “Differential fault analysis of sosemanuk,” in *International Conference on Cryptology in Africa*. Springer, 2011, pp. 316–331. 28

- [55] N. Abderrahim, F. Benmansour, and O. Seddiki, “A chaotic stream cipher based on symbolic dynamic description and synchronization,” *Nonlinear Dynamics*, vol. 78, no. 1, pp. 197–207, 2014. [29](#), [92](#)
- [56] H. Lü, S. Wang, X. Li, G. Tang, J. Kuang, W. Ye, and G. Hu, “A new spatiotemporally chaotic cryptosystem and its security and performance analyses,” *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 14, no. 3, pp. 617–629, 2004. [29](#), [92](#)
- [57] P. Li, Z. Li, W. A. Halang, and G. Chen, “A stream cipher based on a spatiotemporal chaotic system,” *Chaos, Solitons & Fractals*, vol. 32, no. 5, pp. 1867–1876, 2007. [29](#)
- [58] X. Ge, F. Liu, B. Lu, and W. Wang, “Cryptanalysis of a spatiotemporal chaotic image/video cryptosystem and its improved version,” *Physics Letters A*, vol. 375, no. 5, pp. 908–913, 2011. [29](#)
- [59] S. Liu, J. Sun, Z. Xu, and Z. Cai, “An improved chaos-based stream cipher algorithm and its vlsi implementation,” in *Networked Computing and Advanced Information Management, 2008. NCM’08. Fourth International Conference on*, vol. 2. IEEE, 2008, pp. 191–197. [29](#)
- [60] C. Fu and Z.-l. Zhu, “An improved chaos-based stream cipher algorithm,” in *Natural Computation, 2007. ICNC 2007. Third International Conference on*, vol. 3. IEEE, 2007, pp. 179–183. [29](#)
- [61] B. Zhang and C. Jin, “Cryptanalysis of a chaos-based stream cipher,” in *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*. IEEE, 2008, pp. 2782–2785. [29](#)
- [62] R. Yin, J. Yuan, Q. Yang, X. Shan, and X. Wang, “Discretization of coupled map lattices for a stream cipher,” *Tsinghua Science & Technology*, vol. 16, no. 3, pp. 241–246, 2011. [29](#)
- [63] O. Jallouli, M. Abutaha, S. El Assad, M. Chetto, A. Queudet, and O. Deforges, “Comparative study of two pseudo chaotic number generators for securing the iot,” in *Advances in Computing, Communications and Informatics (ICACCI), 2016 International Conference on*. IEEE, 2016, pp. 1340–1344. [29](#)
- [64] D. Zhu, R. Melhem, and D. Mossé, “The effects of energy management on reliability in real-time embedded systems,” in *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*. IEEE, 2004, pp. 35–40. [32](#)
- [65] R. Jejurikar, C. Pereira, and R. Gupta, “Leakage aware dynamic voltage scaling for real-time embedded systems,” in *Proceedings of the 41st annual Design Automation Conference*. ACM, 2004, pp. 275–280. [32](#)
- [66] K. A. Prather, T. Nordmeyer, and K. Salt, “Real-time characterization of individual aerosol particles using time-of-flight mass spectrometry,” *Analytical Chemistry*, vol. 66, no. 9, pp. 1403–1407, 1994. [32](#)
- [67] R. I. Davis and A. Burns, “A survey of hard real-time scheduling for multiprocessor systems,” *ACM computing surveys (CSUR)*, vol. 43, no. 4, p. 35, 2011. [34](#)
- [68] K. G. Shin and P. Ramanathan, “Real-time computing: A new discipline of computer science and engineering,” *Proceedings of the IEEE*, vol. 82, no. 1, pp. 6–24, 1994. [34](#)
- [69] R. I. Davis, A. Zazos, and A. Burns, “Efficient exact schedulability tests for fixed priority real-time systems,” *IEEE Transactions on Computers*, vol. 57, no. 9, pp. 1261–1276, 2008. [34](#)
- [70] P. Pillai and K. G. Shin, “Real-time dynamic voltage scaling for low-power embedded operating systems,” in *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 89–102. [34](#)
- [71] K. Englehart and B. Hudgins, “A robust, real-time control scheme for multifunction myoelectric control,” *IEEE transactions on biomedical engineering*, vol. 50, no. 7, pp. 848–854, 2003. [34](#)
- [72] ITU-T and ISO/IEC, “High Efficiency Video Coding, Document, ITU-T Rec. H.265 and ISO/IEC 23008-2 (HEVC),” 2013. [34](#)
- [73] J.-R. Ohm, G. J. Sullivan, H. Schwarz, T. K. Tan, and T. Wiegand, “Comparison of the Coding Efficiency of Video Coding Standards— Including High Efficiency Video Coding (HEVC),” *IEEE Transactions on circuits and systems for video technology*, vol. 22, no. 12, pp. 1669–1684, 2012. [34](#)

- [74] I. ITU-T and I. JTC, “1: Advanced Video Coding for Generic Audiovisual Services, ITU-T Rec,” *H*, vol. 264, pp. 14 496–10, 2009. [34](#)
- [75] W. Hamidouche, M. Farajallah, M. Raullet, O. Deforges, and S. El Assad, “Selective video Encryption using Chaotic System in the SHVC Extension,” in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2015, pp. 1762–1766. [34](#), [40](#), [105](#), [108](#)
- [76] B. Boyadjis, C. Bergeron, B. Pesquet-Popescu, and F. Dufaux, “Extended Selective Encryption of H. 264/AVC (CABAC) and HEVC Encoded Video Streams,” *IEEE Transactions on circuits and systems for video technology*. [34](#), [40](#), [105](#), [112](#)
- [77] F. Peng, X.-w. Zhu, and M. Long, “An ROI Privacy Protection Scheme for H. 264 Video based on FMO and Chaos,” *IEEE transactions on information forensics and security*, vol. 8, no. 10, pp. 1688–1699, 2013. [34](#), [41](#), [105](#)
- [78] H. Sohn, E. T. AnzaKu, W. De Neve, Y. M. Ro, and K. N. Plataniotis, “Privacy Protection in Video Surveillance Systems using Scalable Video Coding,” in *Advanced Video and Signal Based Surveillance, 2009. AVSS’09. Sixth IEEE International Conference on*. IEEE, 2009, pp. 424–429. [34](#), [41](#), [105](#)
- [79] K. Misra, A. Segall, M. Horowitz, S. Xu, A. Fuldseth, and M. Zhou, “An Overview of Tiles in HEVC,” *IEEE Journal of Selected Topics in Signal Processing*, vol. 7, no. 6, pp. 969–977, 2013. [34](#), [105](#)
- [80] M. Farajallah, W. Hamidouche, O. Déforges, and S. El Assad, “ROI Encryption for the HEVC Coded video Contents,” in *Image Processing (ICIP), 2015 IEEE International Conference on*. IEEE, 2015, pp. 3096–3100. [34](#), [105](#)
- [81] N. Sidaty, W. Hamidouche, and O. Deforges, “A New Perceptual Assessment Methodology for selective HEVC Video Encryption,” in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2017. [34](#), [105](#), [120](#)
- [82] J. Lainema, F. Bossen, W.-J. Han, J. Min, and K. Ugur, “Intra Coding of the HEVC Standard,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1792–1801, 2012. [34](#)
- [83] F. Bossen, B. Bross, K. Suhring, and D. Flynn, “HEVC Complexity and Implementation Analysis,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1685–1696, 2012. [35](#), [110](#)
- [84] V. Sze, M. Budagavi, and G. J. Sullivan, “High Efficiency Video Coding (HEVC),” in *Integrated Circuit and Systems, Algorithms and Architectures*. Springer, 2014. [39](#), [40](#)
- [85] C. C. Chi, M. Alvarez-Mesa, B. Juurlink, G. Clare, F. Henry, S. Pateux, and T. Schierl, “Parallel Scalability and Efficiency of HEVC Parallelization Approaches,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1827–1838, 2012. [40](#)
- [86] E.-k. Ryu, J.-h. Nam, S.-o. Lee, H.-h. Jo, and D.-g. Sim, “Sample Adaptive Offset Parallelism in HEVC,” in *Multimedia and Ubiquitous Engineering*. Springer, 2013, pp. 1113–1119. [40](#)
- [87] D. Marpe, T. Wiegand, and S. Gordon, “H. 264/MPEG4-AVC Fidelity Range Extensions: Tools, Profiles, Performance, and application areas,” in *IEEE International Conference on Image Processing 2005*, vol. 1. IEEE, 2005, pp. I–593.
- [88] H. Schwarz, D. Marpe, and T. Wiegand, “Overview of the Scalable Video Coding Extension of the H. 264/AVC Standard,” *IEEE Transactions on circuits and systems for video technology*, vol. 17, no. 9, pp. 1103–1120, 2007. [41](#)
- [89] O.-Y. Lui and K.-W. Wong, “Chaos-based Selective Encryption for H. 264/AVC,” *Journal of Systems and Software*, vol. 86, no. 12, pp. 3183–3192, 2013. [41](#)
- [90] S. Lian, J. Sun, J. Wang, and Z. Wang, “A chaotic stream cipher and the usage in video protection,” *Chaos, Solitons & Fractals*, vol. 34, no. 3, pp. 851–859, 2007. [71](#)

- [91] H. Xu, X.-J. Tong, M. Zhang, Z. Wang, and L.-H. Li, "Dynamic Video Encryption Algorithm for H.264/AVC based on a Spatiotemporal Chaos System," *JOSA A*, vol. 33, no. 6, pp. 1166–1174, 2016. 41
- [92] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. ACM, 2007, p. 53. 43
- [93] O. Sinnen, *Task scheduling for parallel systems*. John Wiley & Sons, 2007, vol. 60. 43
- [94] U. Banerjee, R. Eigenmann, A. Nicolau, D. A. Padua *et al.*, "Automatic program parallelization," *Proceedings of the IEEE*, vol. 81, no. 2, pp. 211–243, 1993. 44
- [95] D. R. Butenhof, *Programming with POSIX threads*. Addison-Wesley Professional, 1997. 44
- [96] L. Dagum and R. Enon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998. 44, 46
- [97] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996. 44, 46
- [98] P. Pacheco, *An Introduction to Parallel Programming*, 1st ed. Morgan Kaufmann, 1 2011. [Online]. Available: <http://amazon.com/o/ASIN/0123742609/> 44, 46, 74
- [99] K. Ikudome, G. Fox, A. Kolawa, and J. Flower, "An automatic and symbolic parallelization system for distributed memory parallel computers," in *Distributed Memory Computing Conference, 1990., Proceedings of the Fifth*, vol. 2. IEEE, 1990, pp. 1105–1114. 44
- [100] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX Network Programming: The Sockets Networking API*. Addison-Wesley Professional, 2004, vol. 1. 44
- [101] S. Browne, J. Dongarra, and K. London, "Review of performance analysis tools for mpi parallel programs," *NHSE Review*, vol. 3, no. 1, pp. 241–248, 1998. 44
- [102] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced features of the message-passing interface*. MIT press, 1999. 46
- [103] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2004, pp. 97–104. 46
- [104] H.-Q. Jin, M. Frumkin, and J. Yan, "The openmp implementation of nas parallel benchmarks and its performance," NASA Ames Research Center, Tech. Rep., 1999. 46
- [105] F. Cappello and D. Etiemble, "Mpi versus mpi+ openmp on the ibm sp for the nas benchmarks," in *Supercomputing, ACM/IEEE 2000 Conference*. IEEE, 2000, pp. 12–12. 46
- [106] C. Sakamoto, T. Miyazaki, M. Kuwayama, K. Saisho, and A. Fukuda, "Design and implementation of a parallel pthread library (ppl) with parallelism and portability," *Systems and Computers in Japan*, vol. 29, no. 2, pp. 28–35, 1998. 46
- [107] A. Asaduzzaman, F. N. Sibai, S. Aramco, and H. El-Sayed, "Performance and power comparisons of mpi vs pthread implementations on multicore systems," in *Innovations in Information Technology (IIT), 2013 9th International Conference on*. IEEE, 2013, pp. 1–6. 46
- [108] H.-J. Boehm, "Threads cannot be implemented as a library," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 261–268. 46
- [109] B. Barney, "Posix threads programming," *National Laboratory. Disponível em:* < <https://computing.llnl.gov/tutorials/pthreads/> > Acesso em, vol. 5, 2009. 46

- [110] Wikipedia, “Fork–join model — wikipedia, the free encyclopedia,” 2016, [Online; accessed 18-February-2017]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Fork%E2%80%93join_model&oldid=738480477 47
- [111] J. M. Bull, “Measuring synchronisation and scheduling overheads in openmp,” in *Proceedings of First European Workshop on OpenMP*, vol. 8, 1999, p. 49. 52
- [112] M. Isida and H. Ikeda, “Random number generator,” *Annals of the Institute of Statistical Mathematics*, vol. 8, no. 1, pp. 119–126, 1956. 58
- [113] J. F. Dynes, Z. L. Yuan, A. W. Sharpe, and A. J. Shields, “A high speed, postprocessing free, quantum random number generator,” *applied physics letters*, vol. 93, no. 3, p. 031109, 2008. 58
- [114] Z. Gutterman, B. Pinkas, and T. Reinman, “Analysis of the linux random number generator,” in *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 2006, pp. 2 – 16. 58, 68, 87
- [115] G. McGraw, “Software security,” *IEEE Security & Privacy*, vol. 2, no. 2, pp. 80–83, 2004. 58
- [116] C. Lattner, “Llvm and clang: Next generation compiler technology,” 2008, pp. 1–2. 61
- [117] T. Kremenek, “Finding software bugs with the clang static analyzer,” *California: Apple Inc*, pp. 10–20, 2008. 61
- [118] D. Marjamaki, “cppcheck design,” pp. 1–10, 2014. 61
- [119] A. Almossawi, K. Lim, and T. Sinha, “Analysis tool evaluation: Coverity prevent,” *Pittsburgh, PA: Carnegie Mellon University*, pp. 7–11, 2006. 61
- [120] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, “Frama-c,” in *International Conference on Software Engineering and Formal Methods*. Springer, 2012, pp. 233–247. 61
- [121] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100. 62
- [122] J. Weidendorfer, “Sequential performance analysis with callgrind and kcachegrind,” in *Tools for High Performance Computing*. Springer, 2008, pp. 93–113. 62
- [123] A. Jannesari and W. F. Tichy, “Library-independent data race detection,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, pp. 2606–2616, 2014. 62
- [124] L. Kocarev, “Chaos-based cryptography: a brief overview,” *Circuits and Systems Magazine, IEEE*, vol. 1, no. 3, pp. 6–21, 2001. 67
- [125] G. Setti, R. Rovatti, and G. Mazzini, “Chaos-based generation of artificial self-similar traffic,” in *Complex Dynamics in Communication Networks*. Springer, 2005, pp. 159–190. 67
- [126] G. Cimatti, R. Rovatti, and G. Setti, “Chaos-based spreading in ds-ss sensor networks increases available bit rate,” *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 54, no. 6, pp. 1327–1339, 2007. 67
- [127] M. AbuTaha, S. El Assad, M. Farajallah, A. Queudet, and O. Deforge, “Chaos-based cryptosystems using dependent diffusion: An overview,” in *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*. IEEE, 2015, pp. 44–49. 67
- [128] A. Arlicot, “Sequences generator based chaotic maps,” Université de Nantes, Tech. Rep., FEB 2014. 67
- [129] D. Caragata, S. El Assad, H. Noura, and I. Tutanescu, “Secure unicast and multicast over satellite dvb using chaotic generators,” *International Journal of Internet Technology and Secured Transactions*, vol. 2, no. 3-4, pp. 357–379, 2010. 67
- [130] M. Chetto, S. El Assad, and M. Farajallah, “A lightweight chaos-based cryptosystem for dynamic security management in real-time overloaded applications,” *International Journal of Internet Technology and Secured Transactions 7*, vol. 5, no. 3, pp. 262–274, 2014. 67

- [131] S. Smale, “Differentiable dynamical systems,” *Bulletin of the American mathematical Society*, vol. 73, no. 6, pp. 747–817, 1967. [67](#)
- [132] L. Li and J.-H. Lee, “On the security of a strong provably secure identity-based encryption scheme without bilinear pairing,” *International Journal of Internet Technology and Secured Transactions*, vol. 6, no. 3, pp. 178–185, 2016. [67](#)
- [133] M. Masoumi, P. Habibi, A. Dehghan, M. Jadidi, and L. Yousefi, “Efficient implementation of power analysis attack resistant advanced encryption standard algorithm on side-channel attack standard evaluation board,” *International Journal of Internet Technology and Secured Transactions*, vol. 6, no. 3, pp. 203–218, 2016. [67](#)
- [134] I.-H. Jo and B.-S. Koh, “Building a common encryption scrambler to protect paid broadcast services,” *International Journal of Internet Technology and Secured Transactions*, vol. 6, no. 3, pp. 167–177, 2016. [67](#)
- [135] S. S. Gupta, A. Chattopadhyay, K. Sinha, S. Maitra, and B. P. Sinha, “High-performance hardware implementation for rc4 stream cipher,” *IEEE Transactions on Computers*, vol. 62, no. 4, pp. 730–743, 2013. [67](#)
- [136] C. Manifavas, G. Hatzivasilis, K. Fysarakis, and Y. Papaefstathiou, “A survey of lightweight stream ciphers for embedded systems,” *Security and Communication Networks*, vol. 9, pp. 1227–1246, 2015. [67](#)
- [137] J. Machicao, A. G. Marco, and O. M. Bruno, “Chaotic encryption method based on life-like cellular automata,” *Expert Systems with Applications*, vol. 39, no. 16, pp. 12 626–12 635, 2012. [68](#)
- [138] M. A. Taha, S. El Assad, A. Queudet, and O. Déforges, “Design and Efficient Implementation of a Chaos-based Stream Cipher,” in *Int. J. Internet Technology and Secured Transactions, Accepted paper*, 2017, pp. 1 – 16. [68](#), [105](#), [108](#)
- [139] S. El Assad, “Chaos based information hiding and security,” in *Internet Technology And Secured Transactions, 2012 International Conference for.* IEEE, 2012, pp. 67–72. [71](#)
- [140] N. Masuda and K. Aihara, “Cryptosystems with discretized chaotic maps,” *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, vol. 49, no. 1, pp. 28–40, 2002. [71](#)
- [141] Y. Wang, Z. Liu, J. Ma, and H. He, “A pseudorandom number generator based on piecewise logistic map,” *Nonlinear Dynamics*, vol. 83, no. 4, pp. 2373–2391, 2016. [79](#)
- [142] A. Akhshani, A. Akhavan, A. Mobaraki, S.-C. Lim, and Z. Hassan, “Pseudo random number generator based on quantum chaotic map,” *Communications in Nonlinear Science and Numerical Simulation*, vol. 19, no. 1, pp. 101–111, 2014. [79](#)
- [143] O. Jallouli, S. El Assad, M. A. Taha, M. Chetto, R. Lozi, and D. Caragata, “An efficient pseudo chaotic number generator based on coupling and multiplexing techniques,” in *International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2016)*, 2016, pp. 35–40. [79](#), [82](#)
- [144] C.-Y. Song, Y.-L. Qiao, and X.-Z. Zhang, “An image encryption scheme based on new spatiotemporal chaos,” *Optik-International Journal for Light and Electron Optics*, vol. 124, no. 18, pp. 3329–3334, 2013. [83](#)
- [145] B. Elaine and K. John, “Recommendation for random number generation using deterministic random bit generators,” NIST SP 800-90 Rev A, Tech. Rep., 2012. [83](#), [87](#), [97](#), [117](#)
- [146] B. Sunar, W. J. Martin, and D. R. Stinson, “A provably secure true random number generator with built-in tolerance to active attacks,” *Computers, IEEE Transactions on*, vol. 56, no. 1, pp. 109–119, 2007. [86](#)
- [147] J. von zur Gathen, *CryptoSchool*. Springer, 2015. [87](#)

- [148] Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergniaud, and D. Wichs, “Security analysis of pseudo-random number generators with input:/dev/random is not robust,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 647–658. 87
- [149] M. A. Taha, S. El Assad, O. Jallouli, A. Queudet, and O. Déforges, “Design of a pseudo-chaotic number generator as a random number generator,” in *The 11th International Conference on Communications*, 2016, pp. 401 – 404. 87, 103, 105, 108
- [150] O. Jallouli, S. El Assad, M. Chetto, R. Lozi, and D. Caragata, “A novel chaotic generator based on weakly-coupled discrete skewtent maps,” in *International Conference on Internet Technology and Secured Transactions*, 2015, pp. 38–43. 90, 91
- [151] B. Maxime, “Comparative analysis of estream ciphers,” Université de Nantes, Tech. Rep., March 2016. 92
- [152] T. Siegenthaler, “Decrypting a class of stream ciphers using ciphertext only,” *IEEE Transactions on computers*, vol. 100, no. 1, pp. 81–85, 1985. 94
- [153] P. P. Mar and K. M. Latt, “New analysis methods on strict avalanche criterion of s-boxes,” *World Academy of Science, Engineering and Technology*, vol. 48, pp. 150–154, 2008. 97, 119
- [154] Y. Wu, J. P. Noonan, and S. Aгаian, “Npcr and uaci randomness tests for image encryption,” *Cyber journals: multidisciplinary journals in science and technology, Journal of Selected Areas in Telecommunications (JSAT)*, vol. 1, pp. 31–38, 2011. 97, 119
- [155] X. Wang, D. Luan, and X. Bao, “Cryptanalysis of an image encryption algorithm using chebyshev generator,” *Digital Signal Processing*, vol. 25, pp. 244–247, 2014. 97, 119
- [156] Z. Shahid and W. Puech, “Visual Protection of HEVC Video by Selective Encryption of CABAC Binstrings,” *IEEE transactions on multimedia*, vol. 16, no. 1, pp. 24–36, 2014. 105
- [157] G. Van Wallendael, A. Boho, J. De Cock, A. Munteanu, and R. Van de Walle, “Encryption for High Efficiency Video Coding with Video Adaptation Capabilities,” *IEEE Transactions on Consumer Electronics*, vol. 59, no. 3, pp. 634–642, 2013. 105, 112
- [158] “Kvazaar HEVC Encoder .” [Online]. Available: <https://github.com/ultravideo/kvazaar> 105, 106
- [159] “HEVC Decoder.” [Online]. Available: <https://github.com/OpenHEVC/openHEVC> 105
- [160] F. H. H. Institute, “Reference Software Model (hm).” [Online]. Available: <https://hevc.hhi.fraunhofer.de/> 107, 110
- [161] G. Bjøntegaard, “VCEG-M33: Calculation of Average PSNR Differences Between RD-Curves,” Apr. 2001. 113
- [162] H. E.-d. H. Ahmed, H. M. Kalash, and O. F. Allah, “Encryption Efficiency Analysis and Security Evaluation of RC6 Block Cipher for Digital Images,” in *Electrical Engineering, 2007. ICEE’07. International Conference on*. IEEE, 2007, pp. 1–7. 114
- [163] N. Taneja, B. Raman, and I. Gupta, “Selective Image Encryption in Fractional Wavelet Domain,” *AEU-International Journal of Electronics and Communications*, vol. 65, no. 4, pp. 338–344, 2011. 115
- [164] R. L. Joshi and T. R. Fischer, “Comparison of Generalized Gaussian and Laplacian Modeling in DCT Image Coding,” *IEEE Signal Processing Letters*, vol. 2, no. 5, pp. 81–82, 1995. 115
- [165] Q. Meibing, C. Xiaorui, J. Jianguo, and Z. Shu, “Face Protection of H. 264 video based on Detecting and Tracking,” in *Electronic Measurement and Instruments, 2007. ICEMI’07. 8th International Conference on*. IEEE, 2007, pp. 2–172. 115
- [166] S. Bruce, “Applied cryptography: protocols, algorithms, and source code in c,” *John Wiley & Sons, Inc., New York*, 1996. 117
- [167] A. Said, “Measuring the strength of partial encryption schemes,” in *Image Processing, 2005. ICIP 2005. IEEE International Conference on*, vol. 2. IEEE, 2005, pp. II–1126. 117

- [168] I. R. Assembly, *Methodology for the Subjective Assessment of the Quality of Television Pictures*. International Telecommunication Union, 2003. [120](#), [121](#)
- [169] G. Gamst, L. S. Meyers, and A. Guarino, *Analysis of Vvariance Designs: A Conceptual and Computational Approach with SPSS and SAS*. Cambridge University Press, 2008. [122](#)

Appendices



Synthèse des travaux réalisés: Systèmes de Crypto-Compression basés Chaos en Temps Réel et Portables pour Architectures embarquées Efficaces

A.1 Contexte et objectifs:

Le développement extrêmement rapide des technologies de l'information et de la communication et de l'Internet des objets « Internet of Things » (IoT) a soulevé la problématique cruciale de la sécurité de données sensibles (bancaires, industrielles, militaires, médicales) et des données de l'IoT utilisées dans la vie de tous les jours, transitant sur des canaux publiques non protégés. La cryptographie est alors utilisée pour protéger le contenu de ces données sensibles, de différentes applications citées plus haut, contre les attaques cryptographiques passives et actives. Parmi les données sensibles, celles du multimédia (images et vidéo) sont de plus en plus répondues et très utilisées.

De nombreux algorithmes de chiffrement par bloc et par flux sont déjà développés et utilisés pour assurer la sécurité des données. Dans le chiffrement par bloc, tel que par exemple celui réalisé par l'algorithme standard AES, les données sont chiffrées bloc par bloc de taille déterminée (128 bits dans ce cas), et des opérations de confusion et de diffusion assez complexes sont réalisées sur les blocs en clair (plaintext). Des modes cryptographiques sont possibles et utilisés tel que le mode CBC (AES-CBC) permettant de rendre l'algorithme plus sécurisé et les modes CTR (AES-CTR) et OFB (AES-OFB) pour utiliser l'algorithme en tant que générateur de séquences pseudo-aléatoires. Le chiffrement par flux effectue le cryptage des données transmises en continu, bit par bit, octet par octet ou échantillon (de quelques octets) par échantillon. Le chiffrement proprement dit est une opération de confusion assez simple, réalisée par l'opérateur XOR entre les données en clair (plaintext) et le flux en sortie du générateur de nombres pseudo-aléatoires utilisé (keystream).

De nos jours, le développement de plus en plus accru de la cryptographie basée chaos, démontre son efficacité dans la protection des images et vidéos. En effet, les propriétés des systèmes chaotiques et déterministes telles que: ergodicité, sensibilité aux conditions initiales et paramètres de contrôle, nombre important de trajectoires très longues (apériodiques), etc., sont très recherchées pour tout système cryptographique dédié à la sécurité des données. Dans cette thèse, nous nous sommes intéressés au chiffrement par flux qui est plus adéquat pour assurer la protection des images transmises en continu et des vidéo. Dans ce contexte,

la conception et la réalisation de générateurs de nombres pseudo-chaotiques (PCNGs) sécurisés de point de vue cryptographique, jouent un rôle central dans la protection des données sensibles. Par ailleurs, les méthodes de crypto-compression qui traitent le flux vidéo comme des données uniques, sans prendre en compte la structure même de la vidéo compressée, ne sont pas appropriées pour des applications temps réel. Dernièrement, plusieurs travaux de la littérature ont été consacrés à la protection de flux vidéo de la récente norme HEVC «High Efficiency Video Coding».

Les objectifs de cette thèse consistent à concevoir, réaliser et analyser de nouvelles solutions technologiques basées chaos permettant de répondre à la problématique et aux défis soulevés par la sécurité des données sensibles. Pour atteindre les objectifs visés, nous concevons d'abord des PCNGs très efficaces (robustes et rapides), puis des systèmes chaotiques de chiffrement par flux pour les images, et enfin des systèmes de crypto-compression à base de chaos pour protéger les flux vidéo HEVC.

A.2 Résumé des travaux:

La protection des images et vidéos est une problématique cruciale. Dans ce travail nous avons d'abord, conçu et réalisé d'une façon efficace et sécurisée un générateur de nombre pseudo-chaotique (PCNG) mis en œuvre en séquentielle et en parallèle par P-threads. Basé sur ces PCNGs, deux applications centrales ont été conçues, mises en œuvre et analysées. La première traite la réalisation d'un générateur de nombre aléatoire et les résultats obtenus sont très prometteurs. La deuxième concerne la réalisation d'un système de chiffrement/déchiffrement par flux. L'analyse cryptographique des systèmes chaotiques réalisés montre leur robustesse contre des attaques connues. Ce résultat est dû à la structure récursive proposée qui intègre une forte non-linéarité, une technique de perturbation et un multiplexage chaotique. La performance obtenue en complexité de calcul autorise leur utilisation dans des applications temps réel. Ensuite, basé sur le système chaotique précédent, nous avons conçu et mis en œuvre efficacement un système de crypto-compression pour des applications temps réel et portable pour architectures embarquées. Une solution de chiffrement par flux sélectif des contenus vidéo HEVC est réalisée. Puis, un chiffrement d'une région d'intérêt est effectué au niveau CABAC pour les paramètres les plus sensibles incluant des vecteurs de mouvement et des coefficients transformés. Le format le chiffrement conforme de Modes de Prédiction Intra a été aussi vérifié. L'évaluation subjective et des tests de complexité d'altération de taux objectifs ont montré que la solution proposée sécurise le contenu vidéo avec un débit binaire et une complexité de codage légèrement augmentés.

Mots clés :

Générateur de nombres pseudo-chaotiques, RNG, Chiffrement par flux basé chaos, P-threads, Systèmes de Crypto-Compression basés Chaos, Chiffrement sélectif, HEVC, Analyse de la sécurité, Complexité de calcul.

Contributions:

Dans la suite, nous décrivons l'essentiel des contributions réalisées dans cette thèse.

Première contribution : Réalisation des générateurs de nombres pseudo-chaotiques (PCNGs)

L'architecture proposée dans la Figure A.1, pour la réalisation des PCNGs est composé de:

1. un bloc état interne de forte complexité cryptographique ;
2. une fonction de sortie;
3. un bloc de traitement "IV-Setup" (Initialisation Vector) ;
4. un bloc de traitement "Key-setup" (permettant de faire un calcul en parallèle utilisant P-thread);
5. une mémoire non-volatile.

Basé sur cette architecture, nous avons proposé un premier PCNG1 décrit par la Figure A.2 et un deuxième PCNG2 qui diffère du premier uniquement par la fonction de sortie, qui est en l'occurrence

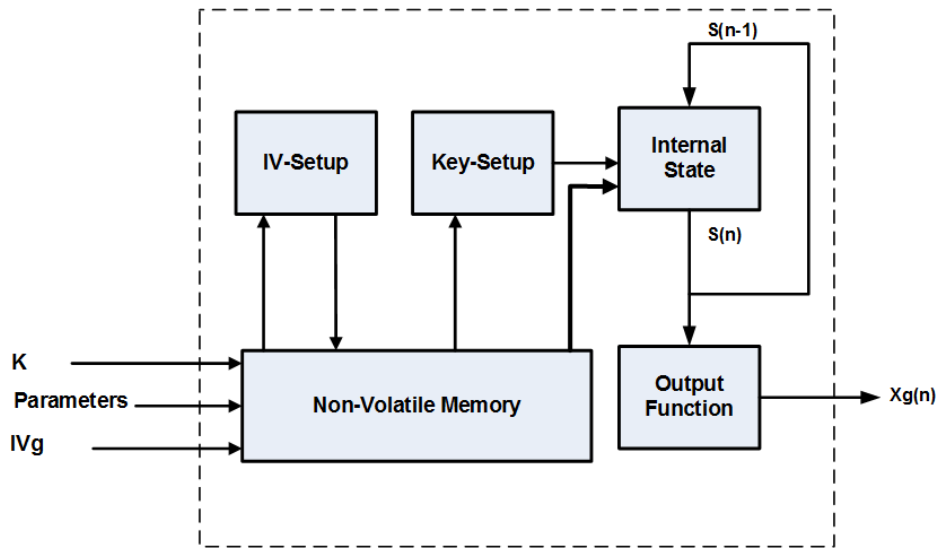


Figure A.1 – Architecture des PCNGs

une opération de multiplexage chaotique entre $X_s(n)$ et $X_p(n)$ pour le PCNG1 ou une opération XOR entre $X_s(n)$ et $X_p(n)$ pour le PCNG2. Les entrées U_s et U_p du générateur de base prennent leurs valeurs à partir du vecteur initial du générateur IV_g , supposé un Nonce : $U_s = LSB_{32}(IV_g)$ et $U_p = MSB_{32}(IV_g)$. L'état interne du générateur chaotique comprend deux filtres récurrents discrets, contenant chacun une fonction non linéaire : une carte chaotique Skew tent pour le premier filtre récurrent et une carte PWLC pour le deuxième filtre récurrent. Chaque filtre récurrent utilise aussi une technique de perturbation basée LFSR et génère à sa sortie une séquence pseudo-chaotique de valeurs entières : X_s pour la cellule récurrente utilisant la carte Skew tent et X_p pour la cellule récurrente utilisant la carte PWLC. Le générateur peut travailler avec un retard variable entre 1 et 3 correspondant à d'éventuelles opérations supplémentaires des filtres. Ces retards sont choisis selon le degré de sécurité nécessaire et la vitesse de génération de séquences requise de l'application envisagée. La méthode de perturbation trouve son fondement sur le fait qu'aucun cycle stable n'existe, c.-à-d. si le système chaotique décrit, à un moment donné, un cycle donné, il peut, par application d'une perturbation, quitter ce cycle immédiatement pour aller vers un autre cycle. Le choix de la séquence perturbatrice est effectué selon les règles suivantes : elle devrait avoir une longue longueur de cycle contrôlable et une distribution uniforme; elle ne devrait pas dégrader les bonnes propriétés statistiques de la dynamique chaotique, donc l'amplitude du signal perturbateur doit être nettement plus petite que celle du signal chaotique, de sorte que le rapport R entre les deux amplitudes maximales, soit supérieur ou égal à 40 dB :

$$R = 20 \times \log \left[\frac{\text{Amplitude max imale de signal chaotique}}{\text{Amplitude max imale de signal perturbateur}} \right] \geq 40 \text{ db} \quad (\text{A.1})$$

Un bon candidat pour la génération de séquences perturbatrices est le registre à décalage à réaction à longueur maximale. En effet, ce dernier est caractérisé par : une bonne fonction d'autocorrélation, par une distribution presque uniforme, par un cycle de longueur maximale égale à $2^k - 1$ (k est le degré du polynôme primitif utilisé) et une implémentation logicielle ou matérielle facile.

L'implémentation logicielle en code C du générateur est effectuée en séquentielle et en parallèle utilisant P-thread. Cette implémentation est sécurisée comme suit: avant tout appel au générateur, une allocation d'un espace mémoire par appel à la fonction malloc() est réalisée. Cet espace mémoire est ensuite verrouillé par la fonction mlock() afin d'éviter, dans le cas de surcharge du système, que certaines données sensibles ne soient copiées dans l'espace d'échange (swap). Une fois la séquence chaotique produite puis sauvegardée dans un fichier, la mémoire est déverrouillée par la fonction munlock() puis effacée par la fonction memset() associée au pointeur const volatile memset_ptr.

En informatique, un processus multi-thread est un processus qui contient plusieurs fils d'exécution ap-

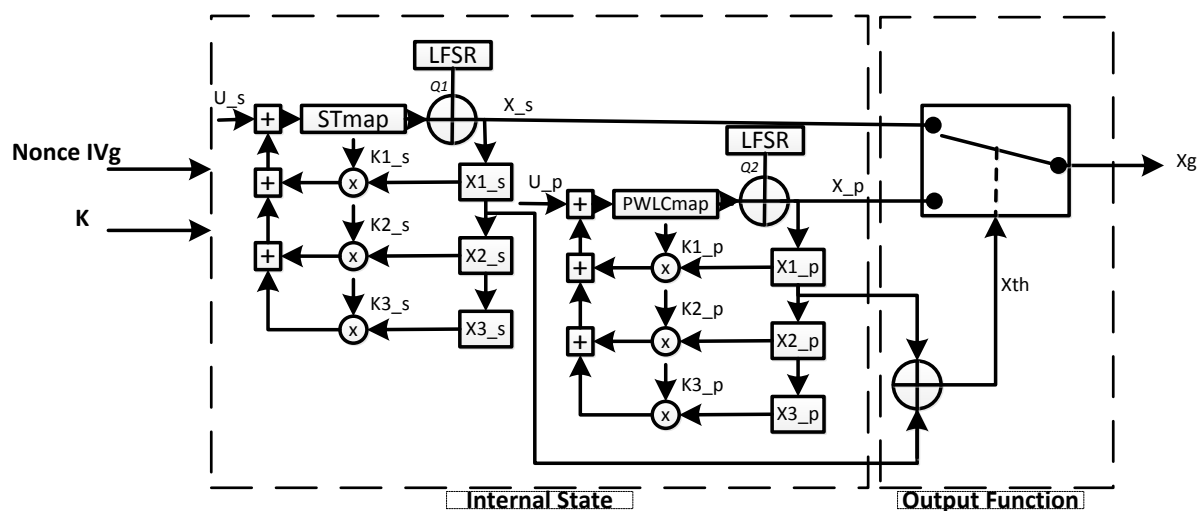


Figure A.2 – Description du PCNG1 proposé

pelés threads qui s'exécutent de manière concurrente. Au sein de notre implémentation, nous avons parallélisé la version séquentielle du générateur chaotique en utilisant l'API standard Pthread définie dans la norme POSIX (Pthread) qui est une bibliothèque de fonctions mise à disposition des programmeurs souhaitant implémenter des programmes parallèles. Contrairement à MPI, Pthread est utilisé pour implémenter du parallélisme sur des machines à mémoire partagée. Il ne s'agit pas d'un langage de programmation tels que le C ou le Java mais d'une librairie devant être liée à la compilation avec les programmes sources. Ainsi, le code source écrit en langage C est compilé à l'aide de gcc en rajoutant l'option `-lpthread`. Dans notre approche multi-thread, les séquences de données sont partitionnées et réparties entre plusieurs threads. Les threads exécutent les mêmes instructions mais sur des ensembles de données différents. Le nombre d'échantillons à traiter, plus précisément le sous-ensemble de données associées aux échantillons traités par chacun des threads, est différent. Les différents threads sont créés et lancés via l'appel système `pthread_create()`. Dans notre cas, nous créons un nombre de threads égal au nombre de cœurs disponibles sur la plate-forme d'exécution. La fonction `pthread_create()` prend notamment en paramètres d'entrée un descripteur de thread. Chaque thread fait appel à la fonction de traitement computation auquel il est associé. Cette fonction assure la génération des échantillons et la conversion en octets. Par la suite, les séquences calculées par les différents threads sont stockées dans un buffer de manière systématique afin d'obtenir un gain de performance maximal. Chaque séquence issue de chaque thread est stockée consécutivement. Dans la fonction `main()`, le fil d'exécution principal qui au préalable a explicitement créé les différents threads, attend la terminaison de tous les threads qu'il a créés via des appels successifs à la fonction `pthread_join()`.

```
int pthread_create(pthread_t *restrict thread ,
    const pthread_attr_t *restrict attr ,
    void *(*computation)(void*) , void *restrict arg );
```

Deuxième contribution : Réalisation de système de chiffrement par flux base PCNG1.

Basé sur le PCNG1, nous avons réalisé et analysé le système de chiffrement/déchiffrement par flux donné par la Figure A.3. On voit clairement que toute la complexité cryptographique et la sécurité réside dans le générateur de nombres pseudo-chaotiques utilisé.

L'analyse de la sécurité des systèmes réalisée et les résultats obtenus lors des différents tests expérimentaux appliqués montrent leur robustesse contre les attaques cryptographiques et statistiques connues. Par ailleurs, les performances obtenues en termes de nombre de cycles nécessaires pour chiffrer un octet (NCpB) indiquent leur intérêt pour des applications temps réel. Ci-dessous, nous donnons quelques résultats obtenus pour le système de chiffrement par flux. Le test statistique χ^2 , appliqué sur l'histogramme de l'image chiffrée prouve l'uniformité de l'histogramme sous test. Aussi, le test statistique de NIST ap-

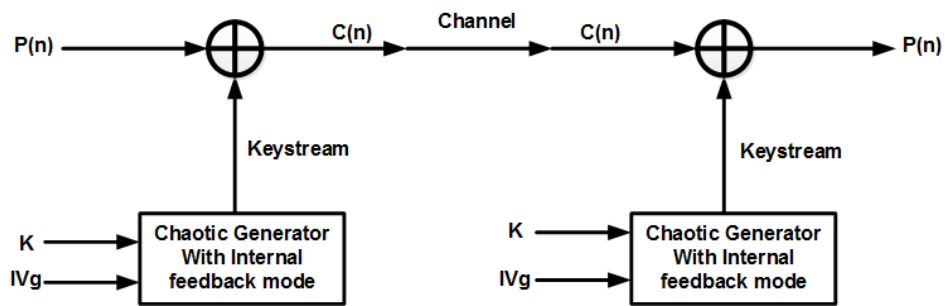


Figure A.3 – Système de chiffrement/déchiffrement par flux

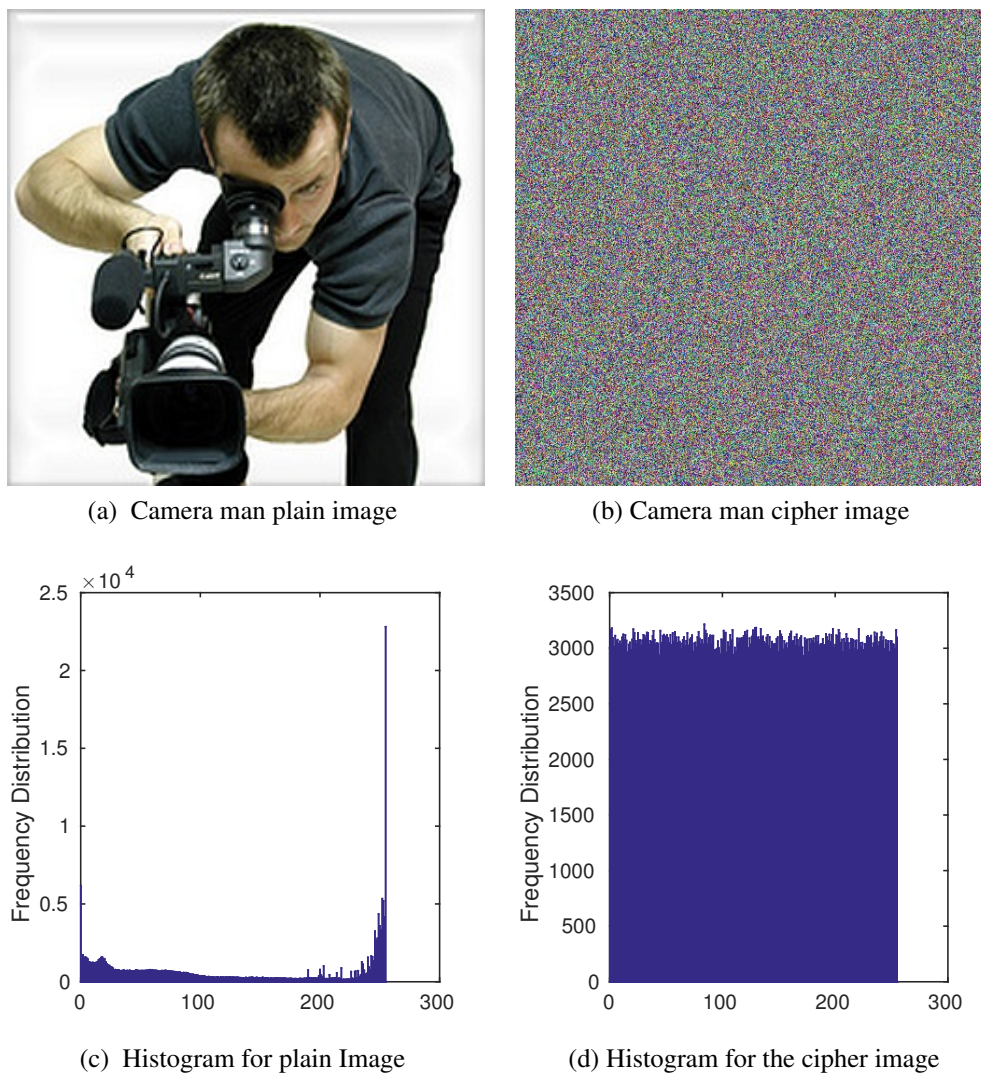


Figure A.4 – a) Image en clair de Caméraman, b) Image chiffrée correspondante, c) Histogramme de l’image en clair, d) Histogramme de l’image chiffrée.

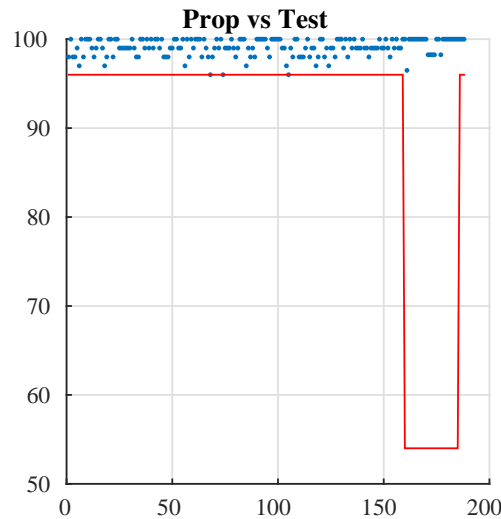


Figure A.5 – Proportion des tests de NIST.

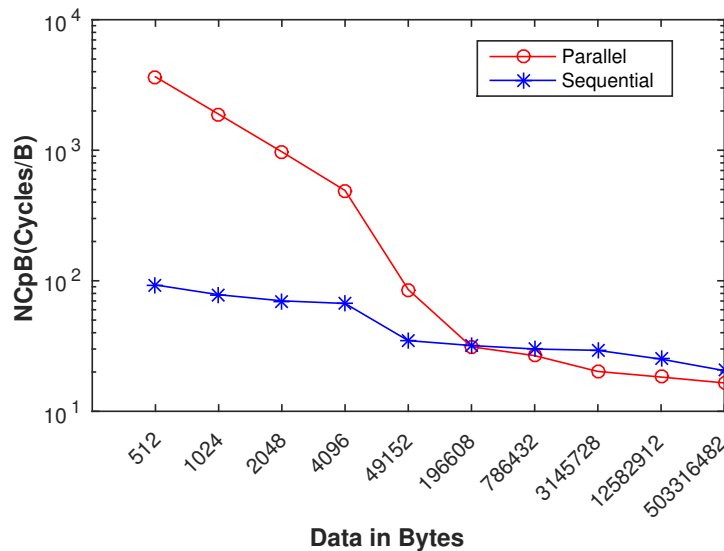


Figure A.6 – Nombre de cycles nécessaires pour chiffrer un octet en fonction de la taille des données.

pliqué sur les séquences générées indique la propriété de quasi-aléatoire de ces séquences. Ci-dessous sur la Figure A.5, nous montrons un exemple de résultat obtenu de test de NIST sur 100 séquences produites, chacune contenant 1 million de bits. Toutes les séquences passent le test de NIST composé de 188 tests et sous tests. Enfin, nous donnons ci-dessous sur la Figure A.6, les performances obtenues du système de chiffrement par flux en termes de nombre de cycles nécessaires pour chiffrer un octet (NCpB) en fonction de la taille des données chiffrées dans les deux cas de figures de l'implémentation séquentielle et parallèle. Ces résultats montrent que, pour des tailles des données à partir de 196608 Octets, le NCpB du système proposé est comparable à celui obtenu par les algorithmes du projet eStream (Rabbit, HC-128, Salsa20/12, SOSEMANNUK, etc.,) et est meilleur pour les très grandes données (à partir de 503316482).

Troisième contribution : réalisation d'un système de crypto-compression basé chaos pour le chiffrement sélectif des flux vidéo HEVC.

Le système proposé de crypto-compression basé chaos pour le chiffrement sélectif des flux vidéo HEVC au niveau CABAC est décrit par la Figure A.7. La solution proposée crypte un ensemble de paramètres

HEVC les plus sensibles incluant le vecteur de mouvement (MV), les signes des différences des MV, les coefficients transformés (TCs), ainsi que leur signe. La solution de chiffrement sélectif proposée conserve la conformité des paramètres « Intra Prediction Modes » (IPMs) de la luminance et de la chrominance au format HEVC. Basé sur le concept de tuile, le système de crypto-compression permet aussi une protection de la vidéo au niveau d'une Région d'Intérêt (ROI) définie dans le standard HEVC. Il permet d'éviter la propagation du chiffrement à l'extérieur des frontières ROI.

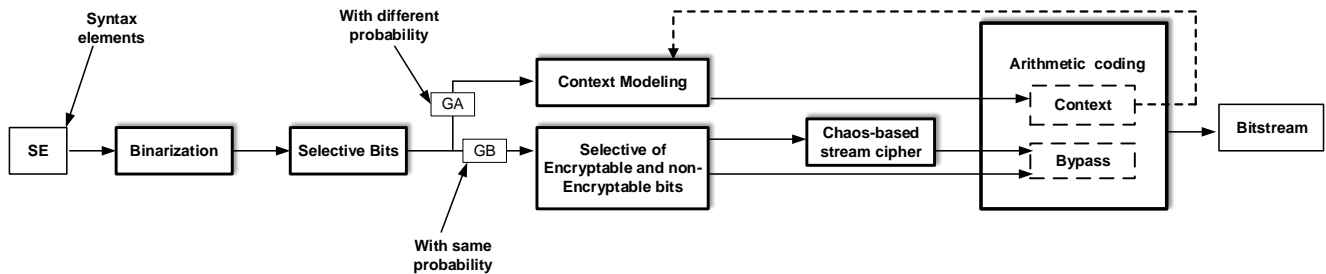


Figure A.7 – Selective encryption in HEVC at CABAC level.



Static and dynamic analysis tools

B.1 Software security analysis tools

In order to validate the correctness of our solution, we conducted a security code review using several static and dynamic techniques: Clang, Gdb, Valgrind, DRD, Callgrind and Leak-analysis tools.

B.1.1 Static Software security analysis tools

B.1.1.1 Clang Static Analyzer

Installation:First we install the clang package from the Ubuntu software center, then we use tool to check our code by writing the following command in the terminal:

```
clang --analyze main.c -pthread -lm -o test
```

The clang tool give us some warnings, in the following we provide the warnings and the solutions:

In file included from main.c:26:

```
./cartes.c:259:16: warning: ^ has lower precedence than <=; <= will be  
evaluated first [-Wparentheses] if ( K->X_1s-1 ^ K->X_1p-1 <= 2147483648)
```

```
Solution : if ( (K->X_1s-1 ^ K->X_1p-1)<= 2147483648)
```

```
./cartes.c:259:16: note: place parentheses around the '<=' expression to silence  
this warning if ( K->X_1s-1 ^ K->X_1p-1 <= 2147483648)
```

```
Solution: if ( (K->X_1s-1 ^ K->X_1p-1)<= 2147483648)
```

```
./cartes.c:259:16: note: place parentheses around the ^ expression to evaluate  
it first if ( K->X_1s-1 ^ K->X_1p-1 <= 2147483648)
```

```
Solution: if ( (K->X_1s-1 ^ K->X_1p-1)<= 2147483648)
```

```
./cartes.c:312:16: warning: ^ has lower precedence than <=; <= will be evaluated  
first [-Wparentheses] if ( K->X_1s-1 ^ K->X_1p-1 <= 2147483648)
```

```
solution: if ( (K->X_1s-1 ^ K->X_1p-1) <= 2147483648)
```

```
./cartes.c:312:16: note: place parentheses around the '<=' expression to
silence this warning if ( K->X_1s-1 ^ K->X_1p-1 <= 2147483648)
```

```
Solution: if ( (K->X_1s-1 ^ K->X_1p-1)<= 2147483648)
```

```
./cartes.c:312:16: note: place parentheses around the ^ expression to
evaluate it first if ( K->X_1s-1 ^ K->X_1p-1 <= 2147483648)
      ^
```

```
Solution: if ( (K->X_1s-1 ^ K->X_1p-1)<= 2147483648)
```

```
main.c:185:8: warning: implicit declaration of function 'mlock' is invalid in
C99 [-Wimplicit-function-declaration] mlock (K, sizeof(*K));
```

```
Solution we add the header #include <sys/mman.h>
      ^
```

```
main.c:225:5: warning: implicit declaration of function 'gettimeofday' is
invalid in C99 [-Wimplicit-function-declaration] gettimeofday(&start, NULL);
```

```
Solution: we add the header #include <sys/time.h>
```

```
main.c:230:51: warning: cast to 'void *' from smaller integer type 'int'
[-Wint-to-void-pointer-cast] if(pthread_create(&th[i],NULL,computation,(void *)i)<0)
```

Solution: we rewrote the Pthread function as follows:

```
#include <pthread.h>
#include <stdio.h>
#include <stdint.h>
```

```
void * fct (void *arg ) {
    int id = (intptr_t)(arg );
    printf("%d\n" , id );
    pthread_exit(NULL);
}
```

```
int main () {
    pthread_t threads [ 8 ] ;
    unsigned int i ;
    for (i = 0 ; i < 8 ; ++i) {
        pthread_create(&threads [ i ] , NULL, fct , (void*)(intptr_t) i ) ;
    }

    for(i = 0 ; i < 8 ; ++i) {
        pthread_join ( threads [ i ] , NULL) ;
    }
    return 0;
}
```

```
main.c:330:6: warning: implicit declaration of function 'munlock' is
invalid in C99 [-Wimplicit-function-declaration] munlock(K);
```

```
Solution: we add the header #include <sys/mman.h>
```

B.1.1.2 Frama-c

It is a set of tools dedicated to the analysis of source code written in C. It gathers several static analysis techniques in a single collaborative framework. Frama-C is Open Source software.

Installation: from Ubuntu software center.

Results: as we can see in the snapshot below no errors were found in our code, all processes are valid.



Figure B.1 – Frama-c analysis report.

B.1.1.3 Cppcheck

First we download the cppcheck tool from: cppcheck.sourceforge.net, then we apply the following commands on the terminal:

```
cd gui
qmake
make
cppceck main.c -pthread -lm -o test
```

Cppcheck 1.76 results :

```
[main.cpp:3]: (style) Variable 'p' is not assigned a value.
[main.cpp:4]: (error) Uninitialized variable: p
```

Solution: we assigned initial value for p.

B.1.2 Dynamic Software security analysis tools

B.1.2.1 Efficient memory-leak tracer for C/C++ programs (open source)

A memory leak is a type of resource leak (particular type of resource consumption by a computer program where the program does not release resources it has acquired), that occurs when a computer program incorrectly manages memory allocations in such a way that memory which is no longer needed is not released. **leak tracer tool** for C/C++ programs. It shows if your program have a memory leak , together with leak count and size.

Installation: we can install from Ubuntu software center. Then we type the following commands in the terminal.

```
gcc main.c -pthread -lm -o test
LeakCheck ./test
```

The results of using this tool in our code is provided hereafter. As we can see the memory leak in our code is zero.

```
#####
# memory overrun protection of 4 Bytes
# initializing new memory with 0xAA
```



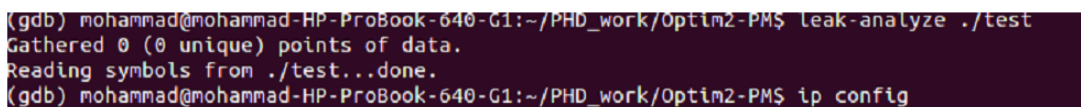
```
# sweeping deleted memory with 0xEE
# aborts on OVERWRITE_MEMORY
# thread save
# leak      0 Bytes      :-)
#####
```

B.1.2.2 Leak-analyzer (open source)

This tool is the same as previous one, but its work with the gdb debugger to analysis the code, that means we must add the option `-g` in the command line as follows:

```
gcc -g main.c -pthread -lm -o test
leak-analyze ./test
```

The result of this tool in our code indicates that it is free from memory errors (see Figure B.2)



```
(gdb) mohammad@mohammad-HP-ProBook-640-G1:~/PHD_work/Optim2-PMS$ Leak-analyze ./test
Gathered 0 (0 unique) points of data.
Reading symbols from ./test..done.
(gdb) mohammad@mohammad-HP-ProBook-640-G1:~/PHD_work/Optim2-PMS$ ip config
```

Figure B.2 – Leak analyzer report.

B.1.2.3 Valgrind

The Valgrind provides a number of debugging and profiling tools that help you make your programs faster and more correct. The most popular of these tools is called Memcheck. It can detect many memory-related errors that are common in C and C++ programs and that can lead to crashes and unpredictable behavior.

Installation: we can download valgrind with all its package(memchek,drd,callgrind,gdb)from the Ubuntu software center, its require about 30 minutes to complete our program checking. We can run it using the following commands:

```
gcc main.c -pthread -lm -o test
valgrind --tool=memcheck --leak-check=yes ./test
```

The results obtained from the Valgrind give us a strong indication that our program haven't any memory leaks, as we see in the results of valgrind checker in Figure B.3 that all heap blocks were freed and no leaks are possible.

B.1.2.4 Callgrind

We use this tool in order to check the cashe memory, by typing the following command in the terminal:

```
valgrind --tool=callgrind --simulate-cache=yes ./test
```

Our code is free from cache error, as we can see in Figure B.4 that represents the analysis report of Callgrind .

B.1.2.5 Thread error detector (DRD)

DRD is a tool that used to check the thread errors in the program. DRD is a Valgrind tool for detecting errors in multithreaded C and C++ programs. The tool works for any program that uses the POSIX threading primitives or that uses threading concepts built on top of the POSIX (Pthread) threading primitives. The command for running this tool provided in the following:

```

==5724== Memcheck, a memory error detector
==5724== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==5724== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==5724== Command: ./test
==5724==
*****
*   You are testing the basic map -- parallel version with pthread()   *
*****
number of cores detected in this computer is 4

->Enter the number of sequences (n=100 for NIST test and m=1 for mapping test): 100

--For your information, you are using N=32 bits--
->Enter the number of sample do you want for each sequences (for example 31250 samples produce 1000000 bits with N=32 bits): 31250
You will compute 1000000 bits for this test !
*****
*   perturbation   *
*****
please enter 1 for perturbation and 2 without perturbation
1
The process is done .....Press any key to finish
==5724==
==5724== HEAP SUMMARY:
==5724==   in use at exit: 0 bytes in 0 blocks
==5724==   total heap usage: 834 allocs, 834 frees, 25,574,175 bytes allocated
==5724==
==5724== All heap blocks were freed -- no leaks are possible
==5724==
==5724== For counts of detected and suppressed errors, rerun with: -v
==5724== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Figure B.3 – Valgrind analysis report.

```

==6409== Callgrind, a call-graph generating cache profiler
==6409== Copyright (C) 2002-2013, and GNU GPL'd, by Josef Weidendorfer et al.
==6409== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==6409== Command: ./test
==6409==
--6409-- warning: L3 cache found, using its data for the LL simulation.
==6409== For interactive control, run 'callgrind_control -h'.
*****
*   You are testing the basic map -- parallel version with pthread()   *
*****
number of cores detected in this computer is 4

->Enter the number of sequences (n=100 for NIST test and m=1 for mapping test): 100

--For your information, you are using N=32 bits--
->Enter the number of sample do you want for each sequences (for example 31250 samples produce 1000000 bits with N=32 bits): 31250
You will compute 1000000 bits for this test !
*****
*   perturbation   *
*****
please enter 1 for perturbation and 2 without perturbation
1
The process is done .....Press any key to finish
==6409==
==6409== Events   : Ir Dr Dw Iimr Dimr Dimw ILmr DLmr DLMw
==6409== Collected : 61237660556 16453094634 12649892518 24442 1178219 394874 2672 1175747 394217
==6409==
==6409== I refs:    61,237,660,556
==6409== I1 misses:    24,442
==6409== L1l misses:    2,672
==6409== I1 miss rate:    0.0%
==6409== L1l miss rate:    0.0%
==6409==
==6409== D refs:    28,502,987,152 (16,453,094,634 rd + 12,049,892,518 wr)
==6409== D1 misses:    1,573,093 ( 1,178,219 rd + 394,874 wr)
==6409== L1d misses:    1,569,964 ( 1,175,747 rd + 394,217 wr)
==6409== D1 miss rate:    0.0% ( 0.0% + 0.0% )
==6409== L1d miss rate:    0.0% ( 0.0% + 0.0% )
==6409==
==6409== LL refs:    1,597,535 ( 1,202,661 rd + 394,874 wr)
==6409== LL misses:    1,572,636 ( 1,178,419 rd + 394,217 wr)
==6409== LL miss rate:    0.0% ( 0.0% + 0.0% )

```

Figure B.4 – Callgrind analysis report.

```
valgrind --tool=drd ./test
```

The results of applying this tool in our program give zero errors (see Figure B.5).

B.2 Some of tools and limitations

In the following we display some tools that we cannot use them in our program.

B.2.1 Csur tool

csur is a generic C code analyzer. Csur is distributed freely provided you don't use it for any commercial purpose. Nevertheless, concerning its limitations, the most problematic is that it does not support applications build on various .c files (as it is our case with main.c, cartes.c, util.c...).

<http://www.lsv.ens-cachan.fr/goubault/Csur/csur.html>

```

==6086== drd, a thread error detector
==6086== Copyright (C) 2006-2013, and GNU GPL'd, by Bart Van Assche.
==6086== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==6086== Command: ./test
==6086==
*****
*   You are testing the basic map -- parallel version with pthread()   *
*****
number of cores detected in this computer is 4

->Enter the number of sequences (n=100 for NIST test and n=1 for napping test): 100
--For your information, you are using N=32 bits--
->Enter the number of sample do you want for each sequences (for example 31250 samples produce 1000000 bits with N=32 bits): 31250
You will compute 1000000 bits for this test !
*****
*   perturbation   *
*****
please enter 1 for perturbation and 2 without perturbation
1
The process is done .....Press any key to finish
==6086==
==6086== For counts of detected and suppressed errors, rerun with: -v
==6086== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 65 from 43)

```

Figure B.5 – DRD analysis report.

B.2.2 Boon tool

This tool applies integer range analysis to determine whether a C program can index an array outside its bounds. While capable of finding many errors that lexical analysis tools would miss, this tool checker is still imprecise: it ignores statement order, it can't model inter-procedural dependencies, and it ignores pointer aliasing.

<https://www.cigital.com/blog/static-analysis-for-security/>

B.2.3 Cqual tool

CQual uses type qualifiers to perform a taint analysis, which detects format string vulnerabilities in C programs. CQual requires a programmer to annotate a few variables as either tainted or untainted and then uses type inference rules (along with pre-annotated system libraries) to propagate the qualifiers. <https://www.cigital.com/blog/static-analysis-for-security/>

B.2.4 Parfai tool and coverity tool

Parfait is similar to the Coverity analysis tool that has been used on the kernel as well as other free software. In both cases, at least for now, the analysis can only be run by the company who owns the tool, or those who have licensed it in the case of Coverity. <https://lwn.net/Articles/344003/>

B.2.5 Blast tool

The big problem in this tool after many years of service, they do not actively maintain BLAST anymore. Another problem it is need the program to satisfy behavioral properties of the interfaces it use.

<http://mtc.epfl.ch/software-tools/blast/index-epfl.php>



Code Documentation

C.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

key	
Parameters of key	155

C.2 File List

Here is a list of all files with brief descriptions:

cartes.c	
Map program	159
cartes.h	163
main.c	
Main program. stream version v3: created by Mohammad Abutaha	166
util.c	
key and parameter	167
util.h	168

C.3 key Struct Reference

```
parameters of key.  
#include <util.h>
```

Public Attributes

- int **N**
- uint64_t **Q**
- uint64_t **g_s**
- uint64_t **g_p**
- uint64_t **reg_s**
- uint64_t **reg_p**
- uint64_t **mask**
- int **delay**

```

— uint64_t P_s
— uint64_t P_p
— uint64_t X_s
— uint64_t X_p
— uint64_t X_1s
— uint64_t tab_Xdelay_s [4]
— uint64_t tab_k_s [4]
— uint64_t U_s
— uint64_t X_1p
— uint64_t tab_Xdelay_p [4]
— uint64_t tab_k_p [4]
— uint64_t U_p
— int maxdegree_p
— int maxdegree_s
— int delta_s
— int delta_p
— int tr
— uint64_t IV

```

C.3.1 Detailed Description

parameters of key.

This structure contain all the parameters of they key.

C.3.2 Member Data Documentation

C.3.2.1 delay

```
int key::delay
```

C.3.2.2 delta_p

```
int key::delta_p
```

C.3.2.3 delta_s

```
int key::delta_s
```

C.3.2.4 g_p

```
uint64_t key::g_p
```

C.3.2.5 g_s

```
uint64_t key::g_s
```

C.3.2.6 IV

```
uint64_t key::IV
```

C.3.2.7 mask

```
uint64_t key::mask
```

C.3.2.8 maxdegree_p

```
int key::maxdegree_p
```

Maximal polynome degree for PWLC map.

C.3.2.9 maxdegree_s

```
int key::maxdegree_s
```

Maximal polynome degree for skewtent map.

C.3.2.10 N

```
int key::N
```

C.3.2.11 P_p

```
uint64_t key::P_p
```

C.3.2.12 P_s

```
uint64_t key::P_s
```

$0 < P < 2^N$ for skewtent and $0 < P < 2^{(N-1)}$ for skewtent.

C.3.2.13 Q

```
uint64_t key::Q
```

C.3.2.14 reg_p

```
uint64_t key::reg_p
```

C.3.2.15 reg_s

```
uint64_t key::reg_s
```

C.3.2.16 tab_k_p

```
uint64_t key::tab_k_p[4]
```

C.3.2.17 tab_k_s

```
uint64_t key::tab_k_s[4]
```

Constant value for the first delay $0 < k_1 < 2^N$

C.3.2.18 tab_Xdelay_p

```
uint64_t key::tab_Xdelay_p[4]
```

In the recursive cell, X1 value correspond to the first delay $0 < X_1 < 2^N$

C.3.2.19 tab_Xdelay_s

```
uint64_t key::tab_Xdelay_s[4]
```

In the recursive cell, X1 value correspond to the first delay $0 < X_1 < 2^N$

C.3.2.20 tr

```
int key::tr
```

C.3.2.21 U_p

```
uint64_t key::U_p
```

Constant value for the first delay $0 < k_1 < 2^N$ Constant value $0 < U < 2^N$

C.3.2.22 U_s

```
uint64_t key::U_s
```

Constant value $0 < U < 2^N$

C.3.2.23 X_1p

```
uint64_t key::X_1p
```

Called X-1 to understand that it is the precedent result X re-injected in the map in the future loop.

C.3.2.24 X_1s

```
uint64_t key::X_1s
```

Called X-1 to understand that it is the precedent result X re-injected in the map in the future loop.

C.3.2.25 X_p

```
uint64_t key::X_p
```

C.3.2.26 X_s

uint64_t key::X_s

Result of the map $0 < X < 2^N$

The documentation for this struct was generated from the following file:

— **util.h**

C.4 cartes.c File Reference

Map program.

```
#include "cartes.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdint.h>
#include <assert.h>
#include <omp.h>
```

Macros

— #define **BIT**(n, x) (((x) >> (n)) & 1)
Shift a value from n bits.

Functions

— uint64_t **PWLCmap** (uint64_t X_1, uint64_t P, int N, double ratio3, double ratio4, uint64_t m1, uint64_t m2)

— uint64_t **STmap** (uint64_t X_1, int N, uint64_t P, double ratio1, double ratio2, uint64_t m1, uint64_t m2)

— void **write_dec** (uint64_t *V, int L, FILE *FIC)
Writes in the file FIC the L elements of the sequence V.

— int **maximal_degree_of_polynome** (uint64_t polynome)

— uint64_t **LFSR** (int maxdegree, int G, uint64_t Q)
Create a linear shift in the register.

— uint64_t **generator** (key *K, int iter_sec, double ratio1, double ratio2, double ratio3, double ratio4, uint64_t m1, uint64_t m2)
Create a chaotic sample.

— uint64_t **generator_no_perturbation** (key *K, int iter_sec, double ratio1, double ratio2, double ratio3, double ratio4, uint64_t m1, uint64_t m2)

— void **iv_setup** (FILE **Fic_parametre, key *K)

— void **write_bin** (uint64_t *V, int L, FILE *FIC, int N)

C.4.1 Detailed Description

Map program.

Author

mohammad Abutaha Copyright

Version

gseqch_v1_

Date

Functions file concerning the map.

C.4.2 Macro Definition Documentation

C.4.2.1 BIT

```
BIT (
    n,
    x ) ( ( x ) >> (n) ) & 1 )
```

Shift a value from n bits.

Parameters

<i>n</i>	The number of bits to shift.
<i>x</i>	The value to shift.

C.4.3 Function Documentation

C.4.3.1 generator()

```
uint64_t generator (
    key * K,
    int iter_sec,
    double ratio1,
    double ratio2,
    double ratio3,
    double ratio4,
    uint64_t m1,
    uint64_t m2 )
```

Create a chaotic sample.

Parameters

<i>*K</i>	a secret key.
<i>iter_sec</i>	loop counter
<i>ratio1</i>	calculated value depend on control parameter p.
<i>ratio2</i>	calculated value depend on control parameter p.
<i>ratio3calculated</i>	value depend on control parameter p.
<i>ratio4</i>	calculated value depend on control parameter p.
<i>m1</i>	equal POWER(K[0].N)

Parameters

<i>m2</i>	equal POWER(K[0].N-1)
-----------	-----------------------

Returns

Xresult

C.4.3.2 generator_no_perturbation()

```
uint64_t generator_no_perturbation (
    key * K,
    int iter_sec,
    double ratio1,
    double ratio2,
    double ratio3,
    double ratio4,
    uint64_t m1,
    uint64_t m2 )
```

C.4.3.3 iv_setup()

```
void iv_setup (
    FILE ** Fic_parametre,
    key * K )
```

C.4.3.4 LFSR()

```
uint64_t LFSR (
    int maxdegree,
    int G,
    uint64_t Q )
```

Create a linear shift in the register.

Parameters

<i>maxdegree</i>	is the maximal degree of the polynome choosen.
<i>G</i>	is the decimal value for the choosen polynome
<i>Q</i>	is the maximal value.

Returns

reg is the new register state.

C.4.3.5 maximal_degree_of_polynome()

```
int maximal_degree_of_polynome (
    uint64_t polynome )
```

C.4.3.6 PWLCmap()

```
uint64_t PWLCmap (
    uint64_t X_1,
    uint64_t P,
    int N,
    double ratio3,
    double ratio4,
    uint64_t m1,
    uint64_t m2 )
```

C.4.3.7 STmap()

```
uint64_t STmap (
    uint64_t X_1,
    int N,
    uint64_t P,
    double ratio1,
    double ratio2,
    uint64_t m1,
    uint64_t m2 )
```

C.4.3.8 write_bin()

```
void write_bin (
    uint64_t * V,
    int L,
    FILE * FIC,
    int N )
```

C.4.3.9 write_dec()

```
void write_dec (
    uint64_t * V,
    int L,
    FILE * FIC )
```

Writes in the file FIC the L elements of the sequence V.

Parameters

<i>V</i>	is a table wich contain all decimal results.
<i>L</i>	is the table's lenght
<i>FIC</i>	is the file were results will be writes.

C.5 cartes.h File Reference

```
#include "util.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/types.h>
#include <inttypes.h>
```

Macros

- #define **BIT**(n, x) (((x) >> (n)) & 1)
- #define **POWER**(n) (1LL << (n))
- #define **_ERROR** -1

Functions

- void **write_dec** (uint64_t *V, int L, FILE *FIC)
Writes in the file FIC the L elements of the sequence V.
- void **write_bin** (uint64_t *V, int L, FILE *FIC, int N)
- int **maximal_degree_of_polynome** (uint64_t polynome)
- uint64_t **STmap** (uint64_t X_1, int N, uint64_t P, double ratio1, double ratio2, uint64_t m1, uint64_t m2)
- uint64_t **PWLCmap** (uint64_t X_1, uint64_t P, int N, double ratio3, double ratio4, uint64_t m1, uint64_t m2)
- uint64_t **LFSR** (int maxdegree, int G, uint64_t Q)
Create a linear shift in the register.
- uint64_t **generator** (**key** *K, int iter_sec, double ratio1, double ratio2, double ratio3, double ratio4, uint64_t m1, uint64_t m2)
Create a chaotic sample.
- uint64_t **generator_no_perturbation** (**key** *K, int iter_sec, double ratio1, double ratio2, double ratio3, double ratio4, uint64_t m1, uint64_t m2)
- void **iv_setup** (FILE **Fic_parametre, **key** *K)

C.5.1 Macro Definition Documentation

C.5.1.1 _ERROR

```
#define _ERROR -1
```

C.5.1.2 BIT

```
#define BIT(
    n,
    x ) ( ( (x) >> (n) ) & 1 )
```

C.5.1.3 POWER

```
#define POWER(
    n ) ( 1LL << (n) )
```

C.5.2 Function Documentation

C.5.2.1 generator()

```
uint64_t generator (
    key * K,
    int iter_sec,
    double ratio1,
    double ratio2,
    double ratio3,
    double ratio4,
    uint64_t m1,
    uint64_t m2 )
```

Create a chaotic sample.

Parameters

<i>*K</i>	a secret key.
<i>iter_sec</i>	loop counter
<i>ratio1</i>	calculated value depend on control parameter p.
<i>ratio2</i>	calculated value depend on control parameter p.
<i>ratio3calculated</i>	value depend on control parameter p.
<i>ratio4</i>	calculated value depend on control parameter p.
<i>m1</i>	equal POWER(K[0].N)
<i>m2</i>	equal POWER(K[0].N-1)

Returns

Xresult

C.5.2.2 generator_no_perturbation()

```
uint64_t generator_no_perturbation (
    key * K,
    int iter_sec,
    double ratio1,
    double ratio2,
    double ratio3,
    double ratio4,
    uint64_t m1,
    uint64_t m2 )
```

C.5.2.3 iv_setup()

```
void iv_setup (
    FILE ** Fic_parametre,
    key * K )
```

C.5.2.4 LFSR()

```
uint64_t LFSR (
    int maxdegree,
    int G,
    uint64_t Q )
```

Create a linear shift in the register.

Parameters

<i>maxdegree</i>	is the maximal degree of the polynome choosen.
<i>G</i>	is the decimal value for the choosen polynome
<i>Q</i>	is the maximal value.

Returns

reg is the new register state.

C.5.2.5 maximal_degree_of_polynome()

```
int maximal_degree_of_polynome (
    uint64_t polynome )
```

C.5.2.6 PWLCmap()

```
uint64_t PWLCmap (
    uint64_t X_1,
    uint64_t P,
    int N,
    double ratio3,
    double ratio4,
    uint64_t m1,
    uint64_t m2 )
```

C.5.2.7 STmap()

```
uint64_t STmap (
    uint64_t X_1,
    int N,
    uint64_t P,
    double ratio1,
    double ratio2,
```

```
uint64_t m1,
uint64_t m2 )
```

C.5.2.8 write_bin()

```
void write_bin (
    uint64_t * V,
    int L,
    FILE * FIC,
    int N )
```

C.5.2.9 write_dec()

```
void write_dec (
    uint64_t * V,
    int L,
    FILE * FIC )
```

Writes in the file FIC the L elements of the sequence V.

Parameters

<i>V</i>	is a table wich contain all decimal results.
<i>L</i>	is the table's lenght
<i>FIC</i>	is the file were results will be writes.

C.6 main.c File Reference

Main program. version v3 Stream: created by Mohammad Abutaha

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include "cartes.h"
#include <stdint.h>
#include <assert.h>
#include <unistd.h>
#include <string.h>
#include <sys/mman.h>
#include "cv.h"
#include "highgui.h"
```

Functions

— int **main** ()
start main program.

C.6.1 Detailed Description

Main program. version v1: created by Mohammad Abutaha.

Version

Stream cipher_v3

Date

Main program for the chaotic sequence generator using the basic map.

C.6.2 Function Documentation

C.6.2.1 main()

```
int main (
    void )
start main program.
generator calling
encryption
```

Returns

EXIT_SUCCESS.

C.7 util.c File Reference

```
#include "util.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <stdint.h>
#include <assert.h>
```

Functions

- **key InitializeStructureToZero ()**
- **uint64_t get_numOFsample (int N)**
Let the user to choose the number of sample in each sequences.
- **key InitializeStruct (FILE **Fic_parametre)**
- **key filling_structure (key *K, FILE **Fic_keys, int j)**
- **void secure_memzero (void *K, size_t len)**

C.7.1 Detailed Description

Version

stream _v3_

Date

25 septemper 2016

C.7.2 Function Documentation

C.7.2.1 filling_structure()

```
key filling_structure (
    key * K,
    FILE ** Fic_keys,
    int j )
```

C.7.2.2 get_numOFsample()

```
int get_numOFsample (
    int N )
```

Let the user to choose the number of sample in each sequences.

Parameters

N	The number of bits.
-----	---------------------

Returns

The number of samples in each sequences: numofsample.

C.7.2.3 InitializeStruct()

```
key InitializeStruct (
    FILE ** Fic_parametre )
```

C.7.2.4 InitializeStructureToZero()

```
key InitializeStructureToZero ( )
```

C.7.2.5 secure_memzero()

```
void secure_memzero (
    void * K,
    size_t len )
```

C.8 util.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <stdint.h>
#include <sys/types.h>
#include <inttypes.h>
```

Classes

- struct **key**
parameters of key.

Typedefs

- typedef struct **key** **key**

Functions

- uint64_t **get_numOFsample** (int N)
Let the user to choose the number of sample in each sequences.
- **key InitializeStructureToZero** ()
- **key InitializeStruct** (FILE **fic)
- **key filling_structure** (**key** *K, FILE **fic, int j)

C.8.1 Typedef Documentation

C.8.1.1 key

```
typedef struct key key
```

C.8.2 Function Documentation

C.8.2.1 filling_structure()

```
key filling_structure (
    key * K,
    FILE ** fic,
    int j )
```

C.8.2.2 get_numOFsample()

```
uint64_t get_numOFsample (
    int N )
```

Let the user to choose the number of sample in each sequences.

Parameters

<i>N</i>	The number of bits.
----------	---------------------

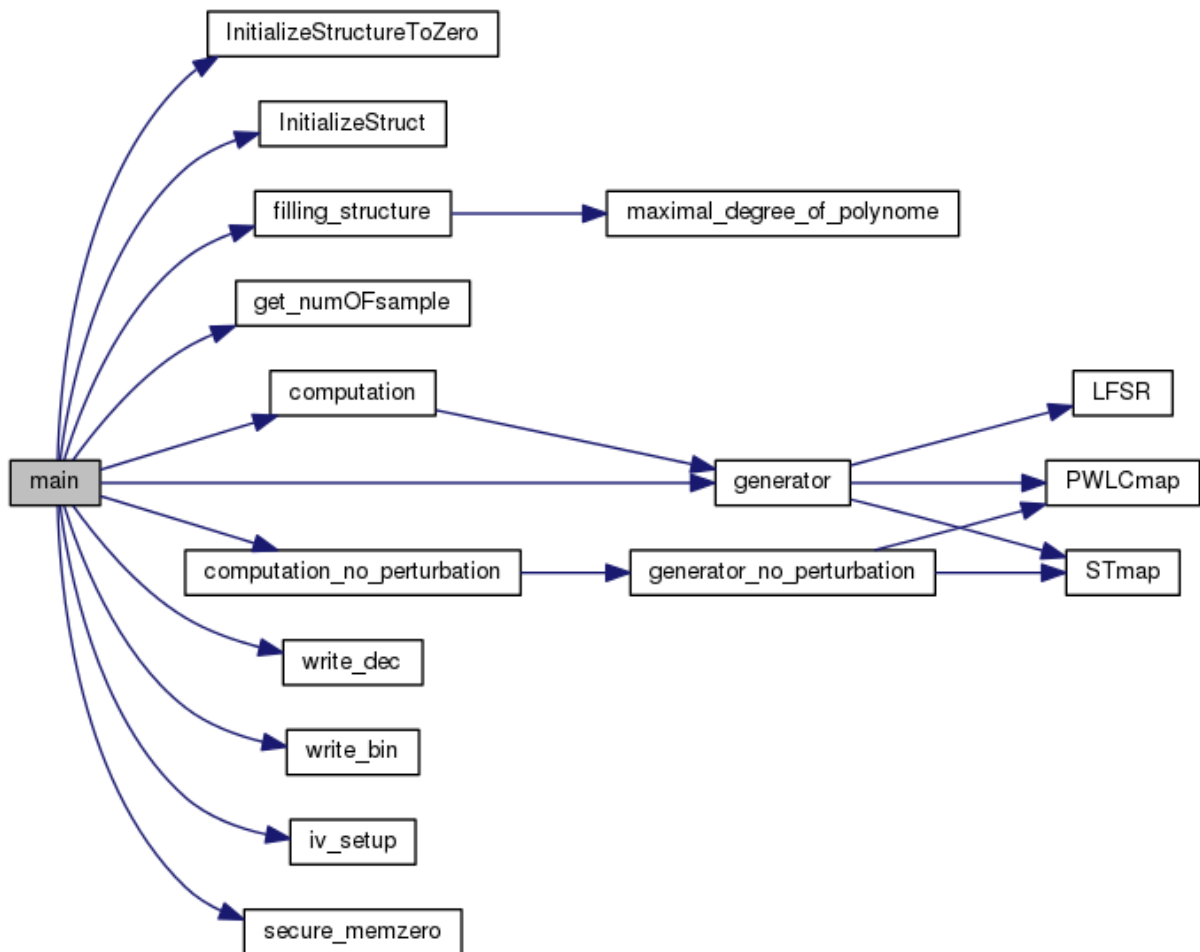


Figure C.1 – Code scheme parallel version.

Returns

The number of samples in each sequences: numofsample.

C.8.2.3 InitializeStruct()

```

key InitializeStruct (
    FILE ** fic )
  
```

C.8.2.4 InitializeStructureToZero()

```

key InitializeStructureToZero ( )
  
```


Thèse de Doctorat

Mohammed ABUTAHA

Systèmes de Crypto-Compression basés Chaos en Temps Réel et Portables pour Architectures embarquées Efficaces

Real-Time and Portable Chaos-based Crypto-Compression Systems for Efficient Embedded Architectures

Résumé

La protection des images et vidéos est une problématique cruciale. Dans ce travail nous avons d'abord, conçu et réalisé d'une façon efficace et sécurisée un générateur de nombre pseudo-chaotique (PCNG) mis en œuvre en séquentielle et en parallèle par P-threads. Basé sur ces PCNGs, deux applications centrales ont été conçues, mises en œuvre et analysées. La première traite la réalisation d'un générateur de nombre aléatoire et les résultats obtenus sont très prometteurs. La deuxième concerne la réalisation d'un système de chiffrement/déchiffrement par flux. L'analyse cryptographique des systèmes chaotiques réalisés montrent leur robustesse contre des attaques connues. Ce résultat est dû à la structure récursive proposée qui intègre une forte non-linéarité, une technique de perturbation et un multiplexage chaotique. La performance obtenue en complexité de calcul indique leurs utilisations dans des applications temps réel. Ensuite, basé sur le système chaotique précédent, nous avons conçu et mis en œuvre efficacement un système de crypto-compression pour des applications temps réel et portable pour architectures embarquées. Une solution de chiffrement par flux sélectif des contenus vidéo HEVC est réalisée. Puis, un chiffrement d'une région d'intérêt est effectué au niveau CABAC pour les paramètres les plus sensibles incluant des vecteurs de mouvement et des coefficients transformés. Le format le chiffrement conforme de Modes de Prédiction Intra a été aussi vérifié. L'évaluation subjective et des tests de complexité d'altération de taux objectifs ont montré que la solution proposée sécurise le contenu vidéo avec un débit binaire et une complexité de codage légèrement augmentés.

Mots clés

Générateur de nombres pseudo-chaotiques, RNG, Chiffrement par flux basé chaos, Systèmes de Crypto-Compression basés Chaos, Chiffrement sélectif, HEVC, Analyse de la sécurité, Complexité de calcul.

Abstract

Image and video protection have gained a lot of momentum over the last decades. In this work, first we designed and realized in an efficient and secure way a pseudo-chaotic number generator (PCNG) implemented in sequential and parallel (with P-threads) versions. Based on these PCNGs, two central applications were designed, implemented and analyzed. The former application deals with the realization of a random number generator (RNG) based PCNG, and the obtained results are very promising. The latter application concerns the realization of a chaos-based stream cipher. The cryptographic analysis and the statistical study of the realized chaotic systems show their robustness against known attacks. This result is due to the proposed recursive architecture which has a strong non-linearity a technique of disturbance, and a chaotic multiplexing. The computation performance indicate their use in real time applications. Second, based on the previous chaotic system, we designed and implemented in effective manner a real time joint crypto-compression system for embedded architecture. An end-to-end selective encryption solution that protects privacy in the HEVC video content is realized. Then, a ROI encryption is performed at the CABAC bin string level for the most sensitive HEVC parameters including motion vectors and transform coefficients. The format compliant encryption of Intra Prediction Modes has been also investigated. It increases a little bit the bit rate. Subjective evaluation and objective rate-distortion-complexity tests showed that the proposed solution performs a protection of privacy in the HEVC video content with a small overhead in bit rate and coding complexity.

Key Words

Pseudo-chaotic number generator, RNG, Chaos-based stream cipher, Chaos-based crypto-compression system, Selective encryption, HEVC, Security analysis, Computing performance.