KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT ELEKTROTECHNIEK–ESAT
Kasteelpark Arenberg 10, 3001 Leuven-Heverlee

Cryptanalysis of Stream Ciphers Based on Arrays and Modular Addition

Promotor:
Prof. Dr. ir. Bart Preneel

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Souradyuti Paul

November 2006

KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT ELEKTROTECHNIEK–ESAT
Kasteelpark Arenberg 10, 3001 Leuven-Heverlee

Cryptanalysis of Stream Ciphers Based on Arrays and Modular Addition

Jury:
Prof. Dr. ir. Etienne Aernoudt, voorzitter
Prof. Dr. ir. Bart Preneel, promotor
Prof. Dr. ir. André Barbé
Prof. Dr. ir. Marc Van Barel
Prof. Dr. ir. Joos Vandewalle
Prof. Dr. Lars Knudsen (Technical University, Denmark)

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Souradyuti Paul

U.D.C. 681.3*D46

November 2006

D/2006/7515/88

ISBN 978-90-5682-754-0

To

my parents for their unyielding ambition to see me educated

and

Prof. Bimal Roy for making cryptology possible in my life ...

My Gratitude

It feels awkward to claim the thesis to be singularly mine as a great number of people, directly or indirectly, participated in the process to make it see the light of day. At the end of such a long and laborious process, it is, therefore, a great pleasure for me to grab this opportunity to express my gratefulness.

First, I would like to express my sincere gratitude to my thesis supervisor Prof. Bart Preneel who not only guided me with technical help through this gestation period as a doctoral student, but also, most importantly, has shown me very convincingly the qualities, which an academician should acquire in order to reach the highest level of success. I strongly believe that I would continue in my pursuit to reach the standards he has set before me in those four and a half years of academic collaboration. I am also extremely indebted to him for putting up with all my nonsensical, silly questions, my emails fired at him at odd hours – be it academic or non-academic – with supreme patience and thereby, instilling in me a strong sense of self-confidence – something which I would always treasure. All in all, in hindsight, I feel that whatever little maturity that I have acquired since the time of being an all nervous student around five years ago, is pretty much due to the academic venture I was engaged with him in.

I am honored to have Prof. André Barbé, Prof. Lars Knudsen, Prof. Marc Van Barel and Prof. Joos Vandewalle as the members of the jury and Prof. Etienne Aernoudt as the chairman of the jury.

During the doctoral period, I was extremely lucky to have the opportunity to supervise the theses of Gorka Munduate and Gautham Sekar. My own academic benefit from the joint work with them was immense.

In the past years, I came in contact with many cryptographers in diverse fields for their comments, remarks, opinions, suggestions on various topics. The list includes Eli Biham, Paul Crowley, Scott Fluhrer, Ilya Mironov, Kenneth G. Paterson, Palash Sarkar and Adi Shamir. I gratefully acknowledge their constructive comments.

My expression of gratitude will forever remain incomplete if I do not mention the name of Dai Watanabe who was the first to have made me appreciate the subtleties of symmetric cryptography. I am also thankful to all the past and present COSIC members for lending their helping hands in times of need; in particular, I would like to mention Alex, An, Antoon, Christophe, Christopher, Elena, Danny, Frederik, Gregory, Hirotaka, Hongjun, Jasper, Joe, Jongsung, Jorge, Michael, Nessim, Özğul, Robert, Stefaan, Taizo and Thomas.

Thanks are also due to Avishek Adhikari, Debrup Chakraborty, Kishan Chand Gupta, Matthew E. McKague, Alexander Maximov, Partha Mukhopadhyay, Sourav Mukhopadhyay, Mridul Nandi, Anirban Pal, Sankardas Roy and Sabyasachi Saha for many useful discussions on various branches of computer science including computer security.

Many thanks to Pela, Elvira and Ida for their priceless help with administrative matters.

Finally, I express my thankfulness to my parents and my brother for their constant encouragement and support without which the thesis would never have been possible.

Souradyuti Paul,
Leuven, November 2006.

Abstract

In modern cryptography, stream ciphers are most useful in applications where information needs to be encrypted/decrypted at high speed (e.g. high resolution streaming video data) or when low footprint (gates/memory) encryption is required. In the literature, there exist plenty of stream ciphers whose internal states are based on arrays and that they use modular additions to generate output streams. The abundance of array-based stream ciphers with modular additions can be attributed to the fact that, when implemented in software skillfully, they are able to produce outputs at a very high speed. The main contribution of this thesis is a unified analysis of stream ciphers based on arrays and modular addition. During the process, we detect cryptographic weaknesses in the designs of 9 widely known stream ciphers or pseudorandom bit generators (PRBGs).

At first, we show some theoretical results on solving an important class of equations known as *differential equations of addition* (DEA) that combine modular additions over two different algebraic groups such as GF(2) and GF(2^{32}). The results include,

- proof of the fact that the satisfiability of an arbitrary set of DEA is in the complexity class $\mathcal{P}$,
- deriving all the solutions of an arbitrary set of DEA.

Next, we apply these results to attack a practical stream cipher named Helix (designed by Ferguson *et al.*) with both chosen plaintexts and adaptive chosen plaintexts.

In the second phase, the thesis closely scrutinizes a number of array-based stream ciphers (or PRBGs) in order to estimate their resistance against distinguishing attacks. We eventually discover, counter-intuitively, that the correlations between the array-indices and their associated array-elements, which apparently seem to be useful from the point of view of implementation purposes, can be exploited to mount distinguishing attacks on such type of ciphers if adequate precautions are not taken. In support of our theoretical findings, we point out distinguishing attacks on 8 practical array-based stream ciphers (or PRBGs),

namely RC4 (designed by Rivest), RC4A (designed by Paul and Preneel), Py, Py6 (designed by Biham and Seberry), IA, ISAAC (designed by Jenkins Jr.), GGHN, NGG (by Gong *et al.*); our attacks are based on the dependence of array-elements on array-indices. In all the cases we work under the assumption that the key-setup algorithms of the ciphers produce uniformly distributed internal states. We detect flaws in the mixing of bits in the keystream generation algorithms. Our analysis can be explained as the extension, development, adaptation and deeper characterization of the *fortuitous states attacks* on the RC4 cipher by Fluhrer and McGrew in 2000.

Samenvatting

In de moderne cryptografie bewijzen stroomcijfers vooral hun nut in toepassingen waar informatie op hoge snelheid ver- en ontsleuteld moet worden (bv. hoge-resolutie videosignalen), of wanneer de beschikbare hardware onderworpen is aan strenge beperkingen (maximaal aantal gates, geheugen, etc.). In de literatuur bestaan er tal van stroomcijfers waarvan de interne toestand bestaat uit arrays, en die gebruik maken van modulaire optelling om de uitgangsstroom te genereren. Het zeer grote aantal array-gebaseerde stroomcijfers met modulaire optelling kan toegeschreven worden aan het feit dat het gebruik van deze componenten in software kan leiden tot zeer snelle implementaties. De voornaamste bijdrage van deze thesis is een unificatie van de analyse van stroomcijfers die gebruik maken van arrays en modulaire optelling. Gedurende dit proces werden cryptografische zwakheden ontdekt in 9 wijdverspreide stroomcijfers en pseudo-willekeurige bit generatoren (PRBGs).

Eerst geven we een aantal theoretische resultaten die toelaten om een belangrijke klasse van vergelijkingen op te lossen. Deze vergelijkingen, gekend onder de naam *differentiële optellingsvergelijkingen* (DEA) combineren modulaire optellingen over twee verschillende algebraïsche groepen zoals GF(2) en GF(2^{32}). De resultaten omvatten:

- een bewijs dat de vraag naar oplosbaarheid voor een willekeurige verzameling DEA gelegen is in de complexiteitsklasse $\mathcal{P}$,
- een methode voor het afleiden van alle oplossingen van een willekeurige verzameling DEA.

Vervolgens passen we deze resultaten toe op het concrete stroomcijfer Helix (ontwikkeld door Ferguson *et al.*), en voeren we een aanval uit gebruik makend van zowel gekozen als adaptief gekozen klaarteksten.

In een tweede fase spitst de thesis zich toe op een aantal array-gebaseerde stroomcijfers, met als doel hun weerstand tegen onderscheidingsaanvallen in kaart te brengen. Uiteindelijk ontdekken we, ietwat tegen-intuïtief, dat de correlaties tussen de array-indices en hun overeenkomstige array-elementen, hoewel nuttig

voor implementatie-doeleinden, anderzijds ook uitgebuit kunnen worden om onderscheidingsaanvallen op te zetten, indien geen adequate voorzorgsmaatregelen genomen werden. Als bevestiging van onze theoretische conclusies, tonen we onderscheidingsaanvallen aan op 8 concrete array-gebaseerde stroomcijfers (of PRBGs): RC4 (ontworpen door Rivest), RC4A (door Paul en Preneel), Py, Py6 (ontwikkeld door Biham en Seberry), IA, ISAAC (ontworpen door Jenkins Jr.), GGHN, NGG (door Gong *et al.*). Al deze aanvallen berusten op de afhankelijkheid van array-elementen en array-indices, en gaan uit van de veronderstelling dat de sleutel-initialisatie procedures van deze cijfers resulteren in een uniform verdeelde initiële toestand. We ontdekken zwakheden in het mixen van bits in de sleutelstroomgeneratie. Onze analyse kan gezien worden als een uitbreiding, aanpassing, en verdere karakterisering van de *fortuitous states* aanvallen op RC4 ontwikkeld door Fluhrer en McGrew in 2000.

Contents

List of Figures

List of Tables

List of Symbols

$\lll$	cyclic left shift
$\ll$	left shift
$\ggg$	cyclic right shift
$\gg$	right shift
$\Rightarrow$	implies
$\Leftrightarrow$	implies and implied by
$\oplus$	bitwise *exclusive-or* of two n-bit integers
$\wedge$	bitwise *product* of two n-bit integers
$\vee$	bitwise *or* of two n-bit integer
$\odot$	multiplication of two n-bit integers modulo 2^n
$+$	addition of two n-bit integers modulo 2^n
$-$	subtraction of two n-bit integers modulo 2^n
ab	$a \wedge b$
$\mathrm{GF}(q)$	finite field with q elements
$l_{(i)}$	the ith bit of an n-bit integer l ($l_{(0)}$ is the lsb)
n	a positive integer
$\mathcal{O}$	order of complexity denoting asymptotic upper bound
Θ	order of complexity denoting asymptotic tight bound
A^c	complement of the event A
$[p,\, q]$	the set of integers $\{p, p+1, \ldots, q\}$
$P[A]$	probability of occurrence of the event A
$P[A\|B]$	conditional probability of A given B
$P_D[\mathcal{F}(z) = c]$	probability of $\mathcal{F}(z) = c$ where z follows distribution D
$\|S\|$	the cardinality of the set S
$X_{(m,n)}$	the segment of $m - n + 1$ bits between the mth and the nth bits of the variable X
$\mathbb{Z}_q$	the set $\{0, 1, 2, \ldots, q-1\}$
$\mathbb{Z}_q^p$	p-fold Cartesian product of the set $\mathbb{Z}_q$, i.e., $\underbrace{\mathbb{Z}_q \times \mathbb{Z}_q \times \cdots \mathbb{Z}_q}_{p \text{ times}}$

List of Abbreviations

ACP	adaptive chosen plaintext
AES	Advanced Encryption Standard
CP	chosen plaintext
DC	differential cryptanalysis
DEA	differential equation(s) of addition
DES	Data Encryption Standard
DTM	deterministic Turing machine
ECRYPT	European network of excellence for cryptology
eSTREAM	ECRYPT stream cipher project
IV	initialization vector
KSA	key scheduling/setup algorithm
lsb	least significant bit
MAC	message authentication code
msb	most significant bit
NESSIE	New European Schemes for Signatures, Integrity and Encryption
$\mathcal{NP}$	class of problems solvable in polynomial time by an NTM
$\mathcal{NPC}$	class containing the hardest problems in $\mathcal{NP}$
NTM	non-deterministic Turing machine
$\mathcal{P}$	class of problems solvable in polynomial time by a DTM
PRBG	pseudorandom bit generator
RAM	random access machine
RSA	Rivest-Shamir-Adleman
S-box	substitution box (or vectorial Boolean function)
SSL	secure sockets layer
TLS	transport layer security/secure sockets layer
WEP	wired equivalent privacy
XOR	bitwise exclusive-or

Chapter 1

Introduction

To begin is half the work..
– Ausonius (310-395)

Cryptology is the science of hiding information from unauthorized users, and making them available to the legitimate ones. The word *cryptology* is derived from two Greek words: *kryptós* which means 'hidden' and *logos* meaning 'word'. Historically, the art of secret writing is as old as the time of Pharaos of ancient Egypt. Julius Caesar (100 BC – 44 BC) is known to have had used some specific techniques, known as *Caesar cipher*, to protect messages of high military worth. In fact, the emergence of classical cryptology as a well structured scientific discipline from a mere black art of middle ages has been greatly influenced by the growing military needs of modern times to disguise sensitive information from the enemy. It is still debated whether the breaking of the German secrecy system *Enigma* by the Allied cryptologists (one of them was the famous British mathematician Alan Turing who is deemed the father of modern computer science) had shortened the duration of the World War II significantly. Interested readers are highly recommended to look at the classical book *The Codebreakers* by David Kahn that traces the evolution of cryptology since the dawn of civilization till the age of the modern computer science [43].

Modern cryptology is all about protecting sensitive information on the Internet from several forms of abuses. As a result the scope of cryptology is no longer limited to just military applications. As *Internet banking*, *Electronic money transfer* (e.g. payment through credit cards), *Internet Shopping* (e.g. ordering books on the amazon.com) are being widely used all over the world nowadays, the subject of *cryptology* has evoked unprecedented interest and enthusiasm in both industry and academia.

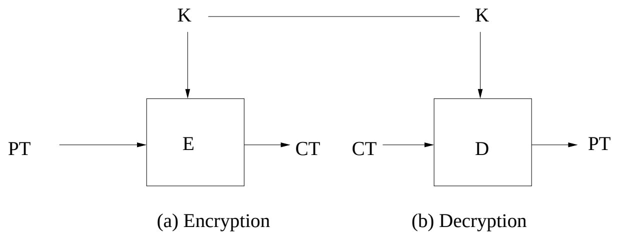

Figure 1.1: Encryption and Decryption: PT=Plaintext, CT=Ciphertext, K= Secret Key, E=Encryption Algorithm, D=Decryption Algorithm

Encryption and Decryption Algorithms. At the heart of cryptology, there are two mathematical functions E and D which are used for *encryption* and *decryption* respectively (schematic diagram in Fig. 1.1). Another component, known as the *secret key* K, also plays an important role in cryptologic applications. The functions E and D are publicly known but the *secret key* is known only to the sender and the receiver of the message. Let us take an example. Suppose Alice wants to send a confidential email to Bob through an unprotected path which can be accessed by all. In other words, the email sent by Alice to Bob can be captured midway by a third party whose name is, say Oscar. Alice (the sender) will encrypt the email (technically called message) using the function E and a secret key K. Thus, if the message is PT then Alice generates the encrypted code $E(K, PT) = CT$. The encrypted code CT is then sent out on an insecure channel. At the receiving end Bob gets CT and decrypts it to recover the original message using the function D and the shared secret key K, i.e., by performing the operation $D(K, CT) = PT$. When Oscar captures the encrypted message on the way, he tries to recover PT from CT without the knowledge of the secret key K. The system fails if Oscar succeeds. The system is considered secure if the recovery of the original message from the encrypted message is 'impossible' without the knowledge of the secret key.

The main research in cryptology centers around designing the mathematical functions E and D such that the communication is secure in a manner argued in the previous paragraph. An unconditionally secure cryptosystem is the classical *Vernam Cipher* or *one-time pad*. The encryption and the decryption functions of the *Vernam cipher* are as follows (more on the perfect security of the *Vernam cipher* in [86]).

$$\begin{aligned} \text{Encryption: } CT &= PT \oplus K, \\ \text{Decryption: } PT &= CT \oplus K. \end{aligned}$$

Unfortunately, the *Vernam cipher* requires the key to be of the same size as that of the plaintext. This condition renders this cipher completely impractical as, in most applications, long messages are required to be encrypted. In fact, no practical cryptosystem has so far been proved to be secure. Scores of cryptosystems are proposed at the conferences and in the journals every day and they are broken, sometimes more quickly than they are designed. This alternate making and breaking of secrecy systems, unlike any other scientific research, keep the practitioners of this subject always on an adrenalin rush.

A Few Notable Encryption/Decryption Algorithms. One of the most widely used cryptographic algorithms, known as the DES Cryptosystem (abbreviation for Data Encryption Standard) designed by a team of IBM and selected as an official Federal Information Processing Standard (FIPS) for the United States in 1976, was once thought to be secure for a long time. DES has a block size of 64 bits and a key size of 56 bits. However, with the emergence of faster machines, the 56-bit key length of DES turned out to be too short to be secure against brute force attack. In addition, new sophisticated mathematical techniques were also developed to even to improve the brute force attack. Those methods are known as (1) *Differential Cryptanalysis*, publicly invented by Biham, Shamir in 1992 [9], and (2) *Linear Cryptanalysis* discovered by Matsui [54] in 1993. After its failure, the DES Cryptosystem is being gradually replaced by a new standard called the AES Encryption Algorithm (Advanced Encryption Standard), also known as Rijndael named after its two Belgian designers Rijmen and Daemen of the Katholieke Universiteit Leuven [19]. The block size of AES is 128 bits. There are three possible key sizes for AES: 128, 192 and 256 bits. Till now, barring a few occasional rumors, AES-Rijndael has survived all attacks and is considered to be exceptionally strong.

Invention of Public Key Cryptology. As explained before, one way of securing cryptographic systems is to make the sender and the receiver share the same secret key K. However, it is often difficult for the receiver and the sender, located wide apart on the globe, to share the same secret key because the transfer of the key over the Internet makes it prone to capture by a third party. Therefore, the question that arises naturally is: how can one design the functions E and D (as explained before) securely without the shared component K? Diffie and Hellman in 1976, initiated a new direction in cryptography by inventing a Public-key encryption algorithm for which it was no more necessary to share a key in secret communication [22]. Since the publication of this landmark paper, the problem of designing a practical and secure cryptosystem without the condition of sharing of a key, became a hot pursuit by the mathematicians. Three MIT scientists Ronald Rivest, Adi Shamir and Leonard Adleman, in 1978, first gave a practical algorithm – RSA Encryption Algorithm – for a Public-Key Cryptosystem which is based on the hardness of a well known number theoretic problem of factoring large integers [78]. Discussion of how this type of cryptosystems works is out of the scope of this thesis, however, the underlying mathematical problem, which is the lifeblood of this type of secrecy system, can be described in a few words. If $l = mn$ where m and n are large prime numbers then, given l, it is hard to determine the prime factors m and n quickly. Another very popular public key encryption system is the El Gamal cryptosystem which is based on the hardness of the discrete logarithm problem. The discrete logarithm problem is: given two elements g and h in a finite group G, find an integer x such that $g^x = h$. For example, the solution to the problem $3^x = 10 \pmod{17}$ is 3, because $3^3 = 27 = 10 \pmod{17}$. This problem is assumed to be hard when considered over some groups of large orders. Discrete logarithm problem, when considered in a slightly complex setting such as an elliptic curve group, can be used to build another important public key cryptosystem, known as Elliptic Curve Cryptosystem (ECC). Many other algebraic hard problems such as *solving multivariate polynomial equations* are also under investigation for the purpose of information hiding [94]. An elaborate discussion of the above cryptosystems can be found in [56, 88].

1.1 The Scope of Cryptology

Today the scope of cryptology is not limited to data *encryption* and *decryption* only as mentioned above. The boundary of the subject is in continuous process of expansion and modification. The main purposes of cryptology are threefold: (i) confidentiality of message, (ii) message authentication and (iii) entity authentication.

- Confidentiality
  - Alice wants to send a message to Bob through an insecure channel. Oscar, who captures the message midway, should not be able to learn any information about the contents.
  - Cryptographic tools or primitives used to protect confidentiality are encryption algorithms and decryption algorithms.
  - Examples: RC4 [76], AES [19].
- Message Authentication
  - This service detects unauthorized alteration of data.
  - Alice wants to send a message to Bob. Oscar, on capturing the message in transit, should not be able to alter the contents and get them

accepted by Bob as original. For example, let the message sent from Alice to Bob be *"India is a friend of Belgium."* The message during transmission should not become *"India is an enemy of Belgium."* and be received by Bob as original.

  - Cryptographic tools or primitives used to secure message authentication are hash functions (e.g. MD4 [74], MD5 [75], RIPEMD-160 [23], SHA-0 [83], SHA-1 [84], SHA-2 [85]), message authentication codes (MAC) [71] and digital signatures (e.g. DSS [24]).

- Entity Authentication
  - Alice is communicating with Bob. Alice (claimant) will be able to prove her identity to Bob (verifier) by demonstrating knowledge of a secret known to be associated with her, without revealing the secret itself to Bob.
  - Challenge-response protocols (e.g. Kerberos [62]), zero-knowledge protocols (e.g. Fiat-Shamir [28] protocol) prove the identity of one party to the other.

The reader is kindly referred to [56] and [88] for extensive discussions on various subareas in cryptology.

1.2 Symmetric Cryptology

According to the nature of the secret keys used in the encryption or decryption algorithms, cryptology can be divided into two broad classes. If both the encryption and decryption algorithms use the same shared key, then they come under symmetric cryptology. The encryption and decryption algorithms shown in Fig. 1.1 use the same secret key. Under symmetric cryptology, there are two ways in which information or data can be encrypted and decrypted: one is using a block cipher and the other is using a stream cipher. It is, however, worth noting that the distinctions between the functionalities of stream ciphers and block ciphers are not very sharp; they are partly quantitative and partly qualitative. These two types of ciphers should be viewed as two concrete points on a continuous design spectrum [82].

Stream Cipher. Stream cipher is an algorithm where

- both the encryption and the decryption functions use the shared secret key,
- the encryption/decryption function to encrypt/decrypt a unit of message (usually message of short length) varies with time,

- the processing of data is usually fast,
- key processing is completely separate from the plaintext processing,
- plaintext processing is remarkably simple.

Block Cipher. Block cipher is an algorithm where

- both the encryption and the decryption functions use the shared secret key,
- encryption/decryption function to encrypt/decrypt a unit of message (usually a large block of message) does not vary with time,
- key and plaintext are processed together in a relatively complex function.

Elaborate discussion on mathematical formalization of block ciphers and stream ciphers can be found in [56] and [88].

As most of the thesis deals with cryptanalysis of various stream ciphers, a brief discussion on different types of stream ciphers sounds justifiable.

1.3 Stream Ciphers

1.3.1 Mathematical Formulation

Typically a stream cipher consists of two phases: (i) a key setup algorithm (KSA) and (ii) a pseudorandom bit generation algorithm (PRBG). The KSAs on both the encryption and decryption sides are identical. However, the PRBG on the encryption side is slightly different from the one on the decryption side.

(i) Key Setup Algorithm. The inputs to the KSA are a secret key K and a known initialization vector IV. The output from the KSA is the initial internal state of the cipher, denoted by S_0. Mathematically,

$$\mathrm{KSA}(K, \mathrm{IV}) = S_0. \tag{1.1}$$

(ii) Pseudorandom Bit Generation Algorithm. In this phase random looking stream of bits are generated sequentially. At any round t of the PRBG, the memory of a stream cipher consists of (1) an internal state S_t, (2) the key K and (3) the initialization vector IV. Henceforth, the K/IV pair will be denoted by K_c. It is also possible that the K_c does not exist as a separate entity at some round t; rather, after going through a transformation it becomes a part of the internal state S_t. Typically, at a round t, a stream cipher produces output Z_t which is XORed with the plaintext P_t to produce the ciphertext C_t.

In general, three functions are executed at any round of the PRBG. The operations on the encryption side is as follows:

$$\begin{cases} Z_t = g(S_t, K_c), \\ C_t = Z_t \oplus P_t, \\ S_{t+1} = f(S_t, K_c, X). \end{cases} \tag{1.2}$$

Similarly, the operations on the decryption side is:

$$\begin{cases} Z_t = g(S_t, K_c), \\ P_t = Z_t \oplus C_t, \\ S_{t+1} = f(S_t, K_c, X). \end{cases} \tag{1.3}$$

The variable X in the above equations is either P_t or C_t. The value of S_0 can be computed from K_c.

Asymptotic Definition of a PRBG. Theoretically, the behavior of a cryptographically strong pseudorandom bit generator (CSPRBG) is analyzed asymptotically in terms of the length of *key* (also known as *seed*). Note that, for most of the practical stream ciphers, such as RC4, Helix, the key lengths are fixed. From the theoretical point of view, a CSPRBG is an algorithm $\mathcal{A}$ that, on being given a random seed k as input, generates a sequence of pseudo-random bits a_1, a_2, $a_3, \cdots$. The function $\mathcal{A}$ possesses the following properties (see Blum and Micali [11]):

1. Each bit a_i can be produced in time polynomial in the length of seed k.

2. Given the algorithm $\mathcal{A}$ and the first s output bits generated by an unknown seed k, it is *computationally infeasible* to predict the $s+1$st bit with biased probability. The string s is polynomial in the length of the seed k.

1.3.2 Classification of Stream Ciphers

Depending on the usage of different variables involved in the functions f and g of the PRBG, as shown in (1.2) and (1.3), stream ciphers are broadly classified into three categories: (1) synchronous, (2) asynchronous and (3) self-synchronous.

Synchronous Stream Cipher

A *synchronous stream cipher* is one which generates Z_t (known as keystream) independently of the plaintext and the ciphertext. Therefore, the encryption function for this case becomes

$$\begin{cases} Z_t = g(S_t, K_c), \\ C_t = Z_t \oplus P_t, \\ S_{t+1} = f(S_t, K_c). \end{cases} \tag{1.4}$$

Similarly, the decryption function is

$$\begin{cases} Z_t = g(S_t, K_c), \\ P_t = Z_t \oplus C_t, \\ S_{t+1} = f(S_t, K_c). \end{cases} \tag{1.5}$$

Example: Py [6]. Working principles of a synchronous stream cipher are shown in Fig. 1.2. Some interesting properties of a synchronous stream cipher are worth

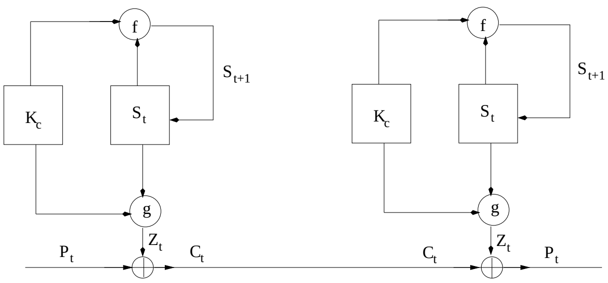

Figure 1.2: Encryption and decryption mechanisms of a synchronous stream cipher

noting.

- The modes of operation of a synchronous stream cipher *only* allows for *known plaintext* or *known ciphertext* attacks as the keystream is produced independently of the *plaintext* or the *ciphertext*. *Chosen plaintext* or *chosen ciphertext* attacks are not possible here.

- If some bits are deleted from or inserted into the ciphertext accidentally during message transmission, it is imperative to restart transmission with a new pair of key/IV to establish synchronization at both the encryption and the decryption ends. This process is known as *resynchronization mechanism*. Evidently, *resynchronization* is quite expensive in this type of ciphers.

- If some ciphertext bits are accidentally flipped during message transmission then the error does not affect the decryption of other ciphertext bits. This is a positive aspect of a synchronous stream cipher.

- A secure synchronous stream cipher is a suitable candidate to be used as a PRBG.

Asynchronous Stream Cipher

An *asynchronous stream cipher* is one where the generated keystream depends on the plaintext or the ciphertext. Example: Helix [33]. Working principles of an asynchronous stream cipher are shown in Fig. 1.3. The encryption function for the cipher in Fig. 1.3 is:

$$\begin{cases} Z_t = g(S_t, K_c), \\ C_t = Z_t \oplus P_t, \\ S_{t+1} = f(S_t, K_c, P_t). \end{cases} \tag{1.6}$$

Similarly, the decryption function is:

$$\begin{cases} Z_t = g(S_t, K_c), \\ P_t = Z_t \oplus C_t, \\ S_{t+1} = f(S_t, K_c, P_t). \end{cases} \tag{1.7}$$

Two important features of this class of ciphers are as follows.

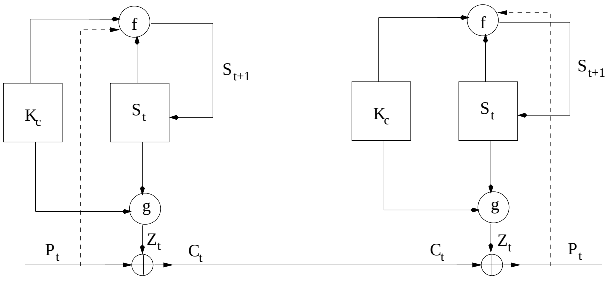

Figure 1.3: Encryption and decryption processes of an asynchronous stream cipher

- In general, the main disadvantage of this type of ciphers is that, if few bits of the ciphertext are altered then the error propagates through other ciphertext bits too.
- This type of ciphers admit of both *chosen plaintext/chosen ciphertext* and *known plaintext/known ciphertext* attacks unlike a *synchronous* stream cipher which only permits *known plaintext/known ciphertext* attacks.

Self-synchronizing Stream Cipher

Self-synchronizing stream ciphers are a special case of *asynchronous stream ciphers*. A *self-synchronizing* stream cipher is one where the keystream Z_t is completely determined by the key/IV pair and a *fixed* of number preceding ciphertext bits. Example: MOSQUITO [42]. Working principles of a self-synchronizing stream cipher are shown in Fig. 1.4.

The advantages of a *self-synchronizing* stream cipher over a *synchronous stream*

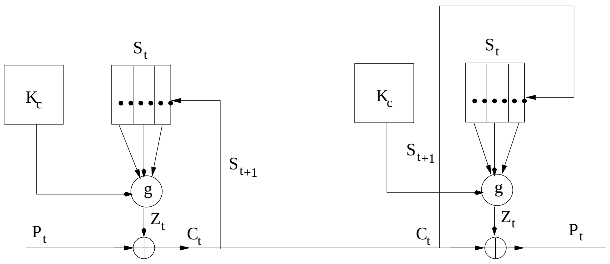

Figure 1.4: Encryption and decryption in a self-synchronizing stream cipher

cipher are listed below.

- As the internal state depends *only* on a *fixed* number of preceding ciphertext bits, the state can be fully recovered even after alteration, modification, deletion or insertion of some ciphertext words during transmission.
- For the same reason as above, error in some ciphertext bits only affects the decryption process for a limited number of subsequent bits.

However advantages the ciphers may have over others, design of a secure *self-synchronizing* stream cipher has so far turned out to be a difficult task. This fact has been reflected in the weaknesses discovered in all of this type of ciphers (SSS, MOSQUITO) of the ongoing ECRYPT eSTREAM project [18, 42].

1.3.3 Different Types of Attacks on Stream Ciphers

Broadly a stream cipher is analyzed against three types of attacks: (i) key recovery attack, (ii) distinguishing attack and (iii) related key attack.

(i) Key Recovery Attack. From the above discussion it is clear that the inputs to the publicly known encryption and decryption algorithms of a stream cipher are some or all of the following parameters,

- secret key,
- an initialization vector (IV) (explained before),
- plaintext,
- ciphertext.

Note that except the secret key an adversary can select all the other parameters suitably to attack the cipher. Therefore, the ultimate target of the adversary is to derive the secret key by properly selecting the other parameters. Depending on the power of adversary, the modes of attacks on stream ciphers are listed below.

- **Ciphertext Only Attack.** This is the strongest form of attack. In this case the adversary recovers the key from the ciphertext only, no matter what the other parameters are.
- **Chosen Ciphertext Attack.** In this case the adversary chooses the ciphertext and observes the corresponding plaintext. From the ciphertext-plaintext pair the adversary finally recovers the key using the decryption algorithm. One variant of *chosen ciphertext attack* is the *adaptive chosen ciphertext attack.* In such type of attacks, the adversary collects a number of ciphertext-plaintext pairs where the next ciphertext is determined from the previous ciphertext-plaintext pairs.
- **Known Plaintext Attack.** As the title suggests, the key is recovered from the known plaintext and the corresponding ciphertext.
- **Chosen Plaintext Attack.** In this case the adversary chooses the plaintext and recovers the key using the plaintext-ciphertext combination. In *adaptive chosen plaintext* attack, the adversary collects a set of plaintext-ciphertext pairs where the next plaintext is computed from the previous plaintext-ciphertext pairs.
- **Known IV Attack.** In this mode of attack, only the IV and the ciphertext is known.
- **Chosen IV Attack.** In this case the adversary chooses the IV and recovers the key using the IV-ciphertext combination. In a similar way as described above, adaptive chosen IV attack is also possible.
- Combination of the above.

(ii) Distinguishing Attack. Apart from key recovery attack, another type of attack, known as distinguishing attack, is also possible on a stream cipher. A *distinguisher* is an algorithm which distinguishes a stream of bits from a perfectly random stream of bits, that is, a stream of bits that has been chosen according to the uniform distribution. There are several ways a cryptanalyst may try to distinguish between a string, generated by an insecure pseudorandom bit generator, and one from a perfectly random source.

(a) Regular or Weak Distinguisher. In this case, the adversary selects a *single* key/IV randomly and produces keystream, seeded by the chosen key/IV, long enough to distinguish it from random with high success probability. This attack

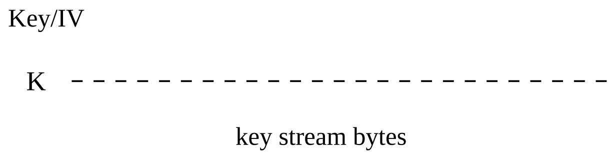

Figure 1.5: Regular distinguisher: adversary collects keystream produced by a single key/IV

scenario is rather common and such *distinguisher* is called a *regular distinguisher* (see Fig. 1.5). In this scenario the adversary is "weak" as she has a keystream produced by a single key rather than many, and therefore the *distinguisher* is also called a *weak distinguisher*.

(b) Prefix or Strong Distinguisher. In this scenario, to build a distinguisher, the adversary may use *many* randomly chosen key/IVs rather than a single key and a few *specified* bytes from each of the keystreams generated by those key/IVs (see Fig. 1.6). The *distinguisher*, so constructed, is called a *prefix distinguisher*. In this case the adversary is "strong" because she may collect outputs to her advantage from many keystreams to detect a bias. Therefore, the *distinguisher* so constructed is also termed a *strong distinguisher*. A bias present in the output at time t in a single stream may hardly be detected by a *regular distinguisher* but a *prefix distinguisher* can easily discover the anomaly with a few bytes. This fact was nicely demonstrated by Mantin and Shamir [53] to detect a strong bias toward zero in the second output byte of RC4.

(c) Hybrid Distinguisher. In addition to that, there exist *hybrid distinguishers* that may fall between the above two extreme cases, that is, the adversary may use *many* key/IVs and for each key/IV she collects *long* keystream.

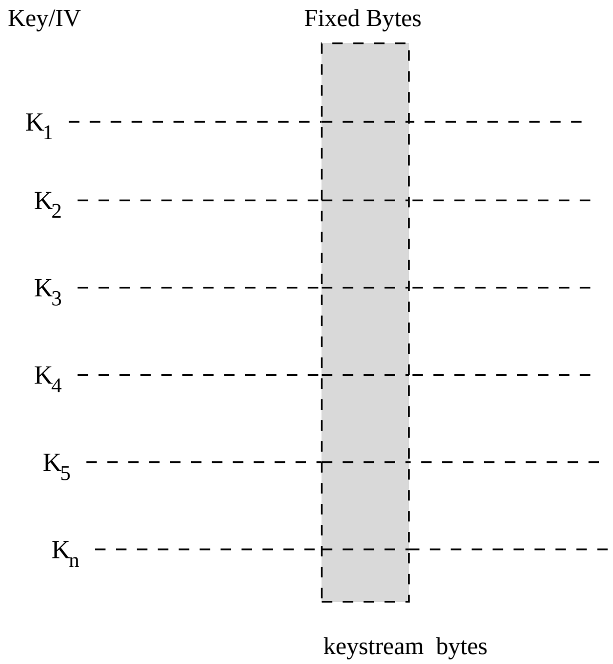

Figure 1.6: Prefix distinguisher: adversary collects a few fixed bytes from keystreams produced by many key/IVs

The idea of constructing distinguishers using *many* randomly chosen key/IVs has been a well studied subject. Goldreich has shown that a distribution which is *computationally indistinguishable* from the *uniform distribution* based on a *single sample* is also *computationally indistinguishable* from the *uniform distribution* based on *multiple samples* [34].

Advantage of a Distinguisher. The *advantage* of a distinguisher $\mathcal{F}$ is a measure which indicates the efficiency of the algorithm $\mathcal{F}$ to distinguish a distribution D_0 from another distribution D_1. It is given by the following formula [3]:

$$\mathrm{Adv}_{\mathcal{F}} = \left| P_{\mathsf{D}_0}[\mathcal{F}(z) = 1] - P_{\mathsf{D}_1}[\mathcal{F}(z) = 1] \right| . \tag{1.8}$$

(iii) Related Key Attack. Apart from the above two attacks, related key attack (or related IV attack) sometimes proves to be very crucial for a stream cipher. In a related key attack it is found that, if some relation exists among a group of key/IVs (known as weak key/IVs), the corresponding outputs are also related. This relation can be used to recover secret key if some other conditions are satisfied [30].

1.4 Summary of the Results: Array and Modular Addition as Stream Cipher Components

The discourse on the rudiments of cryptology now lead us gradually to move deeper into the main contribution of this thesis. One of the major inspirations for working on stream ciphers for a Ph.D. thesis is to catch up with the recent flurry of activities and increased interests among the cryptologists to design sophisticated stream ciphers and to develop powerful cryptanalytic techniques to analyze them. This recent such enthusiasm in the areas of stream ciphers is largely attributed to two European projects: (1) NESSIE which stands for New European Schemes for Signatures, Integrity and Encryption [61] and (2) ECRYPT which is a Network of Excellence within the Information Societies Technology (IST) Programme of the European Commission [25]. The NESSIE project, which lasted four years (from 2000 till 2004), is ironically a failed one with respect to stream ciphers, as none of the submitted stream ciphers could be included in the final portfolio. However, this project indubitably established the requirements for deeper and more comprehensive study of stream ciphers. As a result, another European project eSTREAM (under the umbrella of ECRYPT), which called for primitives on stream ciphers *only*, was started in February 2004. In April 2005, ECRYPT received 34 candidate stream ciphers for its eSTREAM project.

This thesis scrutinizes a special class of stream ciphers which use arrays as internal states and modular additions as the main operations to produce output streams. In the course of our investigation, we show cryptanalytic attacks on 9 well known practical stream ciphers. Below we summarize the results of the thesis chapter by chapter.

Chapter 2. The first contribution of the thesis is some theoretical results on a class of equations known as DEA. Mixing addition modulo 2^n $(+)$ and exclusive-or $(\oplus)$ has a host of applications in symmetric cryptography as the operations are fast and nonlinear over GF(2). We deal with a frequently encountered equation $(x+y)\oplus((x\oplus\alpha)+(y\oplus\beta))=\gamma$. The difficulty of solving an arbitrary system of such equations – named *differential equations of addition* (DEA) – is an important consideration in the evaluation of the security of many ciphers against *differential attacks*. We show that the satisfiability of an arbitrary set of DEA – which has so far been assumed *hard* for large n – is in the complexity class $\mathcal{P}$. We also design an efficient algorithm to obtain all solutions to an arbitrary system of DEA with running time linear in the number of solutions. The next contribution is solving DEA in an *adaptive query model* where an equation is formed by a query (α,β)

and oracle output γ. The challenge is to optimize the number of queries to solve $(x + y) \oplus ((x \oplus \alpha) + (y \oplus \beta)) = \gamma$. Our algorithm solves this equation with only 3 queries in the worst case. Another algorithm solves the equation $(x + y) \oplus (x + (y \oplus \beta)) = \gamma$ with $(n - t - 1)$ queries in the worst case (t is the position of the least significant '1' of x), and thus, outperforms the previous best known algorithm by Muller – presented at FSE '04 [59]– which required $3(n - 1)$ queries. Most importantly, we show that the upper bounds, for our algorithms, on the number of queries match worst case lower bounds. This, essentially, closes further research in this direction as our lower bounds are *optimal*. We also solve the above equations with a set of batch queries. Our algorithms require 6 and 2^{n-2} queries to solve the above equations where the lower bounds are $\frac{3}{4} \cdot 2^{n-2}$ (theoretically proved) and 4 (based on extensive experiments) respectively.

The above results are used to cryptanalyze a recently proposed cipher, namely Helix [33], which was a candidate for consideration in the 802.11i standard. We recover the secret key of the cipher with $2^{10.6}$ *adaptive chosen plaintexts* which is an improvement on the previous best known attack by a factor of 3 (the attack can be improved by a factor of 46.5 in the best case). Furthermore, for the first time, we show a key-recovery attack with $2^{35.6}$ *chosen plaintexts* rather than *adaptive chosen plaintexts*. All the above attacks assume reuse of nonce similar to the assumptions of all previous attacks.

The results of this chapter have been published in the volumes 3574 and 3797 of *Lecture Notes in Computer Science* [66, 67].

Chapter ??. Next, we analyze the RC4 stream cipher which is the most widely used software based stream cipher. The cipher is based on a secret internal state of $N = 256$ bytes and two pointers. This thesis proposes an efficient algorithm to compute a special set of RC4 states named *non-fortuitous predictive states*. These special states increase the probability to guess part of the *internal state* in a known plaintext attack and present a cryptanalytic weakness of RC4. The problem of designing a practical algorithm to compute them has been open since it was posed by Mantin and Shamir in 2001. We also formally prove a slightly corrected version of the conjecture by Mantin and Shamir of 2001 that, using only a known elements along with the two pointers at some round, the RC4 pseudorandom generation algorithm cannot produce more than a outputs in the next N rounds. Then, we present a new statistical bias in the distribution of the first two output bytes of the RC4 keystream generator. The number of outputs required to reliably distinguish RC4 outputs from random strings using this bias is only 2^{25} bytes. Most importantly we show that the bias does not disappear even if the initial 256 bytes of the keystream are dropped.

The results of this chapter have been published in the volumes 2904 and 3017 of *Lecture Notes in Computer Science* [64, 65].

Chapter 4. This chapter proposes a new stream cipher (or PRBG), named

RC4A, which is based on RC4's exchange shuffle model. It is shown that the new cipher offers increased resistance against most attacks that apply to RC4. RC4A uses fewer operations per output byte and offers the prospect of implementations that can exploit its inherent parallelism to improve its performance further. We also mention two distinguishing attacks on this cipher.

The results of this chapter have been included in the volume 3017 of *Lecture Notes in Computer Science* [65].

Chapter 5. After RC4, RC4A and Helix, we focus our attention on another array-based stream cipher Py which was designed by Biham and Seberry. Py is also a submission to the ECRYPT stream cipher competition. The cipher is based on two large arrays (one is 256 bytes and the other is 1040 bytes) and it is designed for high speed software applications (Py is more than 2.5 times faster than the RC4 on Pentium III). In the thesis we show a statistical bias in the distribution of its output-words at the 1st and 3rd rounds. Exploiting this weakness, a distinguisher with advantage greater than 50% is constructed that requires $2^{84.7}$ randomly chosen key/IV's and the first 24 output bytes for each key. The running time and the data required by the distinguisher are $t_{ini} \cdot 2^{84.7}$ and $2^{89.2}$ respectively (t_{ini} denotes the running time of the key/IV setup). We further show that the data requirement can be reduced by a factor of about 3 with a distinguisher that considers outputs of later rounds. In such case the running time is reduced to $t_r \cdot 2^{84.7}$ (t_r denotes the time for a single round of Py). The Py specification allows a 256-bit key and a keystream of 2^{64} bytes per key/IV. As an ideally secure stream cipher with the above specifications should be able to resist the attacks described before, our results constitute an academic break of Py. In addition we have identified several biases among pairs of bits; it seems possible to combine all the biases to build more efficient distinguishers.

The results of this chapter have been published in the volume 4047 of *Lecture Notes in Computer Science* [68].

Chapter 6. In this chapter, from the experience of the attacks on the ciphers described above, we investigate the security of array-based stream ciphers against certain types of distinguishing attacks in a unified way. We argue, counter-intuitively, that the most useful characteristic of an array, namely, the association of array-elements with unique indices, may turn out to be the origins of distinguishing attacks if adequate caution is not maintained. In short, an adversary may attack a cipher simply exploiting the dependence of array-elements on the corresponding indices. Most importantly, the weaknesses are not eliminated even if the indices and the array-elements are made to follow uniform distributions separately. Exploiting these weaknesses we build distinguishing attacks with reasonable advantage on five recent stream ciphers, namely, Py6 (2005, Biham *et al.*), IA, ISAAC (1996, Jenkins Jr.), NGG, GGHN (2005, Gong *et al.*) with data complexities $2^{68.61}$, $2^{32.89}$, $2^{16.89}$, $2^{32.89}$ and $2^{32.89}$ respectively.

Similar to the RC4 and the Py, in all the above cases also we worked under the assumption that the key-setup algorithms of the ciphers produced uniformly distributed internal states. We only investigated the mixing of bits in the keystream generation algorithms. In hindsight, we also observe that the previous attacks on the other array-based stream ciphers (e.g. Py, etc.), can also be explained in the general framework developed in this chapter. It seems that our analyses will be useful in the evaluation of the security of stream ciphers based on arrays and modular addition.

The results of this chapter will be published shortly in a volume of *Lecture Notes in Computer Science* [69].

Chapter 7. Finally we conclude and leave some open problems as future work.

Chapter 2

Differential Equations of Addition and Applications to the Helix Cipher

Science is a differential equation. Religion is a boundary condition.
-Alan Turing (1912-1954)

2.1 Introduction

Addition modulo 2^n. Mixing *addition modulo* 2^n $(+)$ with other Boolean operations such as *exclusive-or* $(\oplus)$, *or* $(\vee)$ and/or *and* $(\wedge)$ is extensively used in symmetric cryptography. The main motivation for including *addition mod* 2^n in cryptographic primitives is that it is a nonlinear transformation over GF(2) and the operation is extremely fast on all present day architectures. Nonlinear transformations are of paramount importance in the design of ciphers as they make functions hard to invert. Helix [33], IDEA [50], Mars [12], RC6 [77], and Twofish [81] which mix modular *addition* with *exclusive-or* are a few examples of the application of *addition*. Very recently Klimov and Shamir also used an update function for internal state, known as a T-function, where *addition* is mixed with *multiplication* and *or* in a certain fashion to achieve many useful properties of a secure stream cipher [45, 46].

Keeping with the trend of widespread use of *addition* in symmetric ciphers, there is a large body of literature that studies equations involving *addition* from many different angles. Staffelbach and Meier investigated the probability distribution of the carry for integer addition [87]. Wallén explained the linear approximations of modular addition [93]. Lipmaa and Moriai [51] investigated the equation $(x + y) \oplus ((x \oplus \alpha) + (y \oplus \beta)) = \gamma$, where α, β are the input differences and γ is the output difference, to compute many differential properties. The dual of the above equation $(x \oplus y) + ((x + \alpha) \oplus (y + \beta)) = \gamma$ was investigated for differential properties by Lipmaa *et al.* [52].

Differential Cryptanalysis (DC). Differential Cryptanalysis, introduced by Biham and Shamir [9], is one of the most powerful attacks against symmetric ciphers. There are broadly two lines of attacks based on DC. The first line of attack exploits the occurrences of input or output differences with nontrivial probabilities. In a cipher that is secure against DC, input and output differences should behave 'pseudorandomly', so that none of them can be guessed from any known values with a nontrivial probability. This line of attack usually results in distinguishing attacks [91]. A second line of attack is much stronger but more difficult to implement than the other. It recovers secret information from known input and output differences, akin to the algebraic attacks [59]. Note that this second line of attack implies the first but the converse is not true. Therefore, provable security against DC – introduced by Lai *et al.* [50] and first implemented by Nyberg and Knudsen [63] – remained a key factor in the evaluation of the security of a cipher. However, the security of many complex modern ciphers against DC is hard to evaluate because of lack of theory to evaluate the security of its components. Our target is to mount a second line of attack (i.e., to recover secret information) on the much used symmetric cipher component *addition modulo* 2^n.

Definition of DEA. There are two basic *addition* equations under DC where differences of inputs and outputs are expressed as *exclusive-or*.

$$(x + y) \oplus (x + (y \oplus \beta)) = \gamma\,, \tag{2.1}$$

$$(x + y) \oplus ((x \oplus \alpha) + (y \oplus \beta)) = \gamma\,. \tag{2.2}$$

These equations are named *differential equations of addition* (DEA). While engaged in cryptanalysis of MD5, Berson noted in 1992 that, for large n, it is *hard* to analyze modular addition when differences are expressed as XOR [4]. This may have motivated the use of *addition* in conjunction with XOR in many symmetric ciphers to increase resistance against DC.

Our Results. The results of this chapter can be divided into four parts.

(i) Solving an Arbitrary Set of DEA (elaborated in Sect. 2.2): we show that the satisfiability of a randomly generated set of DEA is in the complexity class $\mathcal{P}$. In other words, a *Turing machine* can show in $\mathcal{O}(n^k)$ time whether there exists a solution to an arbitrary set of DEA (n denotes the bit-length of x, y and

$k > 0$ is an integer-valued constant computed from the degree of the polynomial (in n) which is an upper bound on the number of equations to be solved). This result, on one hand, gives deeper insight into the behavior of addition under DC. On the other hand this leaves a cautionary note for the cryptographers to be more careful about using addition in the design. Outside cryptography, satisfiability of a system of equations has a natural appeal to many areas such as computational complexity, combinatorics, circuit optimization and computer algebra (remember the most famous $\mathcal{NP}$-Complete satisfiability problem: Boolean formula satisfiability [14]). For example, if a large system of DEA is NOT satisfiable then the whole circuit representing the system of DEA can be safely removed to optimize the circuit complexity. Going beyond the satisfiability problem, we also give an efficient algorithm to compute all the solutions to a randomly generated system of DEA with running time linear in the number of solutions. Another subtle but a noteworthy aspect of our work is the departure from the traditional technique for solving multivariate polynomial equations over GF(2) [2]. We heavily benefit from certain properties of DEA and solve such systems combinatorially.

(ii) Solving DEA with Adaptive Queries (elaborated in Sect. 2.3): next we extend our work to solve DEA in a crypto-friendly *adaptive query model*. The aim is to *minimize* the search space of the secret (x, y) using a *minimum* number of adaptive queries (α, β). Such an optimization problem – typically used to reduce data complexity of adaptive chosen plaintext attacks – was first tackled by Muller [59] for (2.1). But an optimal solution has been elusive until now. We achieve optimal solutions for both the equations. We show that a worst case lower bound on the number of queries $(0, \beta)$ to solve (2.1) is $(n - t - 1)$ where $(n - t) > 1$ with t being the bit-position of the least significant '1' of x. A worst case lower bound on the number of queries (α, β) to solve (2.2) is 3 for $n > 2$. Most importantly, for solving the above equations we also design algorithms whose upper bounds on the number of queries match worst case lower bounds. Note that our algorithm outperforms the previous best known algorithm by Muller to solve (2.1) – presented at FSE '04 – which required $3(n - 1)$ queries [59]. Over and above, our results essentially close further investigation in this particular direction as the equations are solved with an *optimal* number of queries in the worst case. It is particularly interesting to note that, for (2.2), although the number of all queries grows exponentially with the input size n, an optimal lower bound to solve (2.2) is 3 for all $n > 2$, i.e., constant asymptotically.

(iii) Solving DEA with Batch Queries (elaborated in Sect. 2.4): from cryptographic point of view, *batch queries* are more practical than the *adaptive queries*. Attacks based on *batch* and *adaptive* queries are equivalent to *chosen plaintext* (CP) attack and *adaptive chosen plaintext* (ACP) attack respectively. In a CP

attack, the queries are submitted all at once, where in an ACP attack the queries are submitted adaptively, i.e., the next query is submitted based on the answers to the previous queries – a situation which is difficult to implement in practice. There is a large number of research papers launching CP and ACP attacks on many practical ciphers. For example, the boomerang attack introduced by Wagner is an ACP attack [92] and, therefore, less practical. The slide attacks by Biryukov and Wagner can be implemented using both CP and ACP but with different amount of data and time [10]. The time-memory trade-off attack by Hellman [39] and the differential attack on DES by Biham and Shamir [9] are CP attacks; so they are attributed more practical importance.

Therefore, it is also important to solve DEA with *batch* queries where the adversary has access to only one computing oracle (note that, with *adaptive queries*, the attacker has access to two computing machines). Our algorithm solves (2.1) with 2^{n-2} queries submitted in a batch, where a lower bound on the number of queries is $\frac{3}{4} \cdot 2^{n-2}$ (for all $n > 3$), i.e., our lower bound is optimal up to a constant factor of $\frac{4}{3}$. This exponential lower bound constitutes an important theoretical reference point which endorses the equations's strong resistance against DC. On the other hand, (2.2) has been solved with only 6 (for all $n > 2$) *batch* queries which is two more than a conjectured lower bound (note that the total number of queries is 2^{2n}) – this fact shows a major cryptographic weakness of *addition* under DC and therefore, this component should be used with caution.

(iv) Cryptanalysis of the Helix Cipher (elaborated in Sect. 2.5): our results can be used to cryptanalyze a recently proposed cipher, namely Helix [33], which was a candidate for consideration in the 802.11i standard. We recover the secret key of the cipher with $2^{10.6}$ *adaptive chosen plaintexts*. This attack is an improvement on the previous best known attack by a factor of 3 [59]. We also show that the attack can be improved by a factor of 46.5 in the best case. In addition, we also demonstrate, for the first time, that the Helix cipher can also be attacked with $2^{35.6}$ *chosen plaintexts* rather than *adaptive chosen plaintexts*. Note that both of the above attacks are built under the assumption of reuse of nonce similar to the previous attack on it by Muller [59].

2.1.1 Model of Computation

The purpose of the chapter is to solve (2.1) and (2.2) for (x, y) using triples (α, β, γ) where $x, y, \alpha, \beta, \gamma \in \mathbb{Z}_2^n$. The operation '+' over $\mathbb{Z}_{2^n}$ will be viewed as a binary operation over $\mathbb{Z}_2^n$ using the bijection that maps $(l_{(n-1)}, \cdots, l_{(0)}) \in \mathbb{Z}_2^n$ to $l_{(n-1)}2^{n-1} + \cdots + l_{(0)}2^0 \in \mathbb{Z}_{2^n}$. Therefore, '+' : $\mathbb{Z}_2^n \times \mathbb{Z}_2^n \to \mathbb{Z}_2^n$.

The algorithms, described in this chapter, can be implemented on a generic one-processor *Random Access Machine* (RAM) (i.e., instructions are executed sequentially) with a memory that is composed of an unbounded sequence of registers each capable of containing an integer. Typically, RAM instructions consist of simple arithmetic operations (addition and bitwise XOR in our case), storing, addressing (direct and indirect) and branching, each of which is a constant time operation. However, the choice of RAM instructions is relatively less important because algorithms based on two reasonable sets of instructions will have the same asymptotic complexity. A detailed analysis of RAM can be found in [1, 32]. It can be shown that a polynomial-time solvable problem on a RAM is also polynomial-time solvable on a *Turing Machine* and vice versa.

2.2 Solving an Arbitrary System of DEA

Our aim is to solve for (x, y) from the following set of differential equations of addition over $\mathbb{Z}_2^n$,

$$(x+y) \oplus ((x \oplus \alpha[k]) + (y \oplus \beta[k])) = \gamma[k]\,, \qquad k = 1, 2 \cdots m\,. \tag{2.3}$$

We observe that the ith bit of $\gamma[k]$ can be written as

$$\gamma[k]_{(i)} = x_{(i)} \oplus y_{(i)} \oplus c_{(i)} \oplus \tilde{x}_{(i)} \oplus \tilde{y}_{(i)} \oplus \tilde{c}_{(i)}, \quad i \in [0, n-1]\,, \tag{2.4}$$

where $\tilde{x}_{(i)} = x_{(i)} \oplus \alpha[k]_{(i)}$ and $\tilde{y}_{(i)} = y_{(i)} \oplus \beta[k]_{(i)}$ and $c_{(i)}$, $\tilde{c}_{(i)}$ are computed from the following recursion,

$$\begin{aligned} c_{(0)} &= \tilde{c}_{(0)} = 0, \\ c_{(j+1)} &= x_{(j)}y_{(j)} \oplus x_{(j)}c_{(j)} \oplus y_{(j)}c_{(j)}, \\ \tilde{c}_{(j+1)} &= \tilde{x}_{(j)}\tilde{y}_{(j)} \oplus \tilde{x_{(j)}}\tilde{c}_{(j)} \oplus \tilde{y}_{(j)}\tilde{c}_{(j)}, \quad \text{where } j \in [0, n-2]\,. \end{aligned} \tag{2.5}$$

Thus we see that $\gamma[k]_{(i)}$ is a function of the least $(i+1)$ bits of x, y, $\alpha[k]$ and $\beta[k]$. More formally,

$$\begin{aligned} \gamma[k]_{(i)} &= F_i(x_{(0)}, \cdots, x_{(i)}, y_{(0)}, \cdots, y_{(i)}, \\ &\qquad \alpha[k]_{(0)}, \cdots, \alpha[k]_{(i)}, \beta[k]_{(0)}, \cdots, \beta[k]_{(i)})\,. \end{aligned} \tag{2.6}$$

Therefore, from a system of m differential equations of addition, a total of mn multivariate polynomial equations over GF(2) can be formed by ranging (k, i) through all values in (2.6).

Plenty of research has been undertaken to design efficient ways to solve randomly generated multivariate polynomial equations. The classical Buchberger's Algorithm for generating Gröbner bases [2] and its variants [26] are some of them. This problem is $\mathcal{NP}$-complete ($\mathcal{NPC}$) over GF(2). Many other techniques such as relinearization [44] have been proposed to solve a special case of overdefined systems of multivariate polynomial equations. Note that, in our case, the number of unknowns and equations are $2n$ and mn respectively (if $m > 2$ then the system of equations is overdefined). However, taking full advantage of the specific nature of the *differential equations of addition*, we shall use a combinatorial technique to prove that, although the satisfiability of an arbitrary multivariate polynomial equation over GF(2) is $\mathcal{NP}$-complete, this special cryptographically important subclass of equations is in the complexity class $\mathcal{P}$ (see [14], [40] for definitions of $\mathcal{NP}$, $\mathcal{P}$, $\mathcal{NPC}$). Finally, we also derive all the solutions to a system of such equations.

2.2.1 Computing the *Character Set* and the *Useful Set*

From (2.3) we construct $A = \{(\alpha[k], \beta[k], \gamma[k]) \mid k \in [1, m]\}$ assuming $(\alpha[k], \beta[k], \gamma[k])$'s are all distinct.[1] We call A the *character set* for that particular set of equations. Our first step is to transform the system of equations defined in (2.3) into a new set of equations over $\mathbb{Z}_2^n$ as defined below,

$$(x + y) \oplus ((x \oplus \alpha[k]) + (y \oplus \beta[k])) \oplus \alpha[k] \oplus \beta[k] = \tilde{\gamma}[k]\,, \qquad k = 1, 2 \cdots m\,; \tag{2.7}$$

where $\tilde{\gamma}[k] = \gamma[k] \oplus \alpha[k] \oplus \beta[k]$. Now, we construct $\tilde{A}$,

$$\tilde{A} = \{(\alpha, \beta, \tilde{\gamma} = \alpha \oplus \beta \oplus \gamma) \mid (\alpha, \beta, \gamma) \in A\}\,. \tag{2.8}$$

We call $\tilde{A}$ the *useful set*. Let all the solutions for (2.3) and (2.7) be contained in the sets A-consistent and $\tilde{A}$-consistent respectively. It is direct to show that

$$A\text{-consistent} = \tilde{A}\text{-consistent}\,. \tag{2.9}$$

Our aim is to compute $\tilde{A}$-consistent from $\tilde{A}$.

2.2.2 Precomputation

Take an arbitrary element $(\alpha, \beta, \tilde{\gamma}) \in \tilde{A}$ $(n > 1)$. Observe that $\tilde{\gamma}_{(i+1)}$ can be computed using $x_{(i)}$, $y_{(i)}$, $c_{(i)}$, $\alpha_{(i)}$, $\beta_{(i)}, \tilde{\gamma}_{(i)}$, $\forall\, i \in [0, n-2]$, from the following three equations:

$$\begin{aligned}
\tilde{\gamma}_{(i+1)} &= c_{(i+1)} \oplus \tilde{c}_{(i+1)}, \\
c_{(i+1)} &= x_{(i)}y_{(i)} \oplus x_{(i)}c_{(i)} \oplus y_{(i)}c_{(i)}, \\
\tilde{c}_{(i+1)} &= \tilde{x}_{(i)}\tilde{y}_{(i)} \oplus \tilde{x}_{(i)}\tilde{c}_{(i)} \oplus \tilde{y}_{(i)}\tilde{c}_{(i)}
\end{aligned}$$

where $c_{(i)}$ is the carry at the ith position of $(x + y)$, $\tilde{x}_{(i)} = x_{(i)} \oplus \alpha_{(i)}$, $\tilde{y}_{(i)} = y_{(i)} \oplus \beta_{(i)}$ and $\tilde{c}_{(i)} = c_{(i)} \oplus \tilde{\gamma}_{(i)}$. Table 2.1 lists the values of $\tilde{\gamma}_{(i+1)}$ as computed from all values of $x_{(i)}$, $y_{(i)}$, $c_{(i)}$, $\alpha_{(i)}$, $\beta_{(i)}$, $\tilde{\gamma}_{(i)}$.

[1] This can be obtained by taking one of the identical equations in (2.3).

Table 2.1: The values of $\tilde{\gamma}_{(i+1)}$ corresponding to $(x_{(i)},\, y_{(i)},\, c_{(i)})$ (tabulated in column $k = 0$) and $(\alpha_{(i)},\, \beta_{(i)},\, \tilde{\gamma}_{(i)})$ (tabulated in row $r = 0$). A row and a column are denoted by $R(r)$ and $\mathrm{Col}(k)$. $R(r)\times\mathrm{Col}(k)$ denotes the element in the table corresponding to row r and column k.

	$2^2 \cdot \alpha_{(i)} + 2 \cdot \beta_{(i)} + \tilde{\gamma}_{(i)}$								r
$(x_{(i)}, y_{(i)}, c_{(i)})$	0	1	2	3	4	5	6	7	0
(0,0,0)	0	0	0	1	0	1	1	1	1
(1,1,1)									
(0,0,1)	0	0	1	0	1	0	1	1	2
(1,1,0)									
(0,1,0)	0	1	0	0	1	1	0	1	3
(1,0,1)									
(1,0,0)	0	1	1	1	0	0	0	1	4
(0,1,1)									
0	1	2	3	4	5	6	7	8	k

2.2.3 Computation of Parameters G_i, $S_{i,0}$ and $S_{i,1}$ from the *Useful Set* $\tilde{A}$

We now determine an important quantity, denoted by G_i, for a nonempty *useful set* $\tilde{A}$. In G_i, we store the ith and $(i+1)$th bits of $\tilde{\gamma}$ and the ith bit of α and β for all $(\alpha,\, \beta,\, \tilde{\gamma}) \in \tilde{A}$. We call G_i the ith core of the *useful set* $\tilde{A}$. More formally (suppose $n > 1$),

$$G_i = \{(\alpha_{(i)},\, \beta_{(i)},\, \tilde{\gamma_{(i)}},\, \tilde{\gamma}_{(i+1)}) \,|\, (\alpha,\, \beta,\, \tilde{\gamma}) \in \tilde{A}\}, \quad i \in [0,\, n-2]\,. \tag{2.10}$$

In the subsequent discussion we will often use the expression "$G_i \to (x_{(i)},\, y_{(i)},\, c_{(i)})$". The meaning of the expression becomes clear in the following steps where we compute $(x_{(i)},\, y_{(i)},\, c_{(i)})$ such that $G_i \to (x_{(i)},\, y_{(i)},\, c_{(i)})$.

1. Let $|G_i| = g$. Take an element $(\alpha_{(i)},\, \beta_{(i)},\, \tilde{\gamma}_{(i)},\, \tilde{\gamma}_{(i+1)}) \in G_i$.

2. In Table 2.1, find the row(s) of the fourth coordinate $\tilde{\gamma}_{(i+1)}$ in the column specified by the first three coordinates $(\alpha_{(i)},\, \beta_{(i)},\, \tilde{\gamma}_{(i)})$ in $R(0)$ and put them in the set F_{i1}.

3. Find $F_{i1}, \cdots F_{ig}$ for all g elements of G_i.

4. Compute $F_i = \bigcap_j F_{ij}$.

5. Let $R(r) \in F_i$. If $(x_{(i)}, y_{(i)}, c_{(i)})$ is in Col(0)$\times R(r)$ then we say $G_i \to (x_{(i)}, y_{(i)}, c_{(i)})$. If $F_i = \phi$ then no such $(x_{(i)}, y_{(i)}, c_{(i)})$ exists.

Now we compute,

$$S_{i,\,j} = \{(x_{(i)}, y_{(i)}) \,|\, G_i \to (x_{(i)}, y_{(i)}, c_{(i)} = j)\}, \quad (i, j) \in [0,\, n-2][0,\, 1]. \tag{2.11}$$

The example below computes $G_i, S_{i,\,0}, S_{i,\,1}$, following the method described above.

Example. $(G_i, S_{i,\,0}, S_{i,\,1})$ Let $n = 3$ and the *useful set* $\tilde{A} = \{((0,\,1,\,0), (1,\,0,\,1), (0,\,0,\,0)), ((0,0,0), (1,\,1,\,1), (1,0,0)), ((0,\,0,\,1), (0,\,1,\,1), (1,\,1,\,0))\}$. Therefore, $G_0 = \{(0,\,1,\,0,\,0), (1,\,1,\,0,\,1)\}, G_1 = \{(1,\,0,\,0,\,0), (0,\,1,\,0,\,1), (0,\,1,\,1,\,1)\}$ (see (2.10)). Now, from Table 2.1, $F_{01} = \{R(1), R(3)\}$, $F_{02} = \{R(1), R(2)\}$, $F_{11} = \{R(1), R(4)\}$, $F_{12} = \{R(2), R(4)\}$, $F_{13} = \{R(1), R(4)\}$. Therefore, $F_0 = F_{01} \cap F_{02} = \{R(1)\}$ and $F_1 = F_{11} \cap F_{12} \cap F_{13} = \{R(4)\}$. Now $G_0 \to (0,\,0,\,0)$, $G_0 \to (1,\,1,\,1)$ because $(0,\,0,\,0)$, $(1,\,1,\,1)$ are in Col(0)$\times R(1)$. Similarly, $G_1 \to (1,\,0,\,0)$, $G_1 \to (0,\,1,\,1)$. Thus, $S_{0,\,0} = \{(0,\,0)\}$, $S_{0,\,1} = \{(1,\,1)\}$, $S_{1,\,0} = \{(1,\,0)\}$, $S_{1,\,1} = \{(0,\,1)\}$. □

We assume $m = |\tilde{A}| = \mathcal{O}(n^l)$ for some nonnegative integer l. Observing that the number of G_i's (or S_i's) is $\mathcal{O}(n)$, the time and memory to compute all the G_i's and $S_{i,\,j}$'s are $\mathcal{O}(n^k)$ each, because the size of Table 2.1 and $|G_i|$ are $\mathcal{O}(1)$ each ($k = l + 1$). Now, we show a relation between $S_{i,\,0}$ and $S_{i,\,1}$ that will be used to obtain several results.

Proposition 2.1 *For all nonempty useful set* $\tilde{A}$ *and all* $n > 1$, $|S_{i,\,0}| = |S_{i,\,1}|$ $\forall\, i \in [0,\, n-2]$.

Proof. In Table 2.1, we observe that each row $R(r)$ (where $r = 1,\,2,\,3,\,4$) corresponds to two triplets of $(x_{(i)}, y_{(i)}, c_{(i)})$, where one of them is bitwise complement of the other (see Col(0) in Table 2.1). Therefore, from (2.11), for all $j \in [0, 1]$, the number of elements in $S_{i,\,j}$ is the same for all $i \in [0, n-2]$. □

We set,

$$|S_{i,\,0}| = |S_{i,\,1}| = S_i, \quad \forall\, i \in [0,\, n-2]. \tag{2.12}$$

2.2.4 Satisfiability of DEA is in $\mathcal{P}$

In this section, we deal with a decision problem: does there exist a solution for an arbitrary set of differential equations of addition, i.e., is a system of DEA satisfiable?

We have already seen how to compute the *character set* A, the *useful set* $\tilde{A}$, the core G_i's and S_i's from a system of DEA in $\mathcal{O}(n^k)$ (see Sect. 2.2.3). Now, we prove an important theorem that characterizes the membership in $\tilde{A}$-consistent which is, in fact, the solution set.

Theorem 2.2 *Let the useful set $\tilde{A} \neq \phi$ and $n > 1$. The following two statements are equivalent.*
1. $(x, y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$ is such that $G_i \rightarrow (x_{(i)}, y_{(i)}, c_{(i)})$, $\forall\, i \in [0, n-2]$.
2. $(x, y) \in \tilde{A}$-consistent.

Proof. The proof is divided into two parts: (i) proof of $1 \Rightarrow 2$ and (ii) proof of $2 \Rightarrow 1$.

(i) $1 \Rightarrow 2$. From the construction of Table 2.1 and the G_i, any pair (x, y) satisfying property 1 is a solution to the set of equations represented by the set $\tilde{A}$. Therefore, $1 \Rightarrow 2$.

(ii) $2 \Rightarrow 1$. We prove this by contradiction. Suppose, there exists a pair $(x, y) \in \tilde{A}$-consistent that does not satisfy property 1. Therefore, there exists S_i, for some $i \in [0, n-2]$, which is empty. But, if (x, y) is a solution then S_i is non-empty for all $i \in [0, n-2]$. Thus we reach a contradiction.

Combining (i) and (ii), the theorem is proved. □

Armed with the above theorem, we now formulate $|\tilde{A}$-consistent$|$ (i.e., the number of solutions) in the following proposition which will later answer our satisfiability question.

Proposition 2.3 *Let the useful set $\tilde{A} \neq \phi$ and S denote $|\tilde{A}$-consistent$|$. Then,*

$$S = \begin{cases} 0 & \text{if } \tilde{\gamma}_{(0)} = 1 \text{ for some } (\alpha,\, \beta,\, \tilde{\gamma}) \in \tilde{A}, \\ 4 \cdot \prod_{i=0}^{n-2} S_i & \text{if } \tilde{\gamma}_{(0)} = 0,\ \forall (\alpha,\, \beta,\, \tilde{\gamma}) \in \tilde{A} \text{ and } n > 1, \\ 4 & \text{if } \tilde{\gamma}_{(0)} = 0,\ \forall (\alpha,\, \beta,\, \tilde{\gamma}) \in \tilde{A} \text{ and } n = 1. \end{cases}$$

The S_i's are defined in (2.12).

Proof. We prove this result by considering all cases individually.

Case 1 If $\tilde{\gamma}_{(0)} = 1$ for some $(\alpha, \beta, \tilde{\gamma}) \in \tilde{A}$ then $c_{(0)} = 1$ which is impossible.

Case 2 If $\tilde{\gamma}_{(0)} = 0$, $\forall (\alpha, \beta, \tilde{\gamma}) \in \tilde{A}$ and $n > 1$. From Theorem 2.2, S is the number of solutions $(x, y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$ such that $G_i \rightarrow (x_{(i)}, y_{(i)}, c_{(i)}), \forall i \in [0, n-2]$. Let M_k denote the number of solutions for $((x_{(k)}, \cdots, x_{(0)}), (y_{(k)}, \cdots, y_{(0)}))$ such that $G_i \rightarrow (x_{(i)}, y_{(i)}, c_{(i)}), \forall i \in [0, k]$ where $k \in [0, n-2]$. Note that G_k depends only on the least $(k+1)$ bits of x, y, α, β. We consider two subcases.
Case 2(a): $n > 2$. We determine $|\tilde{A}$-consistent$|$ recursively. Let $M_l = M_{l,0} +$

$M_{l,\,1}$ such that $M_{l,\,0}$ solutions produce $c_{(l+1)} = 0$ and $M_{l,\,1}$ solutions produce $c_{(l+1)} = 1$. Therefore, $\forall\, l \in [0,\, n-3]$

$$\begin{aligned} M_{l+1} &= M_{l,\,0} \cdot |S_{l+1,\,0}| + M_{l,\,1} \cdot |S_{l+1,\,1}| \\ &= S_{l+1} \cdot M_l\,. \end{aligned} \tag{2.13}$$

as $|S_{i,\,0}| = |S_{i,\,1}| = S_i,\ \forall\, i \in [0,\, n-2]$ (see Proposition 2.1). Following the recursion in (2.13) we get,

$$M_{n-2} = \prod_{i=0}^{n-2} S_i \tag{2.14}$$

as $M_0 = S_0$. Note that, for all $(\alpha,\, \beta,\, \tilde{\gamma}) \in \tilde{A}$, $\tilde{\gamma}$ can be computed independently of $(x_{(n-1)},\, y_{(n-1)})$; therefore, $(x_{(n-1)},\, y_{(n-1)})$ can take all possible 4 values. Thus,

$$S = 4 \cdot M_{n-2} = 4 \cdot \prod_{i=0}^{n-2} S_i \quad \text{if } n > 2\,. \tag{2.15}$$

Case 2(b): $n = 2$. Using a similar technique as above it can be shown that $S = 4 \cdot S_0$ if $n = 2$.

Case 3 If $\tilde{\gamma}_{(0)} = 0,\, \forall(\alpha,\, \beta,\, \tilde{\gamma}) \in \tilde{A}$ and $n = 1$. It is the trivial case. $S = 4$ when $n = 1$ since, for all $(\alpha,\, \beta,\, \tilde{\gamma}) \in \tilde{A}$, $\tilde{\gamma}$ can be computed independently of $(x_{(n-1)},\, y_{(n-1)})$; therefore, the number of solutions is equal to the all possible 4 values of $(x_{(n-1)},\, y_{(n-1)})$.

Thus the proof is complete. □

Using Proposition 2.3, we now answer the question of satisfiability of DEA in the following claim.

Claim 2.4 *(i) If $\tilde{\gamma}_{(0)} = 1$ for some $(\alpha,\, \beta,\, \tilde{\gamma}) \in \tilde{A}$, then the set of DEA is NOT satisfiable. (ii) If $\tilde{\gamma}_{(0)} = 0,\, \forall(\alpha,\, \beta,\, \tilde{\gamma}) \in \tilde{A}$ and $n > 1$, then the set of DEA is satisfiable if and only if $S_i \neq 0\ \forall i \in [0,\, n-2]$. (iii) If $\tilde{\gamma}_{(0)} = 0,\, \forall(\alpha,\, \beta,\, \tilde{\gamma}) \in \tilde{A}$ and $n = 1$, then the set of DEA is satisfiable.*

Verification of *(i)*, *(ii)* and *(iii)* take time $\mathcal{O}(1)$, $\Theta(n)$ and $\mathcal{O}(1)$ respectively. Therefore, the overall time to decide whether a system of DEA is satisfiable is $\mathcal{O}(n^k) + \mathcal{O}(1) + \Theta(n) + \mathcal{O}(1) = \mathcal{O}(n^k)$. Thus the satisfiability of DEA is in $\mathcal{P}$.

2.2.5 Computing All the Solutions to a System of DEA

Now the only part left unanswered is how to actually compute $\tilde{A}$-consistent, i.e., to extract all the solutions of a system of DEA which is satisfiable. Note, if $n = 1$ then $\tilde{A}$-consistent comprises all 4 values of (x, y). The G_i's can be computed from the *useful set* $\tilde{A}$ in $\mathcal{O}(n^k)$ (see Sect. 2.2.3). Now we compute an intermediate parameter $L_i = \{(x_{(i)}, y_{(i)}, c_{(i)}) \,|\, G_i \to (x_{(i)}, y_{(i)}, c_{(i)})\}$ for all $i \in [0, n-2]$ (note that the L_i's are different from the F_i's which have been computed in Sect. 2.2.3). Computation of the L_i's takes time and memory each $\Theta(n)$. We call the L_i the ith bit solution. Algorithm 1 computes $\tilde{A}$-consistent from the L_i's ($n > 1$).

Algorithm 1 Computing all the solutions to a system of satisfiable DEA

Input: $L_i, \forall i \in [0, n-2]$
Output: $\tilde{A}$-consistent
1: Compute

$$\begin{aligned} M \;=\; & \{((x_{(n-1)}, x_{(n-2)}, \cdots, x_{(0)}), (y_{(n-1)}, y_{(n-2)}, \cdots, y_{(0)})) \\ & \,|\, (x_{(n-1)}, y_{(n-1)}) \in \mathbb{Z}_2^2, (x_{(i)}, y_{(i)}, c_{(i)}) \in L_i, i \in [0, n-2], \\ & c_{(0)} = 0, c_{(i+1)} = x_{(i)}y_{(i)} \oplus x_{(i)}c_{(i)} \oplus y_{(i)}c_{(i)}\}. \end{aligned}$$

2: Return (M).

The idea of the algorithm is to collect in M all $(x, y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$ such that $G_i \to (x_{(i)}, y_{(i)}, c_{(i)}), \forall i \in [0, n-2]$. Theorem 2.2 is the heart of the argument to prove that M is essentially $\tilde{A}$-consistent.

Time and Memory. Algorithm 1 takes time $\Theta(S)$ and memory $\Theta(n \cdot S)$ where S is the number of solutions (an explicit construction of M from the L_i's and its complexity analysis are shown in Appendix A.3).

2.3 Solving DEA in the Adaptive Query Model

In the previous section we have examined how to solve an arbitrary set of DEA. In this section, we deal with the following equations

$$(x + y) \oplus ((x \oplus \alpha) + (y \oplus \beta)) = \gamma, \tag{2.16}$$
$$(x + y) \oplus (x + (y \oplus \beta)) = \gamma \tag{2.17}$$

separately to solve them in an *adaptive query model.*

We, first, outline what is meant by solving (2.16). It means solving the set of 2^{2n} equations generated by ranging (α, β) with the corresponding γ. The number of solutions satisfying 2^{2n} equations is less than that of any subset of the equations. Therefore, solving these 2^{2n} equations reduces the search space of the secret (x, y) to the minimum. This fact is the major motivation for dealing with this problem. The task of a computationally unbounded adversary is to select a subset A of all equations such that the solutions to the chosen subset A are the same as that of the entire 2^{2n} equations. The target of the adversary is to minimize $|A|$. A similar optimization problem can be asked of (2.17) where the number of equations is 2^n. Such an optimization problem for (2.17) has already been tackled by Muller [59] in cryptanalysis of the Helix cipher but an optimal solution has still been elusive. We reach optimal solutions for both equations.

The *adaptive query model* signifies that the adversary forms the set of equations by submitting queries (α, β)'s adaptively and collecting the corresponding γ's. More on that model is explained in the subsequent sections.

2.3.1 The Power of the Adversary

The power of an adversary that solves (2.16) is defined as follows.

1. An adversary has unrestricted computational power and an infinite amount of memory.

2. An adversary can *only* submit queries $(\alpha, \beta) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$ to an honest oracle[2] which computes γ using fixed unknown $(x, y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$ in (2.16) and returns the value to the adversary. We will often refer to that fixed (x, y) as the *seed* of the oracle.

Such an oracle with seed (x, y) is viewed as a mapping $O_{xy} : \mathbb{Z}_2^n \times \mathbb{Z}_2^n \to \mathbb{Z}_2^n$ and defined by

$$O_{xy} = \{(\alpha, \beta, \gamma) \,|\, (\alpha, \beta) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n,\, \gamma = (x + y) \oplus ((x \oplus \alpha) + (y \oplus \beta))\} \,. \quad (2.18)$$

An adversarial model, similar to the one described above for (2.16), can be constructed for (2.17) by setting $(\alpha, \beta) \in \{0\}^n \times \mathbb{Z}_2^n$ and the mapping

$$O_{xy} \,:\, \{0\}^n \times \mathbb{Z}_2^n \to \mathbb{Z}_2^n .$$

The model described above represents a practical adaptively chosen message attack scenario where the adversary makes adaptive queries to an oracle. Based on the replies from the oracle, the adversary computes one or more unknown parameters.

[2] An honest oracle correctly computes γ and returns it to the adversary.

2.3.2 The Task

O_{xy}, defined in (2.18), generates a family of mappings $\mathcal{F} = \{O_{xy} \mid (x, y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n\}$. Note that, if $D \in \mathcal{F}$ then $|D| = 2^{2n}$ for (2.16). Therefore, $D \in \mathcal{F}$ is the *character set* with the number of equations $m = 2^{2n}$ (see Sect. 2.2.1). Our aim is to find all (x, y) satisfying these 2^{2n} equations, i.e., to compute D-consistent from a subset of the *character set* D. If we deal with (2.17) then $|D| = 2^n$.

An Equivalent Task. From the *character set* D one can compute the *useful set* $\tilde{D}$ using (2.8). Therefore, the task is equivalent to the determination of $\tilde{D}$-consistent from a subset of the *useful set* $\tilde{D}$. We call D and $\tilde{D}$ the *total character set* and the *total useful set* as their sizes are maximal and they are generated from a satisfiable set 2^{2n} DEA (because we assumed the oracle to be honest). Note that there is a bijection between D and $\tilde{D}$.

Adjusting the Oracle Output. If the oracle outputs γ on query (α, β), we shall consider the oracle output to be $\tilde{\gamma} = \alpha \oplus \beta \oplus \gamma$ for the sake of simplicity in the subsequent discussions.

Rules of the Game. Now we lay down the rules followed by the adversary to determine the set $\tilde{D}$-consistent that, in turn, gives the essence of the whole problem.

1. The adversary starts with no information about x and y except their size n.

2. The adversary settles on a strategy (i.e., a deterministic algorithm) which is publicly known. Using the strategy, the adversary computes queries adaptively, i.e., based on the previous queries and the corresponding oracle outputs, the next query is determined.

3. The game stops the moment the adversary constructs $\tilde{D}$-consistent. The adversary fails if she is unable to compute $\tilde{D}$-consistent for some $(x, y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$.

We search for an algorithm that determines $\tilde{D}$-consistent for all $(x,\ y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$. Furthermore, there is an additional requirement that, in the worst case of (x, y), the number of queries required by the algorithm is the minimum. We shall elaborate on the meaning of *worst case* in Sect. 2.3.4 which focuses on worst case lower bounds on the number of queries.

2.3.3 The Number of Solutions

In this section we are interested to determine the number of solutions of (2.16) and (2.17) in the adaptive query model. We have already developed a framework where the set of all solutions in the adaptive query model is denoted by $\tilde{D}$-consistent where $\tilde{D}$ is the *total useful set*. Therefore, formally, our effort will be directed to formulate $|\tilde{D}$-consistent$|$. We will see in Theorem 2.10 that, for (2.17), $|\tilde{D}$-consistent$|$ depends on the least significant '1' of x. However, for (2.16), $|\tilde{D}$-consistent$| = 4$, $\forall (x,\, y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$. We shall use these results in Theorem 2.10 and 2.12 of Sect. 2.3.4, to obtain lower bounds on the number of queries to compute $\tilde{D}$-consistent and in Sect. 2.3.5, to prove the correctness of our optimal algorithms.

Theorem 2.5 *Let the position of the least significant '1' of x in the equation*

$$(x + y) \oplus (x + (y \oplus \beta)) = \gamma$$

be t and $x,\, y,\, \beta,\, \gamma \in \mathbb{Z}_2^n$. Let the total useful set $\tilde{D}$ *be given. Then $|\tilde{D}$-consistent$|$ is (i) 2^{t+3} if $n - 2 \geq t \geq 0$, (ii) 2^{n+1} otherwise.*

Proof. *(i)* $n - 2 \geq t \geq 0$. Using the procedure described in Sect. 2.2.3, we construct the ith core G_i, $\forall\, i \in [0,\, n-2]$, from the *total useful set* $\tilde{D}$. We derive that

$$\begin{aligned} G_i &= \{(0,\,0,\,0,\,a_{(i)}),\,(0,\,1,\,0,\,b_{(i)})\}, \quad \forall i \in [0,\,t], &(2.19)\\ G_i &= \{(0,\,0,\,0,\,c_{(i)}),\,(0,\,0,\,1,\,d_{(i)}),\\ &\quad\ (0,\,1,\,0,\,e_{(i)}),\,(0,\,1,\,1,\,f_{(i)})\}, \quad \forall i \in [t+1,\,n-2]\,. &(2.20)\end{aligned}$$

In the above equations, $a_{(i)},\, b_{(i)},\, c_{(i)},\, d_{(i)},\, e_{(i)},\, f_{(i)} \in [0,\,1]$. Note that only (2.19) is relevant if $t = n-2$. The fact that we are able to extract only the first three coordinates of the elements of G_i, $\forall i \in [0,\, n-2]$, can be proved using the following two auxiliary lemmas, the proofs of which are given in Appendix A.1.

Lemma 2.6 *For all* $(0,\,\beta,\,\tilde{\gamma}) \in \tilde{D}$, $\tilde{\gamma}_{(i)} = 0$, $\forall\, i \in [0,\,t]$.

Lemma 2.7 *For all* $i \in [t+1,\, n-1]$ *there exists* $(0,\,\beta,\,\tilde{\gamma}) \in \tilde{D}$ *with* $\tilde{\gamma}_{(i)} = 1$.

However, only the first three coordinates of the elements of the G_i's are sufficient to determine the S_i's (S_i is defined in Sect. 2.2.3). It is easy to verify from Table 2.1 that $S_i = 2$, $\forall i \in [0,\,t]$ and $S_i = 1$, $\forall i \in [t+1,\, n-2]$. From Proposition 2.3

$$|\tilde{D}\text{-consistent}| = S = 4 \cdot \prod_{i=0}^{n-2} S_i = 4 \cdot \underbrace{1 \cdot 1 \cdots 1}_{(n-t-2)\text{ times}} \cdot \underbrace{2 \cdot 2 \cdots 2}_{(t+1)\text{ times}} = 2^{t+3}\,.$$

(ii) The proof is similar to the above one using Proposition 2.3. □

Theorem 2.8 *Let the* total useful set $\tilde{D}$ *be given for the equation*

$$(x + y) \oplus ((x \oplus \alpha) + (y \oplus \beta)) = \gamma$$

with $x,\ y,\ \alpha,\ \beta,\ \gamma \in \mathbb{Z}_2^n$. *Then* $|\tilde{D}$*-consistent*$|$=4.

Proof. Our approach is the same as that of Theorem 2.5.

Case 1 When $n \geq 2$. Corresponding to the *total useful set* $\tilde{D}$ we determine $G_i,\ \forall\, i \in [0,\ n-2]$.

$$\begin{aligned} G_0 &= \{(0,\ 0,\ 0,\ a_{(0)}),\ (0,\ 1,\ 0,\ b_{(0)}),\ (1,\ 0,\ 0,\ c_{(0)}),\ (1,\ 1,\ 0,\ d_{(0)})\}, & (2.21)\\ G_i &= \{(0,\ 0,\ 0,\ e_{(i)}),\ (0,\ 0,\ 1,\ f_{(i)}),\ (0,\ 1,\ 0,\ g_{(i)}),\ (0,\ 1,\ 1,\ h_{(i)}), \\ &\quad (1,\ 0,\ 0,\ m_{(i)}),\ (1,\ 0,\ 1,\ n_{(i)}),\ (1,\ 1,\ 0,\ p_{(i)}),\ (1,\ 1,\ 1,\ q_{(i)})\}, \\ &\quad \forall\, i \in [1,\ n-2]\,. & (2.22)\end{aligned}$$

In the equations $a_{(0)},\ b_{(0)},\ c_{(0)},\ d_{(0)},\ e_{(i)},\ f_{(i)},\ g_{(i)},\ h_{(i)},\ m_{(i)},\ n_{(i)},\ p_{(i)},\ q_{(i)} \in [0,\ 1]$. From Table 2.1 we see that $S_i = 1,\ \forall\, i \in [0,\ n-2]$ (see Sect. 2.2.3 to compute S_i from G_i). Therefore, from Proposition 2.3

$$|\tilde{D}\text{-consistent}| = S = 4 \cdot \prod_{i=0}^{n-2} S_i = 4 \cdot \underbrace{1 \cdot 1 \cdots 1}_{(n-1)\,\text{times}} = 4\,.$$

Case 2 When $n = 1$. If $n = 1$ then $|\tilde{D}$-consistent$| = 4$. The proof is trivial using Proposition 2.3. □

2.3.4 Worst Case Lower Bounds on the Number of Queries

Our target is to design an algorithm (for (2.16) or (2.17)) which computes $\tilde{D}$-consistent for all *seeds* $(x,\ y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$ with adaptive queries. For such an algorithm, the number of required queries may vary with the choice of $(x,\ y)$. In this section we concentrate on a lower bound on the number of queries in the worst case of $(x,\ y)$ under the "rules of the game" stated in Sect. 2.3.2. The significance of the lower bound is that there exists no algorithm that requires less queries in the worst case than the obtained lower bound.

We already noticed that more queries tend to reduce the search space of the secret $(x,\ y)$. In our formal framework, if $A \subseteq B \subseteq \tilde{D}$ then $\tilde{D}$-consistent $\subseteq B$-consistent $\subseteq A$-consistent. This implies that $|\tilde{D}$-consistent$| \leq |B$-consistent$| \leq$

$|A$-consistent$|$. Note that our algorithm constructs $A \subseteq \tilde{D}$, $\forall(x,\, y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$, using the submitted queries and the corresponding outputs such that $|\tilde{D}$-consistent$| = |A$-consistent$|$. The algorithm fails if $|\tilde{D}$-consistent$| < |A$-consistent$|$, for some $(x,\, y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$. We will use the condition – $|A$-consistent$|$ cannot be *strictly* greater than $|\tilde{D}$-consistent$|$ – to compute a lower bound on the number of queries in the worst case. Similar to the previous section, in Theorem 2.9 we identify a property of A, in terms of the ith core G_i, where the above condition is violated. In Theorem 2.10, we use this fact to obtain a lower bound.

Theorem 2.9 *We consider the equation*

$$(x + y) \oplus (x + (y \oplus \beta)) = \gamma\,,$$

where the position of the least significant '1' of x is t with $n-3 \geq t \geq 0$. Let $\phi \subset A \subseteq \tilde{D}$ and $\tilde{D}$ be the total useful set *of the equation. Let the $(n-3)$th core G_{n-3} contain no element $(0,\, \beta_{(n-3)},\, \tilde{\gamma}_{(n-3)},\, \tilde{\gamma}_{(n-2)})$ with $\tilde{\gamma}_{(n-2)} = 1$.*[3] *Then $|A$-consistent$| = 2^{t+3+k}$ for some $k > 0$.*

Proof. If G_{n-3} contains no element $(0,\, \beta_{(n-3)},\, \tilde{\gamma}_{(n-3)},\, \tilde{\gamma}_{(n-2)})$ with $\tilde{\gamma}_{(n-2)} = 1$ then G_{n-2} contains no element $(0,\, \beta_{(n-2)},\, \tilde{\gamma}_{(n-2)},\, \tilde{\gamma}_{(n-1)})$ with $\tilde{\gamma}_{(n-2)} = 1$. Therefore, G_{n-2} is of one of the following forms,

$$G_{n-2} = \{(0,\, 0,\, 0,\, a)\} \quad \text{or } \{(0,\, 0,\, 0,\, a),\, (0,\, 1,\, 0,\, b)\}\,.$$

Now, from Table 2.1, $S_{n-2} = 2^l$ for either of the cases, where $l > 0$. Similarly, using Theorem 2.5, $S_i \geq 2$, $\forall\, i \in [0,\, t]$. Also $S_i \geq 1$, $\forall\, i \in [t+1,\, n-3]$ (when $n-4 \geq t$). Therefore, from Proposition 2.3, $|A$-consistent$| = 2^{t+3+k}$ for some $k > 0$. □

In the following theorem, we partition the entire seed space $\mathbb{Z}_2^n \times \mathbb{Z}_2^n$ and compute a worst case lower bound for each partition. Note that a lower bound (say, l) for any partition shows that, for any algorithm that computes $\tilde{D}$-consistent $\forall(x,\, y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$, there exists at least one seed in that particular partition which requires at least l queries.

Theorem 2.10 *A lower bound on the number of queries $(0,\, \beta)$ to solve*

$$(x + y) \oplus (x + (y \oplus \beta)) = \gamma$$

in the worst case of $(x,\, y)$ is
(i) 0 if $n = 1$,
(ii) 1 if $x = 0$ and $n > 1$,

[3] This implies that there is no oracle output $\tilde{\gamma}$ with $\tilde{\gamma}_{(n-2)} = 1$.

(iii) 1 if $n = 1 + t$ *with* $t > 0$,
(iv) $(n - t - 1)$ *if* $n - 2 \geq t \geq 0$,
where n *is the bit-length of* x, y *and* t *is the position of the least significant '1' of* x.

Proof. For *(i)*, *(ii)* and *(iii)* the lower bounds are trivial.
(iv) $n-2 \geq t \geq 0$. Note that the submission of queries, generation of the *useful set* A and the *core* G_i are the parts of an evolving process (i.e., with every submitted query $(0, \beta)$, A, G_i change). At any time, from the already submitted queries and the outputs, we can always construct $A \subseteq \tilde{D}$ and the ith core G_i, $\forall i \in [0, n-2]$ where $\tilde{D}$ is the *total useful set*. Note that A-consistent contains all the solutions of the equations derived from the already submitted queries. We first divide the case into four disjoint subcases.
Case 1 When $n - 5 \geq t \geq 0$. By Theorem 2.9, a *necessary* condition is that $\exists (0, \beta_{(n-3)}, \tilde{\gamma}_{(n-3)}, \tilde{\gamma}_{(n-2)} = 1) \in G_{n-3}$ otherwise $|A\text{-consistent}| = 2^{t+3+c} > |\tilde{D}\text{-consistent}| = 2^{t+3}$ (invoke Theorem 2.5).

We now define a quantity $V_{n,\,t}$ which denotes the set of all $(x, y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$ such that the position of the least significant '1' of x is t ($V_{n,\,t}$ is a partition). We now define another quantity $l_{(n,\,t)}(k)$ for all $k \in [t+1, n-2]$. Let $l_{(n,\,t)}(k)$ denote a lower bound on the number of *adaptively chosen queries* to have $(0, \beta_{(i)}, \tilde{\gamma}_{(i)}, \tilde{\gamma}_{(i+1)} = 1) \in G_i$ for some $i \in [k, n-2]$ in the worst case of $(x, y) \in V_{n,t}$. In other words, a worst case lower bound $l_{(n,\,t)}(k)$ means that, for any algorithm $\mathcal{A}$ there exists a seed $(x, y) \in V_{n,\,t}$ such that the *adaptively chosen sequence* of $l_{(n,\,t)}(k) - 1$ queries ($l_{(n,\,t)}(k) > 0$) by $\mathcal{A}$ for the seed (x, y) produces oracle outputs $\tilde{\gamma}$'s with $\tilde{\gamma}_{(i+1)} = 0, \forall i \in [k, n-2]$). We will determine $l_{(n,\,t)}(n-3)$ which is a lower bound in this case. For easy reading, in the rest of the proof, we shall denote $l_{(n,\,t)}(k)$ by $l(k)$.

Let $p \in [t+1, n-3]$. Now, for each algorithm, we *always* identify a seed $(x, y) \in V_{n,\,t}$ such that the *adaptively chosen sequence* of $l(p)-1$ queries produces $\tilde{\gamma}$'s with $\tilde{\gamma}_{(i+1)} = 0, \forall i \in [p, n-2]$. From Table 2.1, we construct (a, b), $(a', b') \in V_{n,\,t}$ for *each* (x, y) in the following fashion. The carry bit at the jth position of $(a+b)$ is $c_{(j)}$ and similarly $c'_{(j)}$.

1. (Construction of a and b) $a_{(i)} = x_{(i)}$ and $b_{(i)} = y_{(i)}$, $\forall i \in [0, p]$. If $c_{(i)} = 0$ set $a_{(i)} = 0$, $b_{(i)} = 0$, $\forall i \in [p+1, n-1]$. If $c_{(i)} = 1$ set $a_{(i)} = 1$, $b_{(i)} = 1$, $\forall i \in [p+1, n-1]$.

2. (Construction of a' and b') $a'_{(i)} = x_{(i)}$ and $b'_{(i)} = y_{(i)}$, $\forall i \in [0, p]$. If $c'_{(i)} = 0$ set $a'_{(i)} = 0$, $b'_{(i)} = 1$, $\forall i \in [p+1, n-1]$. If $c'_{(i)} = 1$ set $a'_{(i)} = 1$, $b'_{(i)} = 0$, $\forall i \in [p+1, n-1]$.

We observe from the portion of Table 2.1 – cut off by the rows R(1) and R(3) and the columns Col(1), Col(2), Col(3) and Col(4) – that the seeds (a, b) and

(a', b') produce the same sequence of oracle outputs as (x, y) does on the selected sequence of $l(p) - 1$ queries.

Now we consider each possible $l(p)$th query $(0, \beta)$ and its output $\tilde{\gamma}$ for the seed (x, y). If $(\beta_{(p+1)}, \tilde{\gamma}_{(p+1)}) = (0, 1)$ then (a, b) produces $\tilde{\gamma}_{(i+1)} = 0, \forall i \in [p+1, n-2]$. Similarly, if $(\beta_{(p+1)}, \tilde{\gamma}_{(p+1)}) = (1, 1)$ then (a', b') produces $\tilde{\gamma}_{(i+1)} = 0$, $\forall i \in [p+1, n-2]$. If $(\beta_{(p+1)}, \tilde{\gamma}_{(p+1)}) = (0, 0)$ or $(1, 0)$ then both (a, b) and (a', b') produce $\tilde{\gamma}_{(i+1)} = 0, \forall i \in [p+1, n-2]$. Therefore, either (a, b) or (a', b') produces oracle outputs with $\tilde{\gamma}_{(i+1)} = 0, \forall i \in [p+1, n-2]$, for all the chosen $l(p)$ queries.

Thus, we establish that, for any algorithm there exists a seed $(x, y) \in V_{n,\,t}$ such that the *adaptively chosen* sequence of $l(p)$ queries produces oracle outputs $\tilde{\gamma}$'s with $\tilde{\gamma}_{(i+1)} = 0, \forall i \in [p+1, n-2]$. Therefore, a lower bound on the number of queries such that $\exists (0, \beta_{(i)}, \tilde{\gamma}_{(i)}, \tilde{\gamma}_{(i+1)} = 1) \in G_i$ for some $i \in [p+1, n-2]$ in the worst case of $(x, y) \in V_{n,\,t}$ is $l(p) + 1$. Therefore,

$$l(p+1) = l(p) + 1\,.$$

Following the recursion,

$$l(n-3) = n - t - 4 + l(t+1)\,. \tag{2.23}$$

The following lemma computes a value of $l(t+1)$. See Appendix A.2 for an elaborate proof.

Lemma 2.11 *Let $n - 4 \geq t \geq 0$. For any algorithm $\mathcal{A}$ there exists a seed $(x, y) \in V_{n,\,t}$ such that the adaptively selected sequence of two queries by $\mathcal{A}$ for that particular seed produces oracle outputs $\tilde{\gamma}$'s with $\tilde{\gamma}_{(i+1)} = 0, \forall i \in [t+1, n-2]$.*

From Lemma 2.11, $l(t+1) = 3$. Therefore, from (2.23), $l(n-3) = n - t - 1$.
Case 2 When $n = t + 4$. A worst case lower bound is 3. The proof follows from Lemma 2.11.
Case 3 When $n = t + 3$. A worst case lower bound is 2. From Table 2.1, it is clear that, with only one query $S_{n-2} > 1$; that makes the number of solutions for this case greater than 2^{t+3} which is impossible from Theorem 2.5.
Case 4 When $n = t + 2$. A worst case lower bound is 1 which is trivial. $\square$

Theorem 2.12 *A lower bound on the number of queries (α, β) to solve*

$$(x + y) \oplus ((x \oplus \alpha) + (y \oplus \beta)) = \gamma$$

in the worst case of $(x, y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$ is
(i) 3 if $n > 2$,
(ii) 2 if $n = 2$,
(iii) 0 if $n = 1$.

Proof. *(i)* $n > 2$. Let the first two queries and the corresponding oracle outputs be (α, β), (α', β'), $\tilde{\gamma}$ and $\tilde{\gamma}'$. Depending on the two least significant bits of α, β, α' and β', the oracle returns outputs (i.e., $\tilde{\gamma}$ and $\tilde{\gamma}'$) according to the following rules.

1. If $(\alpha_{(0)}, \beta_{(0)}) = (0, 0)$ then $\tilde{\gamma} = (0, 0, \cdots, 0)_n$.
2. If $(\alpha_{(0)}, \beta_{(0)}) \neq (0, 0)$ and $(\alpha_{(1)}, \beta_{(1)}) = (1, 1)$ then $\tilde{\gamma} = (1, 1, \cdots, 1, 0)_n$.
3. If $(\alpha_{(0)}, \beta_{(0)}) \neq (0, 0)$ and $(\alpha_{(1)}, \beta_{(1)}) \neq (1, 1)$ then $\tilde{\gamma} = (0, 0, \cdots, 0)_n$.
4. If $(\alpha_{(0)}, \beta_{(0)}) = (\alpha'_{(0)}, \beta'_{(0)})$ then $\tilde{\gamma} = \tilde{\gamma}'$.
5. If $(\alpha_{(0)}, \beta_{(0)}) \neq (\alpha'_{(0)}, \beta'_{(0)}) = (0, 0)$ then $\tilde{\gamma}' = (0, 0, \cdots, 0)_n$.
6. If $(\alpha_{(0)}, \beta_{(0)}) \neq (\alpha'_{(0)}, \beta'_{(0)}) \neq (0, 0)$ and $(\alpha'_{(1)}, \beta'_{(1)}) = (0, 0)$ then $\tilde{\gamma}' = (0, 0, \cdots, 0)_n$.
7. If $(\alpha_{(0)}, \beta_{(0)}) \neq (\alpha'_{(0)}, \beta'_{(0)}) \neq (0, 0)$ and $(\alpha'_{(1)}, \beta'_{(1)}) = (1, 1)$ then $\tilde{\gamma}' = (1, 1, \cdots, 1, 0)_n$.
8. If $(\alpha_{(0)}, \beta_{(0)}) \neq (\alpha'_{(0)}, \beta'_{(0)}) \neq (0, 0)$, $(\alpha'_{(1)}, \beta'_{(1)}) \in \{(0, 1), (1, 0)\}$ and $(\alpha_{(1)}, \beta_{(1)}) = (\alpha'_{(1)}, \beta'_{(1)})$ then $\tilde{\gamma}' = (0, 0, \cdots, 0)_n$.
9. If $(\alpha_{(0)}, \beta_{(0)}) \neq (\alpha'_{(0)}, \beta'_{(0)}) \neq (0, 0)$, $(\alpha'_{(1)}, \beta'_{(1)}) \in \{(0, 1), (1, 0)\}$ and $(\alpha_{(1)}, \beta_{(1)}) \neq (\alpha'_{(1)}, \beta'_{(1)})$ then $\tilde{\gamma}'_{(0)} = 0$ and $\tilde{\gamma}'_{(i)} = 1 \oplus \tilde{\gamma}_{(i)}$ for all $i \in [1, n-1]$.

Observing the oracle outputs produced according to the above rules on the first two queries, it is seen from Table 2.1 that one of the following cases occurs.

1. $S_0 \geq 2$ and $S_i \geq 1\ \forall\, i \in [0, n-2]$.
2. $S_1 \geq 2$ and $S_i \geq 1\ \forall\, i \in [0, n-2]$.
3. $S_0 \geq 2$, $S_1 \geq 2$ and $S_i \geq 1\ \forall\, i \in [0, n-2]$.

Clearly, for any of the above cases, the number of valid solutions S, derived from the results of the queries, is *at least* 8 which is not the case with this equation (see Theorem 2.8). Therefore, a lower bound on the number of queries in the worst case is 3.
(ii) $n = 2$. Using Table 2.1, a proof is similar to the proof for *(i)*.
(iii) $n = 1$. A proof is trivial. □

2.3.5 Optimal Algorithms

In this section, we concentrate on designing algorithms that solve (2.17) and (2.16) in the adaptive query model. In the formal framework, if the oracle is seeded with an unknown (x, y) then there is always a *total useful set* $\tilde{D}$ that the unknown seed (x, y) generates. For a particular *total useful set* $\tilde{D}$, there exists a set $\tilde{D}$-consistent containing all values of (x, y). Our algorithm makes adaptive queries (α, β) to the oracle – which is already seeded with a fixed (x, y) – and the oracle returns $\tilde{\gamma}$. The task of the algorithm is to compute $\tilde{D}$-consistent using oracle outputs $\tilde{\gamma}$, for all seeds $(x, y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$. The algorithm is *optimal* if the number of queries in the worst case (i.e., the upper bound) matches the lower bound derived in the relevant theorem (Theorem 2.10 or 2.12). Optimal algorithms to solve (2.17) and (2.16) are presented in Algorithm 2 and 3. Below we discuss the intuition behind the algorithms; while we omit many trivial details, the pseudocode covers all cases.

Discussion: Algorithm 2. The inputs to Algorithm 2 are an oracle O (which is set with the unknown seed and computes $\tilde{\gamma}$), n denoting the size of the unknowns and the precomputed Table 2.1 denoted by the variable T. The output of Algorithm 2 is a set of lists. This set contains the L_i's – the ith bit solution for all $i \in [0, n-2]$ – similar to the ones computed in Sect. 2.2.5. The objective of Algorithm 2 is to correctly compute the L_i's for the *useful set* $\tilde{D}$. The computed L_i's will be given to Algorithm 1 which computes all the individual solutions.

Algorithm 2 works in a way that at first G_i's are computed from the submitted queries such that the number of solutions for them is the same as that for all 2^n queries. Then the L_i's are computed from the G_i's (Step 28). First, we find t, the position of the least significant '1' of x, by making the first query $(0, \beta)$ with β composed of all 1's (Step 3, 10). Once the position of the least significant '1' of x is known, we can immediately compute G_i, $\forall i \in [0, t]$, by invoking Lemma 2.6 (Step 11 to 13). We see that $S_i = 2$, $\forall i \in [0, t]$. This result readily determines that $S_i = 1$, $\forall i \in [t+1, n-1]$ (see Theorem 2.5). To compute G_i, $\forall i \in [t+1, n-2]$ we will use a nice pattern observable in Table 2.1. Consider Col(3) and Col(4) of the Table 2.1. Observe no two rows, contained in these two columns, are identical. Therefore, if, for some queries, G_i contains two elements of the form $(0, 1, 0, a)$ and $(0, 1, 1, b)$ then the corresponding $S_i = 1$. Therefore, the aim of Algorithm 2 is to submit queries such that G_i contains elements of the form $(0, 1, 0, a)$ and $(0, 1, 1, b)$, for all $i \in [t+1, n-2]$. Another interesting pattern of Table 2.1 is that each row, cut off by Col(2), Col(3) and Col(4), has at least one '1'. The algorithm uses this fact to get oracle output $\tilde{\gamma}$ with $\tilde{\gamma}_{(i)} = 1$ if required (Step 25). However, it is always easy to get an oracle output $\tilde{\gamma}$ with $\tilde{\gamma}_{(i)} = 0$ by setting the least significant i bits of β to zero (Step 15 and 21). Note that the algorithm submits the first two queries in Steps 3 and 15. In the loop (from steps 19 to 27) one of the two previous queries is modified to get '0' or

Algorithm 2 Optimal Algorithm to solve the equation $(x+y)\oplus(x+(y\oplus\beta)) = \gamma$

Input: Oracle O, n, Table T
Output: a set of lists

1: If $n \leq 0$ then exit with a comment "Invalid Input";
2: If $n == 1$ then return an empty set ϕ indicating that all solutions are possible and then exit;
3: $\beta = (1, 1, \cdots, 1, 1)_n$; /*The first query*/
4: $\tilde{\gamma} = O(\beta)$; /*Oracle output*/
5: if $\tilde{\gamma} == 0$
6:     For all $i \in [0, n-2]$
7:         $G_i = \{(0, 0, 0, 0), (0, 1, 0, 0)\}$;
8:     Go to Step 28;
9: $t =$ Least-Significant-one$(\tilde{\gamma})$;/*Computing the least significant '1' of the oracle output*/
10: $t = t - 1$; /*Computing the least significant '1' of x*/
11: For all $i \in [0, t-1]$ /*Computing G_i for all $i \in [0, t-1]$*/
12:     $G_i = \{(0, 0, 0, 0), (0, 1, 0, 0)\}$;
13: $G_t = \{(0, 0, 0, 0), (0, 1, 0, 1)\}$; /*Computing G_t*/
14: If $t == n-2$ then Go to Step 28;
15: $\beta' = (1, 1, \cdots, 1, \beta'_{(t+1)} = 1, 0, 0, \cdots, 0)$; /*Query to make $\tilde{\gamma}'_{(t+1)} = 0$*/
16: $\tilde{\gamma}' = O(\beta')$; /*Oracle output*/
17: $G_{t+1} = \{(0, 1, 1, \tilde{\gamma}_{(t+2)}), (0, 1, 0, \tilde{\gamma}'_{(t+2)})\}$; /*Computing G_{t+1}*/
18: If $t == n-3$ then Go to 28;
19: For all $i \in [2, n-t-2]$, in increasing order /*Computing G_i for all $i \in [t+2, n-2]$*/
20:     If $\tilde{\gamma}_{(t+i)} == \tilde{\gamma}'_{(t+i)} == 1$ then
21:         $\beta' = (1, 1, \cdots, 1, \beta'_{(t+i-1)} = 0, 0, \cdots, 0)$;/*Query to make $\tilde{\gamma}'_{(t+i)} = 0$*/
22:         $\tilde{\gamma}' = O(\beta')$ and Go to Step 27; /*Oracle output*/
23:     If $\tilde{\gamma}_{(t+i)} == \tilde{\gamma}'_{(t+i)} == 0$
24:         if $\tilde{\gamma}_{(t+i-1)} == 1$ then swap $((\beta, \tilde{\gamma}), (\beta', \tilde{\gamma}'))$;
25:         $\beta' = (1, 1, \cdots, 1, \beta'_{(t+i-1)} = 0, \beta'_{(t+i-2)}, \cdots, \beta'_{(0)})$; /*Query to make $\tilde{\gamma}'_{(t+i)} = 1$*/
26:         $\tilde{\gamma}' = O(\beta')$; /*Oracle output*/
27:     $G_{t+i} = \{(0, 1, \tilde{\gamma}_{(t+i)}, \tilde{\gamma}_{(t+i+1)}), (0, 1, \tilde{\gamma}'_{(t+i)}, \tilde{\gamma}'_{(t+i+1)})\}$;
28: $L_i = \{(x_{(i)}, y_{(i)}, c_{(i)}) \,|\, G_i \rightarrow (x_{(i)}, y_{(i)}, c_{(i)})\}$, $\forall i \in [0, n-2]$ (Using the table T); /*Computing L_i's*/
29: Return the set $\{L_i | i \in [0, n-2]\}$;

'1' at the required locations of the output. Thus we get G_i, $\forall i \in [t+1, n-2]$, for which $S_i = 1$. Therefore, invoking Theorem 2.5, we get the number solutions for the submitted queries is 2^{t+3} (omitting many trivial cases). This proves the correctness of Algorithm 2. It can be easily verified from Theorem 2.10 that the number of queries used by Algorithm 2 is worst case optimal.

Discussion: Algorithm 3. The basic idea of Algorithm 3 is the same as that of Algorithm 2, i.e., we compute the L_i's for the *total useful set* $\tilde{D}$. But here we use a different trick to optimize the number of queries. Note that, for all $i \in [0, n-2]$, $S_i = 1$ for (2.16) (see Theorem 2.8).

We first submit two queries as shown in Step 3 and 5. For these two queries $G_i = \{(1, 0, \tilde{\gamma}_{(i)}, \tilde{\gamma}_{(i+1)}), (0, 1, \tilde{\gamma}'_{(i)}, \tilde{\gamma}'_{(i+1)})\}$ if i even. Note that, in this case, if $\tilde{\gamma}_{(i)} = \tilde{\gamma}'_{(i)}$ then $S_i = 1$ otherwise $S_i = 2$ (from Table 2.1). $G_i = \{(1, 0, \tilde{\gamma}_{(i)}, \tilde{\gamma}_{(i+1)}), (1, 0, \tilde{\gamma}'_{(i)}, \tilde{\gamma}'_{(i+1)})\}$ if i odd. Note that, in this case, if $\tilde{\gamma}_{(i)} \neq \tilde{\gamma}'_{(i)}$ then $S_i = 1$ otherwise $S_i = 2$. Observe that $S_i = 2 \Leftrightarrow |L_i| = 4$ and $S_i = 1 \Leftrightarrow |L_i| = 2$. Now if $|L_i| = 2$, $\forall i \in [0, n-2]$ then we are done (Step 11). A combinatorial pattern in the precomputed Table 2.1 shows that, if $|L_i| = 4$ then $|L_{i-1}| = 2$ (omitting the proof). In Step 13 and 16, we change the second query (c, d) to obtain the third query so that the new query can remove extra elements from the L_i's with $|L_i| = 4$. The rule is if $|L_i| = 4$ then change the $(i-1)$th bits of (c, d) "suitably" (details in the pseudocode). In Steps 13 and 16, the relation operator "$\rightarrow$" is used in the same sense as in Sect. 2.2.3. Now $L_i's$ are computed from the first and the third queries (Step 21). If $|L_i| = 4$ then we assign $L_i = L'_i$ (Step 23). Now we get $|L_i| = 2$, $\forall i \in [0, n-2]$. Algorithm 3 is *optimal* because it uses maximum 3 queries.

Time and Memory. For each of Algorithm 2 and 3, the memory and the time are $\Theta(n)$ and $\mathcal{O}(n)$ respectively (the oracle takes $\mathcal{O}(1)$-time to compute $\tilde{\gamma}$).

2.4 Solving DEA with Batch Queries

In Sect. 2.3, the following two *differential equations of addition* (DEA) over $\mathbb{Z}_2^n$ have been solved in adaptive query model.

$$(x + y) \oplus ((x \oplus \alpha) + (y \oplus \beta)) = \gamma, \tag{2.24}$$

$$(x + y) \oplus (x + (y \oplus \beta)) = \gamma\,, \tag{2.25}$$

where x, y are the *only fixed* unknown variables. In Sect. 2.1, we have already provided the motivation for solving the above equations with a set of *batch queries* (i.e., queries which are submitted in a batch rather than adaptively). In the following discussion, we follow an approach, similar to the one in Sect. 2.3 which deals with adaptive queries, to solve the above two DEA with *batch queries*. What

Algorithm 3 Optimal algorithm solving $(x + y) \oplus ((x \oplus \alpha) + (y \oplus \beta)) = \gamma$

Input: Oracle O, n, Table T
Output: a set of lists

1: If $n \leq 0$ then exit with a comment "Invalid Input";
2: If $n == 1$ then return an empty set ϕ indicating that all solutions are possible and then exit;
3: $(a, b) = ((11 \cdots 11)_n, (00 \cdots 00)_n)$;/*First Query*/
4: $\tilde{\gamma} = O(a, b)$; /*Oracle Output*/
5: $(c, d) = ((\cdots 101010)_n, (\cdots 010101)_n)$;/*Second Query*/
6: $\tilde{\gamma}' = O(c, d)$; /*Oracle Output*/
7: For all $i \in [0, n-2]$ /*Computing G_i's*/
8: $\quad G_i = \{(a_{(i)}, b_{(i)}, \tilde{\gamma}_{(i)}, \tilde{\gamma}_{(i+1)}), (c_{(i)}, d_{(i)}, \tilde{\gamma}'_{(i)}, \tilde{\gamma}'_{(i+1)})\}$;
9: For all $i \in [0, n-2]$ /*Computing L_i's*/
10: $\quad L_i = \{(x_{(i)}, y_{(i)}, c_{(i)}) \,|\, G_i \to (x_{(i)}, y_{(i)}, c_{(i)})\}, \forall i \in [0, n-2]$ (Using the table T);
11: If $|L_i| == 2, \forall i \in [0, n-2]$ then Go to step 25;
12: For all $i \in [0, n-2]$ and i even /*Checking even L_i's*/
13: $\quad$ If $|L_i| == 4$ then collect $(x_{(i-1)}, y_{(i-1)}, 0) \in L_{i-1}$ and $(1, 0, \tilde{\gamma}_{(i-1)}, \tilde{\gamma}_{(i)}) \in G_{i-1}$.
$\quad$ Select $(\alpha_{(i-1)}, \beta_{(i-1)})$ from the table T such that, $\{(\alpha_{(i-1)}, \beta_{(i-1)}, 0, \tilde{\gamma}_{(i)}), (\alpha_{(i-1)}, \beta_{(i-1)}, 1, \tilde{\gamma}_{(i)})\} \to (x_{(i-1)}, y_{(i-1)}, 0)$;
14: $\quad (c_{(i-1)}, d_{(i-1)}) = (\alpha_{(i-1)}, \beta_{(i-1)})$;
15: For all $i \in [0, n-2]$ and i odd /*Checking odd L_i's*/
16: $\quad$ If $|L_i| == 4$ then collect $(x_{(i-1)}, y_{(i-1)}, 0) \in L_{i-1}$ and $(1, 0, \tilde{\gamma}_{(i-1)}, \tilde{\gamma}_{(i)}) \in G_{i-1}$.
$\quad$ Select $(\alpha_{(i-1)}, \beta_{(i-1)})$ from T such that $\{(\alpha_{(i-1)}, \beta_{(i-1)}, 0, 1 \oplus \tilde{\gamma}_{(i)}), (\alpha_{(i-1)}, \beta_{(i-1)}, 1, 1 \oplus \tilde{\gamma}_{(i)})\}$ $\to (x_{(i-1)}, y_{(i-1)}, 0)$;
17: $\quad (c_{(i-1)}, d_{(i-1)}) = (\alpha_{(i-1)}, \beta_{(i-1)})$;
18: $\tilde{\gamma}' = O(c, d)$;/*Oracle Output*/
19: For all $i \in [0, n-2]$ /*Computing G_i's*/
20: $\quad G_i = \{(a_{(i)}, b_{(i)}, \tilde{\gamma}_{(i)}, \tilde{\gamma}_{(i+1)}), (c_{(i)}, d_{(i)}, \tilde{\gamma}'_{(i)}, \tilde{\gamma}'_{(i+1)})\}$;
21: For all $i \in [0, n-2]$ /*Computing L'_i's*/
22: $\quad L'_i = \{(x_{(i)}, y_{(i)}, c_{(i)}) \,|\, G_i \to (x_{(i)}, y_{(i)}, c_{(i)})\}, \forall i \in [0, n-2]$ (Using the table T);
23: For each $i \in [0, n-2]$
24: $\quad$ If $|L_i| == 4$, then assign $L_i = L'_i$;
25: Return the set $\{L_i | i \in [0, n-2]\}$;

we essentially do in this section is modify the model of the previous section to suit it for *batch queries*. The problem becomes clearer through the following sections.

2.4.1 The Power of the Adversary

The power of an adversary that solves (2.24) is defined as follows.

1. An adversary has unrestricted computational power and an infinite amount of memory.
2. An adversary submits a *set* of queries $\{(\alpha,\, \beta)\}$ in a *batch*, to an honest oracle[4] which computes the γ's using the fixed unknown $(x,\, y)$ in (2.24) and returns them to the adversary. We will often refer to that fixed $(x,\, y)$ as the *seed* of the oracle.

We recall from Sect. 2.3 that the above oracle with seed $(x,\, y)$ can be viewed as a mapping $O_{xy} : \mathbb{Z}_2^n \times \mathbb{Z}_2^n \to \mathbb{Z}_2^n$ and defined by

$$O_{xy} = \{(\alpha,\, \beta,\, \gamma) \,|\, (\alpha,\, \beta) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n,\, \gamma = (x+y) \oplus ((x \oplus \alpha) + (y \oplus \beta))\}\,. \quad (2.26)$$

An adversarial model, similar to the above, can be constructed for (2.25) by setting $(\alpha,\, \beta) \in \{0\}^n \times \mathbb{Z}_2^n$ and the mapping $O_{xy} \,:\, \{0\}^n \times \mathbb{Z}_2^n \to \mathbb{Z}_2^n$.

The above model represents a practical *chosen message attack* scenario (contrast it with the adaptive query model in Sect. 2.3 which represented an *adaptive chosen message attack* scenario) where the adversary submits queries to an oracle in a batch. Based on the replies from the oracle, the adversary computes one or more unknown parameters.

2.4.2 The Task: the Solution Set $\tilde{D}$-consistent

Let

$$\mathcal{F} = \{O_{xy} \,|\, (x,\, y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n\}.$$

Any $D \in \mathcal{F}$ is called a *character set* (see Sect. 2.2 for a definition). The aim of the adversary is to find all the solutions $(x,\, y)$ satisfying the 2^{2n} equations contained in the *character set* D. Note that the set of all such solutions constitutes the set D-consistent. If we work with (2.25) then $|D| = 2^n$.

Equivalent Task. Like the case with adaptive queries in Sect. 2.3, we apply the following transformation to the *character set* D to compute the *useful set* $\tilde{D}$,

$$\tilde{D} = \{(\alpha,\, \beta,\, \tilde{\gamma} = \alpha \oplus \beta \oplus \gamma) \,|\, (\alpha,\, \beta,\, \gamma) \in D\}\,.$$

[4] An honest oracle correctly computes γ and returns it to the adversary.

Note $|\tilde{D}| = 2^{2n}$ if (2.24) is considered. An element $(\alpha,\ \beta,\ \tilde{\gamma}) \in \tilde{D}$ corresponds to the following equation:

$$(x + y) \oplus ((x \oplus \alpha) + (y \oplus \beta)) \oplus \alpha \oplus \beta = \tilde{\gamma}.$$

Now the set $\tilde{D}$-consistent contains all $(x,\ y)$'s satisfying all 2^{2n} equations corresponding to $\tilde{D}$. We already know that

$$D\text{-consistent} = \tilde{D}\text{-consistent}.$$

Therefore, the task is equivalent to determination of $\tilde{D}$-consistent from a subset of the *useful set* $\tilde{D}$. When we deal with (2.25) then $|\tilde{D}| = 2^n$.

Equivalent Oracle Output. In a way similar to the case with adaptive queries in Sect. 2.3, the oracle output γ on query $(\alpha,\ \beta)$ will be adjusted to $\tilde{\gamma} = \alpha \oplus \beta \oplus \gamma$ for easy understanding of many deductions.

Rules of the Game. Below, we describe the rules followed by the adversary who determines the set $\tilde{D}$-consistent. The essence of the whole problem is brought out in the following points.

1. The adversary starts with no information about x and y except their size n.

2. The adversary submits a set of queries $(\alpha,\ \beta)$'s in a batch, irrespective of the seed $(x,\ y)$. The oracle returns to the adversary the set of $\tilde{\gamma}$'s corresponding to the queries and the chosen $(x,\ y)$.

3. The adversary fails if, with the submitted queries, she is unable to compute $\tilde{D}$-consistent for some $(x,\ y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$.

We search for an algorithm that determines $\tilde{D}$-consistent, for all $(x,\ y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$, with the same set of submitted queries. Furthermore, there is an additional challenge to reduce the number of required queries as much close as possible to the minimum.

2.4.3 Lower Bounds on the Number of Queries

In this section we are mainly interested in lower bounds on the number of queries, submitted in a batch, to solve DEA. Note that the trivial method is to submit all possible queries and then solve them according to the method shown in Sect. 2.2, however, the challenge lies with a *nontrivial* solution which uses fewer queries. It is always regarded as an important theoretical benchmark as to how far it is possible to reduce the number of queries. The significance of a lower bound is that no algorithm can solve the equations with less queries.

As noticed in the previous section, the size of search space of the secret (x, y) decreases with the number of queries. Therefore we search for an algorithm that constructs $A \subseteq \tilde{D}$, $\forall (x, y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$, using the submitted queries and the corresponding outputs such that $|\tilde{D}$-consistent$| = |A$-consistent$|$. The algorithm fails if $|\tilde{D}$-consistent$| < |A$-consistent$|$, for some $(x, y) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$. In Theorem 2.13, which is similar to Theorem 2.9, a property of the set A is identified where the above condition is not satisfied. In Theorem 2.14, we use this fact to obtain a lower bound.

Theorem 2.13 *We consider the equation*

$$(x + y) \oplus (x + (y \oplus \beta)) = \gamma \,,$$

where the position of the least significant '1' of x is t with $n-3 \geq t \geq 0$. Let all the submitted queries and the oracle outputs be stored in the set A (note that $\phi \subset A \subseteq \tilde{D}$ where $\tilde{D}$ is a useful *set). Suppose that there is no query $(0, \beta)$ for which the oracle output is $\tilde{\gamma}$ with $\tilde{\gamma}_{(n-2)} = 1$. Then $|A$-consistent$| > |\tilde{D}$-consistent$|$.*

Proof. If there is no query $(0, \beta)$ for which the oracle output is $\tilde{\gamma}$ with $\tilde{\gamma}_{(n-2)} = 1$ then the $(n-3)$th core G_{n-3} (corresponding to A) contains no element $(0, \beta_{(n-3)}, \tilde{\gamma}_{(n-3)}, \tilde{\gamma}_{(n-2)})$ with $\tilde{\gamma}_{(n-2)} = 1$. This implies G_{n-2} contains no element $(0, \beta_{(n-2)}, \tilde{\gamma}_{(n-2)}, \tilde{\gamma}_{(n-1)})$ with $\tilde{\gamma}_{(n-2)} = 1$. Therefore, G_{n-2} is of one of the following forms,

$$G_{n-2} = \{(0,\, 0,\, 0,\, 0)\} \quad \text{or } \{(0,\, 0,\, 0,\, 0),\, (0,\, 1,\, 0,\, b)\}\,.$$

Now, from Table 2.1, $S_{n-2} = 2^l$ for either of the cases, where $l > 0$. Similarly, using Theorem 2.5, $S_i \geq 2$, $\forall\, i \in [0, t]$. Also $S_i \geq 1$, $\forall\, i \in [t+1, n-3]$ (when $n-4 \geq t$). Therefore, from Proposition 2.3, $|A$-consistent$| = 2^{t+3+k}$ for some $k > 0$. From Theorem 2.5, $|\tilde{D}$-consistent$| = 2^{t+3}$. Therefore, $|A$-consistent$| > |\tilde{D}$-consistent$|$. □

Instead of establishing a lower bound for the entire seed space $\mathbb{Z}_2^n \times \mathbb{Z}_2^n$, we derive a lower bound for a subset of $\mathbb{Z}_2^n \times \mathbb{Z}_2^n$, denoted by $V_{n,\,0}$ which is the collection of all (x, y)'s with $x_{(0)} = 1$ (that is, the position of the least significant '1' of x is zero). Note that $|V_{n,\,0}| = 2^{2n-1}$. As $V_{n,\,0}$ is a proper subset of $\mathbb{Z}_2^n \times \mathbb{Z}_2^n$, the lower bound derived for $V_{n,\,0}$ is also a lower bound for the entire seed space $\mathbb{Z}_2^n \times \mathbb{Z}_2^n$.

Theorem 2.14 *A lower bound on the number of queries $(0, \beta)$'s, submitted in a batch, to solve*

$$(x + y) \oplus (x + (y \oplus \beta)) = \gamma$$

where $(x, y) \in V_{n,\,0}$ *is (i)* $3 \cdot 2^{n-4}$ *if* $n \geq 4$*, (ii) 2 if* $n = 3$*, (iii) 1 if* $n = 2$ *and (iv) 0 if* $n = 1$.

Proof. *(i)* When $n \geq 4$. Let all the submitted queries and the oracle outputs be stored in the set A ($\phi \subset A \subseteq \tilde{D}$ where $\tilde{D}$ is a *useful* set) and $|A$-consistent$| = |\tilde{D}$-consistent$|$ for all $(x, y) \in V_{n,\,0}$. By Theorem 2.13, a *necessary* condition is that there must exist at least one query $(0, \beta)$ for which the oracle output is $\tilde{\gamma}$ with $\tilde{\gamma}_{(n-2)} = 1$ otherwise $|A$-consistent$| > |\tilde{D}$-consistent$|$. We shall henceforth denote a query $(0, \beta)$ by β.

We first encode the bit-string of a query β as the edges and the corresponding output $\tilde{\gamma}$ as the nodes (denoted by circles) on a path of the full binary tree as shown in Fig. 2.1. The possible values of $\beta_{(i)}$ are denoted as the edges of the tree between the depth i and the depth $(i + 1)$ (the root of the tree is at depth 0). Similarly the possible values of $\tilde{\gamma}_{(i)}$ can be assigned to the nodes at the depth i. Note that all possible 2^n queries are encoded in the tree.

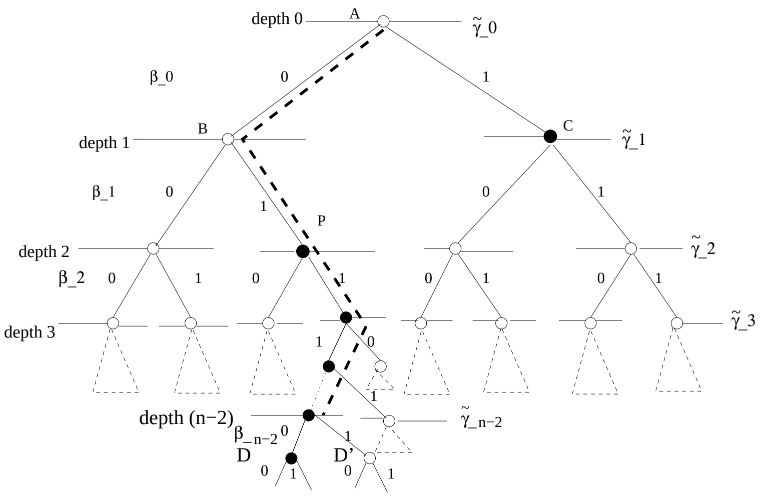

Figure 2.1: An arbitrary path P in the subtree (black node indicates value 1 and white node 0)

The Approach. We shall isolate a subtree and show that, if an arbitrary path in that subtree is *not* present as the prefix of one of the submitted queries then there exists a seed $(x, y) \in V_{n,\,0}$ such that, on all other queries, the outputs are $\tilde{\gamma}$'s with $\tilde{\gamma}_{(n-2)} = 0$. Therefore a lower bound is the number of all paths present in that particular subtree.

Node Assignment Rule. The rule shows how to select the values of $\tilde{\gamma}$'s for all the queries. In the nodes of the *entire* tree we now put the values of $\tilde{\gamma}$. We select an *arbitrary* path P (which is a prefix of a query) in the subtree whose leaf nodes are at the depth $(n-2)$ and whose first two edges are $(0,\ 1)$ and $(1,\ 0)$ and $(1,\ 1)$ (see Fig. 2.1). Note that the values at the nodes B and C will be 0 and 1 respectively because $x_{(0)} = 1$. Now we put 1 in all nodes on the path P from the depth 2 till the depth $(n-2)$. The two child nodes D and D' of the last node on P are assigned 0 and 1 arbitrarily. All other nodes in the tree are assigned 0. The intuition that such an assignment rule gives a valid solution is derived from an observation in Table 2.1 (see Sect. 2.2.2) that the matrix cut off by rows R(1), R(2) and R(3) and columns Col(2), Col(3) and Col(4) has only diagonal elements 1 (formally proved in Lemma 2.15).

Proof Resumed. Suppose P is not a prefix of any query in A. As shown in Fig. 2.1, all nodes at depth $(n-2)$, except the one on the path P, are assigned zero. Therefore, there is no query β in A such that the corresponding output $\tilde{\gamma}$ has $\tilde{\gamma}_{(n-2)} = 1$. This leads to a contradiction. Therefore, there must be a query in A whose prefix is P. Now P is an arbitrary path in the subtree constructed above. Now the total number of paths (or prefixes of queries) in the subtree is $3 \cdot 2^{n-4}$. The following lemma completes the proof.

Lemma 2.15 *For any arbitrary P with the first two edges $(0,\ 1)$ or $(1,\ 0)$ or $(1,\ 1)$ in the tree constructed above, all queries and their outputs encoded in the tree according to the* Node Assignment Rule*, produce a valid solution $(x,\ y) \in V_{n,\,0}$.*

Proof. For any arbitrary P, the core G_i's $(0 \le i \le n-2)$, computed from the values of $\tilde{\gamma}$'s and β's (according to the *Node Assignment Rule*), are of one of the following forms:

$$\begin{aligned} G_0 &= \{(0,0,0,0),\,(0,1,0,0)\}\,, \\ G_i &= \{(0,0,0,0),\,(0,1,0,0),\,(0,0,1,a_{(i)}),\,(0,1,1,b_{(i)})\}\,, \quad 1 \le i \le n-2 \end{aligned}$$

where $a_{(i)},\ b_{(i)} \in [0,\ 1]$ and $a_{(i)} = 1 \oplus b_{(i)}$. Now each $S_i > 0$ (obtained from Table 2.1 using the G_i's). Therefore, the number of valid solutions $S = 4 \cdot \prod_{i=0}^{n-2} S_i > 0$ (see Proposition 2.3). In fact the number solutions is 8 (verification of a part of Theorem 2.5). □

The proofs of *(ii)*, *(iii)* and *(iv)* are immediate from Table 2.1. □

Lower Bound for the equation $(x+y) \oplus ((x \oplus \alpha) + (y \oplus \beta)) = \gamma$. For the equation lower bounds on the number of *batch* queries, for $n = 2$ and 3, are 2 and 3 respectively. This can be proved by searching through all possible x, y, α, β and $\tilde{\gamma}$ exhaustively. However, the situation becomes intractable when $n \geq 4$ when the number of all possible x, y, α, β and $\tilde{\gamma}$ for only 3 queries becomes extremely large (2^{60} for $n = 4$). We did extensive experiments with many test vectors and found that three queries were insufficient to solve the equations when $n \geq 4$. We state the following conjecture.

CONJECTURE 2.16 *A lower bound on the number of queries* (α, β)*, submitted in a batch, to solve*

$$(x + y) \oplus ((x \oplus \alpha) + (y \oplus \beta)) = \gamma$$

is 4 *if* $n \geq 4$*.*

2.4.4 Algorithms

In this section, we present two algorithms Algorithm 4 and Algorithm 5 to solve (2.25) and (2.24) with a set of *batch* queries. The inputs to the algorithms are the *bit-length* n, the oracle O and the Table 2.1. The outputs are the G_i's (defined in Sect. 2.2.3) computed from a set of queries and their replies. Our target is to select a subset of all possible queries such that the set of solutions derived from the G_i's is the same as $\tilde{D}$-consistent, for all (x, y)'s. An algorithm to compute the actual solution set from the set of G_i's is described in Sect. 2.2.5. Below we discuss the motivation and correctness of the algorithms; we leave many smaller details, while the pseudocode covers all cases.

Discussion: Algorithm 4. The number of queries required by the algorithm is 2^{n-2} which is one fourth of all possible 2^n queries (note that a lower bound in $\frac{3}{4} \cdot 2^{n-2}$). The two *for loops* (in steps 8-17 and 10-13) are the most important parts of the algorithm. For easy understanding of the algorithm, let us see how the algorithm works when the position of the least significant '1', of x is zero (i.e., $x_{(0)} = 1$). Note that we have to submit queries such that the G_i's $(0 \leq i \leq n-2)$ obtained from them correspond to $S_0 = 2$ and $S_i = 1\, \forall i \in [1, n-2]$ (see Theorem 2.5). In the tth iteration of the bigger loop we submit a set of queries which ensures that $G_t = \{(0, 1, 1, a), (0, 0, 1, b), (0, 1, 0, c)\}$ which implies that $S_t = 1$. The tth iteration also produces at least one output $\tilde{\gamma}$ with $\tilde{\gamma}_{(t+1)} = 1$ which will be used in the next loop. If there is no output with $\tilde{\gamma}_{(t+1)} = 1$ then $S_{t+1} > 1$ and hence the algorithm fails (see Theorem 2.13). The proof of correctness of the algorithm when $x_{(0)} \neq 1$ is similar to the above argument.

Discussion: Algorithm 5. The number of queries required by the algorithm is 6 which is two more than the best known lower bound (also note that the number of all possible queries is 2^{2n}). The proof of correctness of this algorithm is by showing that the submitted queries produce $S_i = 1$ for all $i \in [0, n-2]$

Algorithm 4 Algorithm to solve the equation $(x + y) \oplus (x + (y \oplus \beta)) = \gamma$ with batch queries

Input: Oracle O, n, Table T
Output: The core G_i's

1: If $n \leq 0$ then exit with a comment "Invalid Input";
2: If $n = 1$ then return an empty set ϕ indicating that all 4 solutions are possible and exit;
3: $\beta = (1,\, 1,\, \cdots,\, 1,\, 1)_n$; /*The first query*/
4: $\tilde{\gamma} = O(\beta)$; /*Oracle output*/
5: $Q = \{\beta\}$ and $A = \{(0,\, \beta,\, \tilde{\gamma})\}$; /*Collecting query and output*/
6: If $n = 2$ then Go to Step 20;
7: If $n > 3$ /*If $n = 3$, the execution automatically jumps to Step 18)*/
8: For all $t \in [1,\, n-3]$ in increasing order
9: $\{$Initialize $Q' = \phi$
10: For all $\beta \in Q$
11: $\{\beta' = (1,\, 1, \cdots, \beta'_{(t)} = 0,\, \beta_{(t-1)}, \cdots, \beta_{(0)})$; /*New query*/
12: $O(\beta') = \tilde{\gamma}'$; /*Oracle output*/
13: $Q' = Q' \cup \{\beta'\}$ and $A = A \cup \{(0, \beta', \tilde{\gamma}')\}$;$\}$
14: $\beta' = (1,\, 1, \cdots, \beta'_{(t)} = 1,\, 0, \cdots, 0)$; /*New query*/
15: $O(\beta') = \tilde{\gamma}'$; /*Oracle output*/
16: $Q' = Q' \cup \{\beta'\}$ and $A = A \cup \{(0, \beta', \tilde{\gamma}')\}$;
17: $Q = Q \cup Q'$;$\}$
18: $\beta' = (1,\, 1, \cdots, \beta'_{(n-2)} = 1,\, 0, \cdots, 0)$; /*Last query*/
19: $O(\beta') = \tilde{\gamma}'$ and $A = A \cup \{(0, \beta', \tilde{\gamma}')\}$; /*Oracle output*/
20: Return the core G_i's for all $i \in [0, n-2]$ computed from A.

(Theorem 2.8). Six queries are submitted in steps 3, 5, 8, 10, 12, 14. Now we consider only the first two queries in steps 3 and 5. For these queries, $G_i = \{(1,\, 0,\, \tilde{\gamma}_{(i)},\, \tilde{\gamma}_{(i+1)}),\, (0,\, 1,\, \tilde{\gamma}'_{(i)},\, \tilde{\gamma}'_{(i+1)})\}$ if i even. Note that, in this case, if $\tilde{\gamma}_{(i)} = \tilde{\gamma}'_{(i)}$ then $S_i = 1$ otherwise $S_i = 2$ (from Table 2.1). Also observe that $G_i = \{(1,\, 0,\, \tilde{\gamma}_{(i)},\, \tilde{\gamma}_{(i+1)}),\, (1,\, 0,\, \tilde{\gamma}'_{(i)},\, \tilde{\gamma}'_{(i+1)})\}$ if i odd. In this case, if $\tilde{\gamma}_{(i)} \neq \tilde{\gamma}'_{(i)}$ then $S_i = 1$ otherwise $S_i = 2$. Next, we observe a combinatorial pattern in the precomputed Table 2.1 which shows that, if $S_i = 2$ then $S_{i-1} = 1$ (proof is direct). The 3rd and the 4th queries are generated from the second query assuming $S_i = 2$ for some odd i's. We change all the even numbered bits of the second query $(\alpha[2],\, \beta[2])$. It can be shown that making $(\alpha[2]_{(i)},\, \beta[2]_{(i)}) = (0,0)$ and $(1,1)$ for all even i's ensures that $S_i = 1$ for all odd i's. Exactly the same way the 5th and the 6th queries are generated from the second query $(\alpha[2],\, \beta[2])$ assuming that $S_i = 2$ for some even i's. Now we change the odd numbered bits of $(\alpha[2],\, \beta[2])$ to $(0,0)$ and $(1,1)$ to ensure that all $S_i = 1$ for all even i's. Therefore, the number of solutions derived from these 6 queries is $S = 4 \cdot \prod_{i=0}^{n-2} S_i = 4$ as suggested in Theorem 2.8.

Algorithm 5 Algorithm to solve the equation $(x + y) \oplus ((x + \alpha) + (y \oplus \beta)) = \gamma$

Input: Oracle O, n, Table T
Output: the core G_i's

1: If $n \leq 0$ then exit with a comment "Invalid Input";
2: If $n = 1$ then return an empty set ϕ indicating that all 4 solutions are possible and exit;
3: $(\alpha[1],\, \beta[1]) = ((11 \cdots 11)_n,\, (00 \cdots 00)_n)$;/*First Query*/
4: $\tilde{\gamma}[1] = O(\alpha[1],\, \beta[1])$; /*Oracle Output*/
5: $(\alpha[2],\, \beta[2]) = ((\cdots 101010)_n,\, (\cdots 010101)_n)$;/*Second Query*/
6: $\tilde{\gamma}[2] = O(\alpha[2],\, \beta[2])$; /*Oracle Output*/
7: If $n = 2$ then Go to Step 16;
8: $(\alpha[3],\, \beta[3]) = ((\cdots 101010)_n,\, (\cdots 000000)_n)$;/*Third Query*/
9: $\tilde{\gamma}[3] = O(\alpha[3],\, \beta[3])$; /*Oracle Output*/
10: $(\alpha[4],\, \beta[4]) = ((\cdots 111111)_n,\, (\cdots 010101)_n)$;/*Fourth Query*/
11: $\tilde{\gamma}[4] = O(\alpha[4],\, \beta[4])$; /*Oracle Output*/
12: $(\alpha[5],\, \beta[5]) = ((\cdots 000000)_n,\, (\cdots 010101)_n)$;/*Fifth Query*/
13: $\tilde{\gamma}[5] = O(\alpha[5],\, \beta[5])$; /*Oracle Output*/
14: $(\alpha[6],\, \beta[6]) = ((\cdots 101010)_n,\, (\cdots 111111)_n)$;/*Sixth Query*/
15: $\tilde{\gamma}[6] = O(\alpha[6],\, \beta[6])$; /*Oracle Output*/
16: $A = \{(\alpha[i],\, \beta[i],\, \tilde{\gamma}[i]) \mid$ for all i's$\}$
17: Return the core G_i's for all $i \in [0, n-2]$ computed from A.

2.5 Cryptographic Applications

In this section we discuss the practical implications of the results derived in the earlier sections of this chapter.

Cryptographic implications of Modular addition under DC. The fact that an arbitrary system of DEA is in the complexity class $\mathcal{P}$, shows a major *differential weakness of modular addition* which is one of the largely used block cipher components. The weakness is more alarming because, with a maximum of *only* 3 *adaptively chosen queries* the search space of the secret of one type of DEA can be reduced from 2^{2n} to *only* 4 for all $n \geq 1$ (see Theorem 2.8 and Algorithm 3). In addition, we have also shown that *only* 6 *queries* submitted in a *batch*[5] are sufficient to reduce the search space of the secret (x, y) from 2^{2n} to only 4, for all $n \geq 4$ (see Algorithm 5). These facts should be recognized as warnings to the designers who want to use combination of *modular addition* and XOR in their primitives. On the other hand the high exponential lower bound on the number of *batch queries* for solving another type of DEA ($3 \cdot 2^{n-4}$ for all $n \geq 4$) underlines an important theoretical reference point establishing it as *relatively stronger* under DC (see Theorem 2.14). It is, however, not known at this moment, how much influence these results have on many modern symmetric ciphers that mix other nonlinear operations with addition, yet our results leave promises to be used in evaluation of cryptographic strengths of many modern symmetric ciphers. One direct application of our results in practical cryptanalysis is described below.

Cryptanalysis of the Helix Cipher. Helix, proposed by Ferguson *et al.* [33], is a stream cipher with a combined MAC functionality. This cipher was a candidate for consideration in the 802.11i standard. The main component of the primitive is the combination of *addition* and XOR. The fact that the internal state of Helix depends on the plaintext allows for cryptanalysis with *chosen plaintexts* (CP) and *adaptive chosen plaintext* (ACP) both. The only attack so far on the cipher is by Muller who mounted an ACP attack which recovered the secret key of the Helix cipher with 2^{12} *adaptively chosen plaintexts* under the assumption that the nonce could be reused.

Here, we do not describe the details of the stream cipher Helix or Muller's attack on that. We refer the readers to [33] and [59] for an elaborate description of the cipher and the attack by Muller. Our key recovery attack goes along the lines of Muller's attack. However, we pick out the part which is critical to our attacks. The crux of the attacks is simply solving the equation $(x+y) \oplus (x+(y \oplus \beta)) = \gamma$ with β's and the corresponding γ's to recover the secret information (x, y), in

[5] *Batch queries* form a more practical attack scenario than the one by *adaptive queries.*

Table 2.2: New CP and ACP attacks on the Helix cipher

Type	Data (worst case)	Improvement	Date (best case)	Improvement
ACP	$50 \cdot 31 = 2^{10.6}$	factor of 3	$50 \cdot 2 = 100$	factor of 46.5
CP	$50 \cdot 2^{30} = 2^{35.64}$	New	-	-

the frameworks described in Sect. 2.3 and 2.4. In [59], it is found that, to recover the secret key of the Helix cipher, we need to solve the above equation 50 times. Throughout the process, β corresponds to a CP or an ACP according to whether the attack is built in a *batch query model* or an *adaptive query model*. Algorithm 2 and 4 solve the above equation with $(n-1)$ ACPs (when the *lsb* of x is '1') and 2^{n-2} CPs respectively. Note that the previous best known algorithm by Muller required $3(n-1)$ ACPs to solve the equation. Therefore, to mount the attacks, Algorithm 2 and 4 need to be run for 50 times. So, the total number of ACPs and CPs required are $50 \cdot (n-1)$ and $50 \cdot 2^{n-2}$ respectively. For the specified block-size $n = 32$, the data complexities of our ACP and CP attacks are shown in Table 2.2.

2.6 Conclusion and Further Research

In this chapter, we have shown the importance of solving DEA from both theoretical and practical points of view. The chapter seals any further search to improve lower bounds on the number of queries for solving DEA with *adaptive queries*. We also established nontrivial lower bounds on the number of *batch queries* to solve two types of DEA; in one case the lower bound is optimal up to a constant factor and the other case it is optimal up to a constant difference. Moreover, our algorithm improves the data complexity of an ACP attack on Helix cipher. We also recover the secret key of the Helix cipher with *chosen plaintexts* rather than *adaptive chosen plaintexts* which constitutes the *only* CP attack on this cipher so far. Our work leaves room for further research. One possible research direction may be to close the gap between the lower and upper bounds on the number of queries to solve DEA with *batch queries*. Another way to extend the work is to analyze components which combine more complex transformations such as *modular multiplication*, T-functions with *modular addition*.

Chapter 3

Cryptanalysis of the RC4 Stream Cipher

Chapter 4

Design and Analysis of RC4A

4.1 Introduction

In Chapter **??**, we have discussed the design principles and several weaknesses of the RC4 cipher. In this chapter we modify the RC4 keystream generator, within the scope of the existing model of an exchange shuffle, in order to achieve better security. The new cipher is named RC4A. We compare its security to the original RC4. Most of the known attacks on RC4 are less effective on RC4A. Furthermore, the new cipher needs fewer instructions per byte and it is possible to exploit the inherent parallelism to improve its performance.

4.2 RC4A: An Attempt to Improve RC4

As most of the existing known plaintext attacks on RC4 harness the stronger correlations between the internal and external states (in generic term b-predictive a-state attack which is described in Chapter **??**), in principle, making the output generation dependent on more random variables weakens the correlation between them, i.e., the probability to guess the internal state by observing output sequence can be reduced. The larger the number of the variables the weaker will be the correlation between them. On the other hand, intuitively, the large number of variables increases the time complexity as it involves more arithmetic operations.

4.2.1 RC4A Description

The KSA and PRBG of RC4A are described below.

1. **KSA.** First, we take one randomly chosen key k_1 whose size is equal to the key-size of RC4. Another key k_2 of equal size is also generated from a secure PRBG (e.g. PRBG of RC4) using k_1 as the seed. Applying the keys k_1 and k_2 to the KSA of RC4, we construct two S-boxes S_1 and S_2 from the identity permutation on $\{0, 1 \cdots, N-1\}$. We assume that S_1 and S_2 are two random permutations on $\{0, 1, 2, \ldots, N-1\}$, i.e., after the KSA, S_1 and S_2 are assumed to be uniformly distributed. Clearly the KSA of RC4A is equivalent to two applications of KSA of RC4 (the KSA of RC4 is described in Fig. **??**).

2. **PRBG.** In Fig. 4.1, we show the pseudo-code of the PRBG algorithm of RC4A. *Our efforts are focussed on the security of the PRBG only.*

Convention. All the arithmetic operations are computed modulo N. The transition of the internal states of the two S-boxes are based on an exchange shuffle as before. Here we introduce two variables j_1 and j_2 corresponding to S_1 and S_2 instead of one. The only modification is that the index-pointer $S_1[i] + S_1[j]$ evaluated on S_1 produces output from S_2 and vice-versa (see steps 5, 6 and 9, 10 of Fig. 4.1). The next round starts after each output generation.

RC4A uses fewer instructions per output byte than RC4. To produce two successive output bytes the i pointer is incremented once in case of RC4A where it is incremented twice to produce as many output words in RC4.

Parallelism in RC4A. The performance of RC4A can be further improved by extracting the parallelism latent in the algorithm. The parallel steps of the algorithm can be easily found by drawing a dependency graph of the steps shown in Fig. 4.1. In the following list the parallel steps of RC4A are shown within brackets.

1. (3, 7).
2. (4, 5, 9).
3. (6, 10).
4. (8, 2).

The existence of many parallel steps in RC4A is certainly an important aspect of this new cipher and it offers the possibility of a faster stream cipher if RC4A is implemented efficiently.

1. Set $i = 0, j_1 = j_2 = 0$
2. i++
3. $j_1 = j_1 + S_1[i]$
4. Swap$(S_1[i], S_1[j_1])$
5. $I_2 = S_1[i] + S_1[j_1]$
6. Output=$S_2[I_2]$
7. $j_2 = j_2 + S_2[i]$
8. Swap$(S_2[i], S_2[j_2])$
9. $I_1 = S_2[i] + S_2[j_2]$
10. Output=$S_1[I_1]$
11. Repeat from step 2.

Figure 4.1: PRBG of RC4A

4.3 Security Analysis of RC4A

The RC4A pseudorandom bit generator has passed all the statistical tests listed in [70]. RC4A achieves two major gains over RC4. By making every byte depend on at least two random values (e.g. O_1 depends on $S_1[1]$, $S_1[j_1]$ and $S_2[S_1[1]+S_1[j_1]]$) of S_1 and S_2 the secret internal state of RC4A becomes $N!^2 \times N^3$. So, for $N = 256$, the number of secret internal states for RC4A is approximately 2^{3392} when the number is only 2^{1700} for RC4.

In the following sections we describe how RC4A resists the two major attacks on it: one attempts to derive the entire internal state deterministically and another to derive a part of the internal state probabilistically.

4.3.1 Precluding the Backtracking Algorithm by Knudsen *et al.*

As mentioned earlier that the "guess on demand" backtracking algorithm by Knudsen *et al.* is so far the best algorithm to deduce the internal state of RC4 from the known plaintext [47]. Now we briefly discuss the functionality of the variant of the algorithm to be applied for RC4A.

The algorithm simulates RC4A by observing only the output bytes in recursive function calls. The values of $S[i]$ and $S[j]$ in one S-box are guessed from the permutation elements to agree with the output and its possible location in the other S-box. If they match then the algorithm calls the round function for the next round. If an anomaly occurs then it backtracks through the previous rounds and re-guesses. The number of outputs m, needed to uniquely determine the entire internal state, is bounded below by the inequality, $2^{nm} > (2^n!)^2$. Therefore, $m \geq 2N$ (note, $N = 2^n$).

Theorem 4.1 (RC4 vs RC4A) *If the expected computational complexity to derive the secret internal state of RC4A from known $2N$ initial output bytes with the algorithm by Knudsen et al. is C_{rc4a} and if the corresponding complexity for RC4 using N known initial output bytes is C_{rc4} then C_{rc4a} is much higher than C_{rc4} and C_{rc4a} can be approximated to C_{rc4}^2 under certain assumptions.*[1]

PROOF. According to the algorithm by Knudsen *et al.*, the internal state of RC4 is derived using only the first N output bytes, that is, simulating RC4 for the first N rounds. The variant of this algorithm which works on RC4A uses the initial $2N$ bytes, thereby runs for the first $2N$ rounds.

Let the algorithms A_1 and A_2 derive the secret internal states for RC4 and RC4A respectively. At every round the S-boxes are assigned either 0, 1, 2, or 3 elements and move to the next round.

Let, at the tth round, A_2 go to the next round after assigning k elements an expected number of $m_{k,t}$ times. So the number of value assignments in the tth round is $\sum_{k=0}^{3} k \cdot m_{k,t}$. Note, each of the $\sum_{k=0}^{3} m_{k,t}$ iterations gives rise to an S-box arrangement in the next round. It is possible that we reach some S-box arrangements from which no further transition to the next rounds is possible because of contradictions. In such case, we assume assignment of zero elements in the S-box till the last round is reached. Let the number of S-box arrangements at the tth round from which these $\sum_{k=0}^{3} m_{k,t}$ arrangements are generated is L_t. Consequently,

$$\sum_{k=0}^{3} m_{k,t} = L_{t+1} \,. \tag{4.1}$$

Now we set,

$$\sum_{k=0}^{3} k \cdot m_{k,t} = \tilde{k}_t \cdot \sum_{k=0}^{3} m_{k,t} = \tilde{k}_t \cdot L_{t+1} \,. \tag{4.2}$$

[1]The complexity is measured in terms of the number of value assignments.

In (4.2), $\tilde{k}_t$ is the expected number elements which are assigned to the S-boxes in each iteration in that particular round. If each of L_t is assumed to produce an expected $\tilde{L}_{t+1}$ number of S-box arrangements in the next round then (4.2) becomes,

$$\sum_{k=0}^{3} k \cdot m_{k,t} = \tilde{k}_t \cdot (\tilde{L}_{t+1} \cdot L_t)\,. \tag{4.3}$$

Denoting the total number of value assignments in the tth round by $C(t)$, it is easy to note from (4.3),

$$C(t) = \tilde{k}_t \cdot (\tilde{L}_{t+1} \cdot L_t)\,. \tag{4.4}$$

Proceeding this way we see that,

$$C(t+s) = \tilde{k}_{t+s} \cdot L_t \cdot \prod_{i=t}^{t+s} \tilde{L}_{i+1}\,. \tag{4.5}$$

If $t = 1$ then $L_t = 1$. Setting $t + s = n$ in (4.5), we get,

$$C(n) = \tilde{k}_n \cdot \prod_{i=1}^{n} \tilde{L}_{i+1}\,. \tag{4.6}$$

From (4.1) and (4.2), $C(n)$ can be evaluated $\forall n \in \{1, 2, \ldots 2N\}$ when $m_{k,t}$ is known $\forall (k,t) \in \{0, 1, 2, 3\}\{1, 2, \ldots 2N\}$.

It is important to note that on a random output sequence $\tilde{k}_{2f-1} \approx \tilde{k}_{2f}$ and $\tilde{L}_{2f} \approx \tilde{L}_{2f+1}$ $\forall f \in \{1, 2, \ldots N\}$. The reason behind the approximation is that, with the algorithm by Knudsen *et al.*, the difference between the expected number of assignments in the S-boxes in the $(2f-1)$th and the $2f$th rounds is very small. Therefore, the overall complexity C_{rc4a} becomes,

$$\begin{aligned}
C_{rc4a} &= \sum_{n=1}^{2N} C(n) \\
&= \sum_{n=1}^{2N} (\tilde{k}_n \cdot \prod_{i=1}^{n} \tilde{L}_{i+1}) \\
&= \tilde{k}_1 \cdot \tilde{L}_2 + \sum_{i=1}^{N} (\tilde{k}_{2i} \cdot \prod_{j=1}^{i} \tilde{L}^2_{2j+1}) + \sum_{i=2}^{N} (\tilde{k}_{2i-1} \cdot \tilde{L}_{2i} \cdot \prod_{j=1}^{i-1} \tilde{L}^2_{2j}) \\
&= \tilde{k}_1 \cdot \tilde{L}_2 + \sum_{i=1}^{N} (\tilde{k}_{2i-1} \cdot \prod_{j=1}^{i} \tilde{L}^2_{2j}) + \sum_{i=2}^{N} (\tilde{k}_{2i-1} \cdot \tilde{L}_{2i} \cdot \prod_{j=1}^{i-1} \tilde{L}^2_{2j})\,.
\end{aligned}$$

Replacing $\tilde{k}_q$ and $\tilde{L}_q$ by $x_{\frac{q+1}{2}}$ and $g_{\frac{q}{2}}$ we get,

$$C_{rc4a} = \sum_{i=1}^{N}(x_i \cdot \prod_{j=1}^{i} g_j^2) + x_1 \cdot g_1 + \sum_{i=2}^{N}(x_i \cdot g_i \cdot \prod_{j=1}^{i-1} g_j^2)\,. \tag{4.7}$$

Applying a similar technique as above it is easy to see that,

$$C_{rc4} = \sum_{i=1}^{N}(x_i \cdot \prod_{j=1}^{i} g_j)\,. \tag{4.8}$$

Again we note that the difference between the expected number of elements that are already assigned in S_1 for RC4A at round $(2t-1)$ and the expected number of elements in S for RC4 at round t is negligible. Therefore, the corresponding $\tilde{k}_t$ and $\tilde{L}_{t+1}$ for RC4 can be approximated to $\tilde{k}_{2t-1}$ and $\tilde{L}_{2t}$ for RC4A.

As the g_i's are real numbers greater than 1 and the x_i's are non-negative real numbers, from (4.7) and (4.8) it is easy to see that $C_{rc4a} \gg C_{rc4}$.

We observe from the algorithm that $x_i \in \{y : 0 \leq y \leq 3, y \in \mathbb{R}\}$. It is clear from the algorithm that x_i decreases as i increases. Intuitively, x_i is less than one in the last rounds. Therefore, assuming $C_{rc4a} \approx \prod_{i=1}^{N} g_i^2$ and $C_{rc4} \approx \prod_{i=1}^{N} g_i$, we get $C_{rc4a} \approx C_{rc4}^2$. □

By Theorem 4.1, the expected complexity to deduce the secret internal state of RC4A ($N = 256$) with the algorithm by Knudsen *et al.* is 2^{1558} when the corresponding complexity is 2^{779} for RC4.

4.3.2 Resisting the Fortuitous States Attack

Fluhrer and McGrew discovered certain RC4 states in which only m known consecutive S-box elements participate in producing the next m successive outputs. Those states are defined to be *Fortuitous States* (see [31, 64] for a detailed analysis). *Fortuitous States* increase the probability to guess a part of internal state in a known plaintext attack (see (**??**)). The larger the probability of the occurrence of a fortuitous state, the smaller will be the number of required rounds to obtain one of them.

RC4A also weakens the fortuitous state attack to a large degree. A moment's reflection shows that RC4A does not have any fortuitous state of length 1. Now we will compare the probability of the occurrence of a fortuitous state of length $2a$ in RC4A to that of length a in RC4. It is easy to note that a fortuitous state of length $2a$ of RC4A implies and is implied by two fortuitous states of length a of RC4 appearing simultaneously in S_1 and S_2. If C denotes the number of fortuitous states of length a of RC4 then the expected number of fortuitous states of length $2a$ in RC4A is C^2/N. Let P_a denote the probability of the occurrence of a fortuitous state of length a in RC4 and P_{2a} denote the probability of the occurrence of a fortuitous state of length $2a$ in RC4A. Then, for small values of a, $P_a = \frac{C}{N^{a+2}}$ and $P_{2a} = \frac{C^2}{N^{2a+4}}$ which immediately implies $P_{2a} < P_a$.

4.3.3 Resisting the 2nd Byte Attack by Mantin and Shamir

One may observe that the strong positive bias of the second output byte of RC4 toward zero [53] is also diminished in this new cipher RC4A as more random variables are required to be *fixed* for the bias-producing state to occur. Unlike the case with RC4, fixing a single S-box element did not yield any RC4A state that made the Mantin and Shamir's attack possible.

4.4 Attacks on the RC4A Stream Cipher

So far, two distinguishing attacks have been found on RC4A.

- The first attack was by Maximov who experimentally detected that the distribution of tth and $t+2$th outputs of RC4A is not uniform [55]. Using these weaknesses they built a distinguisher on RC4A which worked with 2^{58} output bytes. The attack is based on determining a set of internal states for which the outputs occur with a bias. These states are similar to the *predictive states* as discussed in Chapter **??**.

- The best distinguishing attack, so far, on RC4A is by Tsunoo *et al.* who builds a distinguisher with 2^{23} output bytes. The pivotal observation in their distinguisher is that, if the second element of the first S-box of RC4A is 2 then the first and the third outputs are *always* different. This bias-producing RC4A state is captured formally, using the notation used in this chapter, in the following claim.

  Claim 4.2 *For RC4A, if $S_1[1] = 2$ then $O_1 \neq O_3$.*

  See [90] for the detailed analysis of the distinguisher.

4.5 Open Problems and Directions for Future Work

Although RC4A has an improved security over the original cipher against most of the known plaintext attacks, it is still as vulnerable as RC4 against the attack by Golić which uses the positive correlation between the second binary derivative of the least significant bit output sequence and 1. The weakness originates from the slow change of the S-box in successive rounds that seems to be inherent in any model based on exchange shuffle. Therefore, this still remains an open problem whether it is possible to remove this weakness from the output words of the stream cipher based on an exchange shuffle while retaining all of its speed and security.

Our work leaves room for more research. It is worthwhile to note that one output byte generation in this existing model of exchange shuffle involves two random pointers; j and $S[i] + S[j]$. In RC4 both the pointers fetch values from a single S-box. We obtained better results by making $S[i] + S[j]$ fetch value from a different S-box. What if we obtain $S[j]$ from another S-box and generate output using three S-boxes?

4.6 Conclusions

In this chapter we attempted to improve the security of RC4 by introducing more random variables in the output generation process thereby reducing the correlation between the internal and the external states. However, we would like to mention that the security of RC4A could be further improved. For example, one could introduce key-dependent values of i and j at the beginning of the first round, and one could address the weaknesses of the Key Scheduling Algorithm. In this chapter, we have assumed that the original Key Scheduling Algorithm produces a uniform distribution of the initial permutation of elements, which is certainly not correct.

Chapter 5

Cryptanalysis of Py

Prediction is very difficult, especially of the future.
– Niels Bohr (1885-1962)

5.1 Introduction

In this chapter we look into another popular array-based stream cipher known as Py. The cipher Py, designed by Biham and Seberry [6], was submitted to the ECRYPT project [25] as a candidate for Profile 1 which covers software based stream ciphers suitable for high-speed applications. In the last couple of years a growing interest has been noticed among cryptographers to design fast and secure stream ciphers because of weaknesses being found in many *de facto* standards such as RC4 and also due to the failure of the NESSIE project [61] to find a stream cipher that met its very stringent security requirements. The current stream cipher, namely Py, is one of the attempts in this direction.

Py is the most recent addition to the class of stream ciphers whose design principles are motivated by that of RC4 (see [36, 41, 65, 95, 98]). Like RC4, Py also uses the technique of random shuffle to update the internal state. In addition, Py uses a new technique of rotating all array elements in every round with a minimal running time. The high performance (it is 2.5 times faster than the RC4 on Pentium III) and its apparent security make this cipher very attractive for selection to the Profile 1 of the ECRYPT project.

This chapter identifies several biased pairs of output bits of Py at rounds t and $t+2$ (where $t > 0$). The weaknesses originate from the non-uniformity of the distributions of carry bits in modular addition used in Py. Using those biases, we have constructed a class of distinguishers. We show that the best of them works successfully with $2^{84.7}$ randomly chosen key/IVs, the first 24 bytes for each key (i.e., a total of $2^{89.2}$ bytes) and running time $t_{ini} \cdot 2^{84.7}$ where t_{ini} is the running time of the key/IV setup of Py. We also show that a simple adjustment to the above distinguisher reduces the number of key/IVs, the data complexity and the running time to $2^{28.7}$, a total of $2^{87.7}$ bytes and $t_r \cdot 2^{84.7}$ respectively, where t_r is the running time of a single round of Py. Note that the allowable key-size and keystream length of Py are 256 bits and 2^{64} bytes respectively. Therefore, these results imply that – even if our attack has a larger total complexity – Py fails to provide the security level expected from an ideal stream cipher with the parameter sizes of Py. Therefore, we believe that our results present a theoretical break of the cipher; see Sect. 5.9 for an elaborate discussion on this issue. It is important to note that the weaknesses of Py which are described in this chapter, still cannot be implemented in practice in view of the its high time complexity. However, the individual distinguishers open the possibility to combine them in order to generate more efficient distinguishers.

5.2 Description of Py

Py is a synchronous stream cipher which normally uses a 32-byte key (however, the key can be of any size from 1 byte to 256 bytes) and a 16-byte initial value or IV (IV can also be of any size from 1 byte to 64 bytes). The allowable keystream length per key/IV is 2^{64} bytes. Py works in three phases – a key setup algorithm, an IV setup algorithm and a round function which generates two output-words (each output-word is 4 bytes long). The internal state of Py contains two S-boxes Y, P and a variable s. Y contains 260 elements each of which is 32 bits long. The elements of Y are indexed by [-3, -2,..., 256]. P is a permutation of the elements of $\{0, ..., 255\}$. The main feature of the stream cipher Py is that the S-boxes are updated like 'rolling arrays' [6]. The technique of 'rolling arrays' means that, in each round of Py, *(i)* one or two elements of the S-boxes are updated (line 1 and 7 of Algorithm 6) and *(ii)* all the elements are cyclically rotated by one position toward the left (line 2 and 8 of Algorithm 6). In our analysis, we have assumed that, after the key/IV setup, Y, P and the variable s are uniformly distributed and independent. Under this assumption we analyzed the round function of Py (or Pseudorandom Bit Generation Algorithm) which is described in Algorithm 6. See [6] for a detailed description of the key/IV setup algorithms.

The inputs to Algorithm 6 are $Y[-3, ..., 256]$, $P[0, ..., 255]$ and s, which are obtained after the key/IV setup. Lines 1 and 2 describe how P is updated and rotated. In the update stage, the 0th element of P is swapped with another element in P, which is accessed indirectly, using $Y[185]$. The next step involves a cyclic rotation by one position, of the elements in P. This implies that the entry in $P[0]$ becomes the entry in $P[255]$ in the next round and the entry in $P[i]$ becomes the entry in $P[i-1]$ ($\forall i \in \{1, 2, ..., 255\}$). Lines 3 and 4 of Algorithm 6 indicate how s is updated and its elements rotated. Here, the '$ROTL32(s, x)$' function implies a cyclic left rotation of s by x bit-positions. The output-words (each 32-bit) are generated in lines 5 and 6. The last two lines of the algorithm explain the update and rotation of the elements of Y. The rotation of Y is carried in the same manner as the rotation of P.

Algorithm 6 Single Round of Py

Input: $Y[-3, ..., 256]$, $P[0, ..., 255]$, a 32-bit variable s
Output: 64-bit random output
/*Update and rotate P*/
1: swap ($P[0]$, $P[Y[185]\&255]$);
2: rotate (P);
/* Update s*/
3: $s+= Y[P[72]] - Y[P[239]]$;
4: $s = ROTL32(s, ((P[116] + 18)\&31))$;
/* Output 8 bytes (least significant byte first)*/
5: output $((ROTL32(s, 25) \oplus Y[256]) + Y[P[26]])$;
6: output $((\quad s \quad \oplus Y[-1]) + Y[P[208]])$;
/* Update and rotate Y*/
7: $Y[-3] = (ROTL32(s, 14) \oplus Y[-3]) + Y[P[153]]$;
8: rotate(Y);

5.2.1 Notation and Convention

As Py uses different types of internal and external states (e.g. integer arrays, 32-bit integer) and they are updated every round, it is important to denote all the states and rounds in a simple but consistent way. In every round of Py, the S-box P and the variable s are updated before the output generation (see Algorithm 6). The other S-box, namely Y, is updated after the output generation.

1. In the beginning of any round i, the components of the internal state are denoted by P_{i-1}, s_{i-1} but Y_i.

2. At the end of any round i, the internal state is updated to P_i, s_i and Y_{i+1}. (If the above two conventions are followed, we have P_i, s_i and Y_i in the formulas for the generation of the output-words in round i (line 5 and 6 of Algorithm 6)).

3. The nth element of the arrays Y_i and P_i, are denoted by $Y_i[n]$ and $P_i[n]$ respectively.

4. The output-words generated in line 5 and line 6 of Algorithm 6 are referred to as the '1st output-word' and the '2nd output-word' respectively.

5. $O_{l,m}$ denotes the lth ($l \in \{1,2\}$) output-word generated in the mth round of Py. $O_{l,m(j)}$ denotes the jth bit of $O_{l,m}$. For example, $O_{1,3(5)}$ denotes the 5th bit of the 1st output-word in round 3.

6. As mentioned in the list of symbols in the beginning of this thesis, the '+' operator denotes *addition modulo* 2^{32} as a rule, except when it is used to increment elements of P (particularly in expressions of the form $P_i[n] = P_j[m] + 1$, where '+' denotes *addition over* $\mathbb{Z}$). Similarly, the meaning of '$-$' should be understood.

5.2.2 Assumption

We assume that the key setup and the IV setup algorithms of Py are perfect, i.e., after the execution of them, the permutation P, the elements of Y and the s are uniformly distributed and independent. When we are interested in the analysis of the mixing of bits of the internal state by the PRBG, the above assumption is reasonable, particularly when it is difficult to derive any relation between inputs and outputs of the key/IV setup algorithm. Apart from that the assumption is in agreement with a claim made in Sect. 6.4 of [6] that the key/IV setup leaks no statistical information on the internal state.

5.3 Motivational Observation

Our main observation is that, if certain conditions on the elements of the S-box P are satisfied then the least significant bit (lsb) of the 1st output-word at the 1st round is equal to the lsb of the 2nd output-word at the 3rd round.

Theorem 5.1 $O_{1,1(0)} = O_{2,3(0)}$ *if the following six conditions on the elements of the S-box P are simultaneously satisfied.*

1. $P_2[116] \equiv -18 \bmod 32$ *(event A),*

2. $P_3[116] \equiv 7 \bmod 32$ *(event B),*

3. $P_2[72] = P_3[239] + 1$ *(event C),*

4. $P_2[239] = P_3[72] + 1$ *(event D),*

5. $P_1[26] = 1$ *(event E),*

6. $P_3[208] = 254$ *(event F).*

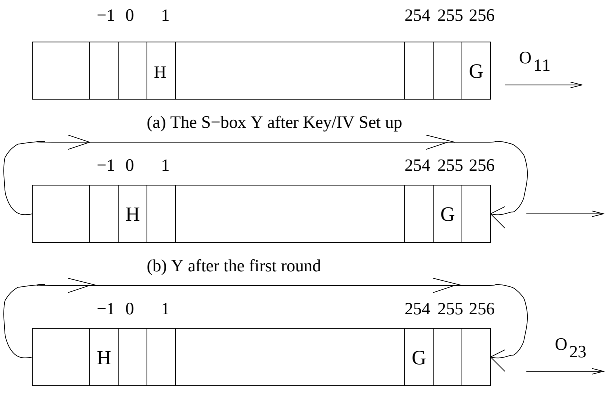

Figure 5.1: (a) $P_1[26] = 1$ (condition 5): G and H are used in $O_{1,1}$, (b) Y_2 (i.e., Y after the 1^{st} round), (c) $P_3[208] = 254$ (condition 6): G and H are used in $O_{2,3}$

PROOF. The formulas for the $O_{1,1}$, $O_{2,3}$ and s_2 are given below (see Sect. 5.2):

$$\begin{aligned} O_{1,1} &= (ROTL32(s_1, 25) \oplus Y_1[256]) + Y_1[P_1[26]]\,, & (5.1)\\ O_{2,3} &= (s_3 \oplus Y_3[-1]) + Y_3[P_3[208]]\,, & (5.2)\\ s_2 &= ROTL32(s_1 + Y_2[P_2[72]] - Y_2[P_2[239]], \\ &\quad P_2[116] + 18 \bmod 32)\,. & (5.3) \end{aligned}$$

• Condition 1 (i.e., $P_2[116] \equiv -18 \bmod 32$) reduces (5.3) to

$$s_2 = s_1 + Y_2[P_2[72]] - Y_2[P_2[239]]\,.$$

• Condition 2 (i.e., $P_3[116] \equiv 7 \bmod 32$) together with Condition 1 implies

$$s_3 = ROTL32((s_1 + Y_2[P_2[72]] - Y_2[P_2[239]] + Y_3[P_3[72]] - Y_3[P_3[239]]), 25)\,.$$

• Condition 3 and 4 (that is, $P_2[72] = P_3[239] + 1$ and $P_2[239] = P_3[72] + 1$) reduce the previous equation to

$$s_3 = ROTL32(s_1, 25)\,. \tag{5.4}$$

From (5.1), (5.2), (5.4) we get:

$$\begin{aligned} O_{1,1} &= (ROTL32(s_1, 25) \oplus Y_1[256]) + Y_1[P_1[26]]\,, & (5.5)\\ O_{2,3} &= (ROTL32(s_1, 25) \oplus Y_3[-1]) + Y_3[P_3[208]]\,. & (5.6) \end{aligned}$$

In Fig. 5.1, conditions 5 and 6 are described. According to the figure,

$$\begin{aligned} H &= Y_1[P_1[26]] = Y_3[-1]\,, && (5.7) \\ G &= Y_1[256] = Y_3[P_3[208]]\,. && (5.8) \end{aligned}$$

Applying (5.7) and (5.8) in (5.5) and (5.6) we get,

$$O_{1,1(0)} \oplus O_{2,3(0)} = Y_1[256]_{(0)} \oplus Y_1[P_1[26]]_{(0)} \oplus Y_3[-1]_{(0)} \oplus Y_3[P_3[208]]_{(0)} = 0\,.$$

This completes the proof. □

5.4 Bias in the Distribution of the 1st and the 3rd Outputs

In this section, we shall compute $P[O_{1,1(0)} \oplus O_{2,3(0)} = 0]$ using the results of Sect. 5.3. We now recall the six events (or conditions) A, B, C, D, E, F as described in Theorem 5.1. First, we shall compute $P[A \cap B \cap C \cap D \cap E \cap F]$. The elements involved in the calculation of the probability are $P_1[26]$, $P_2[72]$, $P_2[116]$, $P_2[239]$, $P_3[72]$, $P_3[116]$, $P_3[208]$, and $P_3[239]$. Now we observe that Algorithm 6 ensures that all the above elements occupy unique indices in round 1. We calculate the probabilities step by step using Bayes' rule under the assumption described in Sect. 5.2.2.

1. $P[E] = \frac{1}{256}$,
2. $P[E \cap F] = P[F|E] \cdot P[E] = \frac{1}{255} \cdot \frac{1}{256}$,
3. $P[A \cap E \cap F] = P[A|E \cap F] \cdot P[E \cap F] = \frac{8}{254} \cdot \frac{1}{255} \cdot \frac{1}{256}$,
4. $P[A \cap B \cap E \cap F] = P[B|A \cap E \cap F] \cdot P[A \cap E \cap F] = \frac{8}{253} \cdot \frac{8}{254} \cdot \frac{1}{255} \cdot \frac{1}{256}$,
5. Similarly, $P[A \cap B \cap C \cap E \cap F] = \frac{247}{251 \cdot 252} \cdot \frac{8}{253} \cdot \frac{8}{254} \cdot \frac{1}{255} \cdot \frac{1}{256}$,
6. $P[A \cap B \cap C \cap D \cap E \cap F] \approx \frac{244}{249 \cdot 250} \cdot \frac{247}{251 \cdot 252} \cdot \frac{8}{253} \cdot \frac{8}{254} \cdot \frac{1}{255} \cdot \frac{1}{256} \approx 2^{-41.9}$.

Under the assumption of randomness and uniformity of the distributions of the S-box elements and of s after the key/IV setup, if any of the six events – described in

Theorem 5.1 – does not occur then $P[O_{1,1(0)} \oplus O_{2,3(0)} = 0] = \frac{1}{2}$ (see Appendix C.1 for a justification for that). That is,

$$P[O_{1,1(0)} \oplus O_{2,3(0)} = 0 | (A \cap B \cap C \cap D \cap E \cap F)^c] = \frac{1}{2}\,.$$

We denote the event $A \cap B \cap C \cap D \cap E \cap F$ by L and its complement by L^c. Therefore,

$$\begin{aligned} P[O_{1,1(0)} \oplus O_{2,3(0)} = 0] &= P[O_{1,1(0)} \oplus O_{2,3(0)} = 0|L] \cdot P[L] \\ &+ P[O_{1,1(0)} \oplus O_{2,3(0)} = 0|L^c] \cdot P[L^c] \\ &= 1 \cdot 2^{-41.91} + \frac{1}{2} \cdot (1 - 2^{-41.91}) \\ &= \frac{1}{2} \cdot (1 + 2^{-41.91})\,. \end{aligned} \tag{5.9}$$

Note that, if Py had been an ideal PRBG then the above probability would have been exactly $\frac{1}{2}$.

5.5 The Distinguisher

The distinguishers that we construct in this section and Sect. 5.6, using the bias described in Sect. 5.4, are *prefix distinguishers*. In Sect. 5.7, we build a *regular distinguisher*; however, the number of outputs needed for this distinguisher exceeds the allowable keystream length per key/IV. In Section 5.8, we propose a *hybrid distinguisher* mainly to reduce the time cost of our *prefix distinguisher*. An elaborate introductory discussion on various types distinguishers can be found in Sect. 1.3.3.

Algorithm 7 *A Distinguisher* separating Py from Random

Input: An n-bit sequence $(z_1, z_2, z_3, \cdots, z_n)$
Output: Whether the sequence is random or generated by Py
1: Compute LLR $= \sum_i \log(\frac{P_0[z_i]}{P_1[z_i]})$;
2: If LLR ≥ 0 then return 1 (i.e., "The sequence is from Py") else 0 (i.e., "The sequence is random");

Algorithm 7. The *prefix distinguisher* that separates Py from random is described in Algorithm 7. The input to the algorithm is a realization of the sequence of binary random variables $(z_1, z_2, z_3, \cdots, z_n)$. The adversary first generates n key/IV pairs $X_1, X_2, X_3, \cdots, X_n$ randomly and then computes

$z_i = O_{1,1(0)} \oplus O_{2,3(0)}$ for all X_i, $1 \leq i \leq n$. Using the results obtained by Baignères, Junod and Vaudenay [3], we see that Algorithm 7 is an *optimal distinguisher*. Given a fixed number of samples, an *optimal distinguisher* attains the *maximum advantage*. Note that the random variables z_i's are independent of each other and each of them follows the distribution computed in Sect. 5.4 (call the distribution D_0). Let the uniform distribution on alphabet $[0, 1]$ be denoted by D_1. In Algorithm 7, $P_0[z_i]$ (shorthand for $P_{\mathsf{D}_0}[z_i]$) denotes the probability of occurrence of z_i when chosen according to D_0 (similarly $P_1[z_i]$ and $P_{\mathsf{D}_1}[z_i]$).

Let the Algorithm 7, the sequence of variables $(z_1, z_2, z_3, \cdots, z_n)$ and the quantity $\sum_i \log(\frac{P_0[z_i]}{P_1[z_i]})$ be denoted by $\mathcal{F}$, Z and LLR respectively. Now we will compute the *advantage* of $\mathcal{F}$ (the advantage of this distinguisher has been independently calculated by Paul Crowley [17]). Now we recall the *advantage* of a distinguisher as described in (1.8):

$$\mathrm{Adv}_{\mathcal{F}}^{n} = \left| P_{\mathsf{D}_0^n}[\mathcal{F}(Z) = 1] - P_{\mathsf{D}_1^n}[\mathcal{F}(Z) = 1] \right| . \tag{5.10}$$

Following the results in [3], we see that for large n,

$$\begin{aligned} P_{\mathsf{D}_0^n}[\mathcal{F}(Z) = 1] &= P_{\mathsf{D}_0}[\mathrm{LLR} \geq 0] \approx \Phi\left(\frac{\sqrt{n}\mu_0}{\sigma_0}\right) , \\ P_{\mathsf{D}_1^n}[\mathcal{F}(Z) = 1] &= P_{\mathsf{D}_1}[\mathrm{LLR} \geq 0] \approx \Phi\left(\frac{\sqrt{n}\mu_1}{\sigma_1}\right) . \end{aligned}$$

where Φ is the standard normal distribution function expressed as,

$$\Phi(z) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z} e^{-\frac{1}{2}u^2}\, du \,.$$

If the two distributions D_0 and D_1 are close (i.e.,$\left|P_0[z] - P_1[z]\right| \ll P_1[z]$) then

$$\mu_0 \approx -\mu_1 \approx \frac{1}{2} \sum_{z \in [0,1]} \frac{(P_0[z] - P_1[z])^2}{P_1[z]} \text{ and } \sigma_0^2 \approx \sigma_1^2 \approx \sum_{z \in [0,1]} \frac{(P_0[z] - P_1[z])^2}{P_1[z]} \,.$$

The above equations suggest that, for a given n, using the known distributions D_0 and D_1, the *advantage* of Algorithm 7 can be computed from (5.10). Some simple calculations show that, if $P_0[0] - P_1[0] = \frac{1}{M}$, then, to ensure the *advantage* of the distinguisher to be greater than 0.5, the required number of samples is

$$n = 0.4624 \cdot M^2 \,. \tag{5.11}$$

In the present case $P_0[0] - P_1[0] = \frac{1}{2^{42.9}}$ (see Sect. 5.4). Therefore, from (5.11), $n = 0.4624 \cdot \left(2^{42.9}\right)^2 = 2^{84.7}$ samples (i.e., as many randomly chosen key/IVs) can distinguish Py from random with an advantage that exceeds 0.5. The time cost to build this distinguisher is $t_{ini} \cdot 2^{84.7}$ where t_{ini} is the running time of the key/IV setup of Py. Note that, for each key/IV, we collect the first 24 bytes of the keystream. Therefore, the number of bytes required to establish the distinguisher is $2^{84.7} \cdot 24 = 2^{89.2}$.

5.6 Biases among other Pairs of Bits and Distinguishers

In Sect. 5.4, we have shown a bias in $(O_{1,1(0)}, O_{2,3(0)})$. In this section, we show that the bias is present in $(O_{1,1(i)}, O_{2,3(i)})$, where $0 \leq i \leq 31$; however, the bias gradually reduces as i increases. From (5.1) and (5.2), we get:

$$\begin{aligned} O_{1,1(i)} &= ROTL32(s_1, 25)_{(i)} \oplus Y_1[256]_{(i)} \oplus Y_1[P_1[26]]_{(i)} \oplus c_{1(i)}\,, \\ O_{2,3(i)} &= s_{3(i)} \oplus Y_3[-1]_{(i)} \oplus Y_3[P_3[208]]_{(i)} \oplus c_{3(i)}\,, \end{aligned}$$

where $0 \leq i \leq 31$ and c_1, c_3 are the carry terms in (5.1) and (5.2) respectively.

A Special Case. If all the 6 conditions of Theorem 5.1 are satisfied, $O_{1,1}$ and $O_{2,3}$ can be written in the following form (see Theorem 5.1):

$$\begin{aligned} O_{1,1} &= (S \oplus G) + H\,, && (5.12) \\ O_{2,3} &= (S \oplus H) + G\,, && (5.13) \end{aligned}$$

which implies that

$$O_{1,1(i)} \oplus O_{2,3(i)} = c_{1(i)} \oplus c_{3(i)}, \qquad 0 \leq i \leq 31\,,$$

where the carries $c_{1(i)}$ and $c_{3(i)}$ can be calculated from the following recursive relations (note that $c_{1(0)} = c_{3(0)} = 0$),

$$\begin{aligned} c_{1(i)} &= c_{1(i-1)}(S_{(i-1)} \oplus G_{(i-1)}) \oplus c_{1(i-1)}H_{(i-1)} \oplus \\ &\quad H_{(i-1)}(S_{(i-1)} \oplus G_{(i-1)})\,, && (5.14) \\ c_{3(i)} &= c_{3(i-1)}(S_{(i-1)} \oplus H_{(i-1)}) \oplus c_{3(i-1)}G_{(i-1)} \oplus \\ &\quad G_{(i-1)}(S_{(i-1)} \oplus H_{(i-1)})\,. && (5.15) \end{aligned}$$

Computing $P[O_{1,1(i)} \oplus O_{2,3(i)} = 0]$. Note that

$$\begin{aligned} P[O_{1,1(i)} \oplus O_{2,3(i)} = 0] &= P[O_{1,1(i)} \oplus O_{2,3(i)} = 0|L] \cdot P[L] \\ &+ P[O_{1,1(i)} \oplus O_{2,3(i)} = 0|L^c] \cdot P[L^c] \\ &= \underbrace{P[c_{1(i)} \oplus c_{3(i)} = 0|L]}_{p_i} \cdot P[L] \\ &+ \underbrace{P[O_{1,1(i)} \oplus O_{2,3(i)} = 0|L^c]}_{X_i} \cdot P[L^c] \,, \end{aligned} \tag{5.16}$$

where $i \in [0, 31]$ and the event L is $A \cap B \cap C \cap D \cap E \cap F$. Note that four components are involved in (5.16); they are $P[L]$, $P[L^c]$, p_i and X_i. Next, we show how to determine these four quantities.

1,2. Computing $P[L]$ and $P[L^c]$: the results in Sect. 5.4 show that $P[L] = 2^{-41.9}$ and $P[L^c] = (1 - 2^{-41.9})$.

3. Computing p_i: now we recursively compute $P[c_{1(i)} \oplus c_{3(i)} = 0|L]$, denoted by p_i in (5.16) (similarly p_{i-1} should be understood), from the following equation derived directly from (5.14) and (5.15).

$$\begin{aligned} c_{1(i)} \oplus c_{3(i)} &= (c_{1(i-1)} \oplus c_{3(i-1)})(S_{(i-1)} \oplus G_{(i-1)} \oplus H_{(i-1)}) \oplus \\ &\quad S_{(i-1)}(G_{(i-1)} \oplus H_{(i-1)}) \,. \end{aligned} \tag{5.17}$$

Note that the variables G, H, S are uniformly distributed and independent. The truth table for (5.17) is shown in Table 5.1. From Table 5.1, using Bayes' rule, we obtain the following recursion to compute p_i,

$$p_i = \frac{p_{i-1}}{2} + \frac{1}{4} \,.$$

We already know that $p_0 = 1$ (i.e., $P[O_{1,1(0)} \oplus O_{2,3(0)} = 0|L] = 1$). Therefore, solving the above recurrence relation, finally we get

$$p_i = \frac{1}{2} + \frac{1}{2^{i+1}} \,, \quad 0 \le i \le 31 \,. \tag{5.18}$$

4. Computing X_i: according to the results obtained in Appendix C.1 it is reasonable to assume that

$$X_i = \frac{1}{2} \,, \qquad \text{for all } i \in [0, 24] \cup [26, 31] \,.$$

General Expression. Using the above results, recalling (5.16), we find,

$$P[O_{1,1(i)} \oplus O_{2,3(i)} = 0] \quad = \quad \frac{1}{2}(1 + 2^{-(41.9+i)}) \,, \tag{5.19}$$

Table 5.1: Truth table for (5.17). The last column in each row indicates the probability of the occurrence of that row

$c_{1(i-1)} \oplus c_{3(i-1)}$	$S_{(i-1)}$	$B_{(i-1)}$	$A_{(i-1)}$	$c_{1(i)} \oplus c_{3(i)}$	Probability
0	0	0	0	0	$p_{i-1}/8$
0	0	0	1	0	$p_{i-1}/8$
0	0	1	0	0	$p_{i-1}/8$
0	0	1	1	0	$p_{i-1}/8$
0	1	0	0	0	$p_{i-1}/8$
0	1	0	1	1	
0	1	1	0	1	
0	1	1	1	0	$p_{i-1}/8$
1	0	0	0	0	$(1-p_{i-1})/8$
1	0	0	1	1	
1	0	1	0	1	
1	0	1	1	0	$(1-p_{i-1})/8$
1	1	0	0	1	
1	1	0	1	1	
1	1	1	0	1	
1	1	1	1	1	

where $i \in [0,24] \cup [26,31]$. It is also reasonable to assume (due to the event L' as described in Appendix C.1) that

$$\begin{aligned} P[O_{1,1(25)} \oplus O_{2,3(25)} = 0] &\geq \frac{1}{2}(1 + 2^{-(41.9+25)}) \\ &\geq \frac{1}{2}(1 + 2^{-66.9}) . \end{aligned}$$

From (5.19), one may see that $P[O_{1,1(i)} \oplus O_{2,3(i)} = 0]$ attains the maximum value if $i = 0$. Our distinguisher, described in Sect. 5.4 and 5.5, exploits the case if $i = 0$. Equation (5.19) suggests that many distinguishers can be generated using different $(O_{1,1(i)}, O_{2,3(i)})$'s rather than only $(O_{1,1(0)}, O_{2,3(0)})$, however, the amount of bias decreases as i increases (i.e., we get the most effective distinguisher if $i = 0$). For example, if $i = 1$,

$$P[O_{1,1(1)} \oplus O_{2,3(1)} = 0] = \frac{1}{2}(1 + 2^{-42.91}) .$$

For the above case, taking the 1st bits of $O_{1,1}$ and $O_{2,3}$, the number of samples (i.e., the number of key/IVs) required to establish a distinguisher with advantage exceeding 0.5 is $2^{86.7}$ (see (5.11)). Similarly, if we consider $i = 2$ then the number of required samples is $2^{88.7}$.

5.7 Generalizing the Bias at Rounds t and $t+2$: A Distinguisher Using a Single Keystream

Under assumptions similar to those in Sect. 5.2.2, the results of Sect. 5.3 and Sect. 5.4 are valid even if we consider any rounds t and $t+2$ ($t > 0$) instead of just rounds 1 and 3. In other words, instead of $(O_{1,1(0)}, O_{2,3(0)})$, one can show that the bias exists even in the distribution of $(O_{1,t(i)}, O_{2,(t+2)(i)})$. Now, we state a theorem which is the generalized version of Theorem 5.1.

Theorem 5.2 $O_{1,t(0)} = O_{2,(t+2)(0)}$ *if the following six conditions on the elements of the S-box P are simultaneously satisfied.*

1. $P_{t+1}[116] \equiv -18 \pmod{32}$,
2. $P_{t+2}[116] \equiv 7 \pmod{32}$,
3. $P_{t+1}[72] = P_{t+2}[239] + 1$,
4. $P_{t+1}[239] = P_{t+2}[72] + 1$,
5. $P_t[26] = 1$,

6. $P_{t+2}[208] = 254$.

Using the above theorem and the techniques used before, it is easy to show that (see (5.9))

$$P[O_{1,t(0)} \oplus O_{2,(t+2)(0)} = 0] = \frac{1}{2}(1 + 2^{-41.91})\,.$$

The fact that the above probability is valid, $\forall\, t > 0$, allows us to generate a *regular distinguisher* with the number of rounds $2^{84.7}$ of a *single keystream* (see Sect. 1.3.3 for a definition of a *regular distinguisher*). This means that $2^{84.7} \cdot 2^3 = 2^{87.7}$ bytes of *a single* stream generated by a randomly chosen key/IV are sufficient to distinguish Py from random with success probability greater than 0.5. The workload here is also comparable to $2^{87.7}$. However, this attack is rendered ineffective because the amount of required bytes falls outside the allowable keystream length of 2^{64} bytes.

5.8 A More Efficient Hybrid Distinguisher

The results of Sect. 5.5 and Sect. 5.7 lead us in a natural way to build a *hybrid* distinguisher by making a trade-off between the number of key/IVs and output bytes per key/IV. It is apparent from the previous discussion that, to realize our distinguisher, we need $2^{84.7}$ pairs of internal states (recall that the internal state of Py consists of the arrays P, Y and a 32-bit integer s) with each pair being separated by one round. Then, under the assumption that the first state of each pair is randomly generated, those pairs can be used to build a distinguisher. As the allowable number of rounds per key/IV is $2^{64-8} = 2^{56}$, the number of required key/IVs is $2^{84.7-56} = 2^{28.7}$ to construct this *hybrid distinguisher*. The main difference between the *prefix distinguisher* in Sect. 5.5 and this *hybrid distinguisher* is that the running time to build this *hybrid distinguisher* is much smaller, as it requires the key/IV setup to run only for $2^{28.7}$ times compared to $2^{84.7}$ times for the previous *prefix distinguisher*. Therefore, the time and the data complexity of this distinguisher are $t_r \cdot 2^{84.7}$ and $2^{87.7}$ bytes respectively, where t_r is the running time of a single round of Py. Furthermore, this *hybrid* distinguisher does not breach the cipher specifications.

5.9 Do Our Distinguishers Break the Cipher Py?

The subject of what constitutes a break of a *practical stream cipher* or a PRBG is a highly contentious issue even if the area is quite well developed in theory. The definition of a cryptographically strong pseudorandom bit generator (CSPRBG), given by Blum and Micali, has been provided in Sect. 1.3.1. We see that, theoretically, a PRBG is studied according to how it behaves when the length of *seed* is increased asymptotically. The major problem in fitting the analyses of practical stream ciphers into the above framework is that, most of the ciphers work with *fixed sized* keys and keystream bits (e.g. Py allows 256-bit key and 2^{64} bytes of keystream per key/IV pair). Such constraints make the asymptotic analyses of practical stream ciphers impossible. For a practical PRBG with a *fixed sized key* (such as Py), given the first s output bits generated by an unknown key/IV, the $s+1$st bit can be predicted with a high probability with running time bounded above by a trivial exhaustive search. As there is no non-trivial upper bound on the running time of a distinguishing attack on a stream cipher (or PRBG) with a fixed sized key, any legal distinguishing attack with running time less than exhaustive search constitutes an academic break of the cipher.[1] Therefore, our attacks from Sect. 5.5, Sect. 5.6 and Sect. 5.8 imply a theoretical break of Py. However, it should be noted that each of the attacks presented in the chapter requires a workload larger than 2^{85} and therefore, poses no practical threats to the cipher.

Do our distinguishing attacks on Py violate the designers' claims?

The stream cipher Py is claimed by the designers to have up to 256-bit security (see Appendix A of [6]). In the authors' words, "The security claims are for keys up to 256 bits (32 bytes) and IVs up to 128 bits (16 bytes)". 256-bit is also the category of security level under which Py is included in the ECRYPT project [20]. According to the discussion on the definition of *n-bit security* of a perfectly secure stream cipher, it is clear that this claim is compromised by our attacks.

However, in Sect. 6.1 of [6], the authors claim, "There are no distinguishing attacks that succeed given less than 2^{64} bytes of key stream with a complexity less than of exhaustive search." It is understood from [5], that those 2^{64} bytes, as mentioned in the claim, may be generated by many keys rather than a single key. Under this interpretation, our attacks do not violate this claim, since our best attack requires $2^{87.7}$ bytes of output.

As a result we conclude that two claims, mentioned above, contradict each other with respect to the attacks mentioned in this chapter. At this point, we leave it to the reader to decide on the implications of our distinguishers.

[1] A *legal distinguishing attack* is the one which does not violate the specified parameters of the cipher.

5.10 Future Work

One could try to combine the individual biases of the pairs of bits presented here to develop a more sophisticated distinguisher with fewer output bytes. Paul Crowley has reduced the time and output bytes of our distinguisher to 2^{72} each, by analyzing our observation in Sect. 5.3 using a Hidden Markov Model [17]. A plausible strategy consists of identifying many more correlations between internal and external states of Py in order to reduce the time and data complexity of the distinguisher.

5.11 Conclusion and Remarks

The chapter presented several weaknesses of the stream cipher Py. We discovered a class of distinguishers for the cipher, the best of which works with $2^{87.7}$ bytes and comparable time. We also showed that the output stream of Py with a recommended keystream length of 2^{64} bytes, contains biases at different points – this fact can be exploited to build more effective distinguishers. These results break the cipher Py academically. However, the data complexity for the best distinguishing attack falls well beyond the time complexity what is feasible today. Therefore, these weaknesses pose no practical threat to the security of the cipher at this moment. However, very recently Wu and Preneel have reported key recovery attacks on Py with chosen IVs [97].

Chapter 6

Array-based Stream Ciphers with Short Indices and Large Elements: Attacks on Py6, IA, ISAAC, NGG, GGHN

Life must be understood backwards; but it must be lived forward.
– Kierkegaard (1813-1855)

6.1 Introduction

In Chapter **??**, 4 and 5, we have already analyzed the design principles and the weaknesses of three important array-based stream ciphers RC4, RC4A and Py. In this chapter, we take a closer look at a specific type of array-based stream ciphers (or PRBGs) where the size of the index is shorter than that the element pointed to by the index.

In general, stream ciphers are of paramount importance in fast cryptographic applications such as encryption of streaming data where information is generated at a high speed. Unfortunately, the state-of-the art of this type of ciphers, to euphemize, is not very promising as reflected in the failure of the NESSIE project to select a single cipher for its profile [61] and also the attacks on a number of submissions for the ongoing ECRYPT project [25]. Because of plenty of common features as well as dissimilarities, it is almost impossible to classify the entire gamut of stream ciphers into small, well-defined, disjoint groups, so that one group of ciphers can be analyzed in isolation of the others. However, in view of the identical data structures and similar operations in a number of stream ciphers and the fact that they are vulnerable against certain kinds of attacks originating from some basic flaws inherent in the design, it makes sense to scrutinize the class of ciphers in a unified way. In this chapter, we take a closer look at the stream ciphers connected by a common feature that each of them uses (i) one or more arrays[1] as the *main* part of the internal state and (ii) the operation *modular addition* in the *pseudorandom bit generation algorithm*. Apart from *addition* over different *groups* (e.g, GF(2^n) and GF(2)), the stream ciphers under consideration only admit of simple operations such as memory access (direct and indirect) and cyclic rotation of bits, which are typical of any fast stream cipher. In the present discussion we omit the relatively rare class of stream ciphers which may nominally use *array* and *addition*, but their security depends significantly on special functions such as those based on algebraic hard problems, the Rijndael S-box etc.

To the best of our knowledge, the RC4 stream cipher, designed by Ron Rivest in 1987, is the first stream cipher which exploits the features of an array for generating pseudorandom bits, using a few simple operations. Since then a large number of array-based ciphers (or PRBGs) – namely, RC4A [65], VMPC stream cipher [98], IA, IBAA, ISAAC [41], Py [6], Py6 [8], Pypy [7], HC-256 [95], NGG [60], GGHN [36] – have been proposed that are inspired by the RC4 design principles. The Scream family of ciphers [38] also uses arrays and modular additions in their round functions, however, the security of them hinges on a tailor-made function derived from the Rijndael S-box rather than on the mixing of *additions* over different *groups* (e.g., GF(2^n) and GF(2)) and cyclic rotation of bits; therefore, this family of ciphers is excluded from the class of ciphers discussed in the chapter.

First, in Table 6.1, we briefly review the pros and cons of the RC4 stream cipher which is the predecessor of all the ciphers to be analyzed later. Unfortunately, the RC4 cipher is compatible with the old fashioned 8-bit processors only. Except RC4A and the VMPC cipher (which are designed to work on 8-bit processors), all the other ciphers described before are suitable for modern 16/32-bit architectures. Moreover, those 16/32-bit ciphers have been designed with an ambition of incorporating all the positive aspects of RC4, while ruling out its negative properties as listed in Table 6.1. However, the chapter demonstrates that a certain amount of caution is necessary to adapt RC4-like ciphers to 16/32-bit architecture. Here, we mount distinguishing attacks on the ciphers Py6, IA, ISAAC, NGG, GGHN – all of them are designed to suit 16/32-bit processors – with data $2^{68.61}$, $2^{32.89}$,

[1] An array is a data structure containing a set of elements associated with unique indices.

Table 6.1: Pros and cons of the RC4 Cipher

Advantages of RC4	Disadvantages of RC4
Arrays allow for huge *secret* internal state	Not suitable for 16/32-bit architecture
Fast because of fewer operations per round	Several distinguishing attacks
Simple design	Weak Key-setup algorithm
No key recovery attacks better than brute force	

$2^{16.89}$, $2^{32.89}$ and $2^{32.89}$ respectively, by exploiting similar weaknesses in their designs (note that another 32-bit array-based cipher Py has already been attacked in a similar fashion in Chapter 5 and in [17]). Summarily, the attacks on the class of ciphers, described in this chapter, originate from the following basic although not independent facts. However, note that our attacks are based on the assumptions that the key-setup algorithms of the ciphers are 'perfect', that is, after the execution of the algorithms they produce uniformly distributed internal states (more on that in Sect. 6.1.1).

- Array-elements are large (usually of size 16/32 bits), but the array-indices are short (generally of size 8 bits).
- Only a few elements of the arrays undergo changes in consecutive rounds.
- Usage of both pseudorandom index-pointers and pseudorandom array elements in a round, which apparently seems to provide stronger security than the ciphers with fixed pointers, may leave room for attacks arising from the *correlation* between the index-pointers and the corresponding array-elements (see discussion in Sect. 6.2.2).
- Usage of simple operations like *addition* over GF(2^n) and GF(2) in the output generation.

Essentially our attacks based on the above facts have it origins in the *fortuitous states* attack on RC4 by Fluhrer and McGrew [31].

A general framework to attack array-based stream ciphers (or PRBGs) with the above characteristics is discussed in Sect. 6.2. Subsequently in Sect. 6.3.1, 6.3.2 and 6.3.3, as concrete proofs of our argument, we show distinguishing attacks on five stream ciphers. The methods of our attacks are clearly motivated by the *fortuitous states attack* by Fluhrer and McGrew on the RC4 stream cipher [31]. The purpose of the chapter is, by no means, to claim that the array-based ciphers are intrinsically insecure, and therefore, should be rejected without analyzing its merits; rather, we stress that when such a cipher turns out to be extremely fast – such as Py, Py6, IA, ISAAC, NGG, GGHN – an alert message should better be issued for the designers to recheck that they are free from the weaknesses described here. In Sect. 6.5, we comment on the security of three other array-based ciphers (or PRBGs) IBAA, Pypy and HC-256 which, for the moment, do not come under attacks, however they are slower than the ones attacked in this chapter.

Notation. At any round t, some part of the internal state is updated before the output generation and the rest is updated after that. Example: in Algorithm 10, the variables a and m are updated before the output generation in line 5. The variables i and b are updated after or at the same time with output generation. Our convention is: a variable S is denoted by S_t at the time of output generation of round t. As each of the variables is modified in a single line of the corresponding algorithm, after the modification its subscript is incremented. Output at round t is denoted by Z_t.

6.1.1 Assumption

In this chapter, we concentrate solely on the *mixing of bits* by the keystream generation algorithms (i.e., the PRGB) of several array-based stream ciphers and assume that the corresponding key-setup algorithms are *perfect*. A *perfect* key-setup algorithm produces internal state that leaks no statistical information to the attacker. In other words, because of the *difficulty* of deducing any relations between the inputs and outputs of the key-setup algorithm, the internal state produced by the key-setup algorithm is assumed to follow the uniform distribution.

6.2 Stream Ciphers Based on Arrays and Modular Addition

6.2.1 Basic Working Principles

The basic working principles of the PRBG of a stream cipher (see Chapter **??** for a definition of a PRBG), based on one or multiple arrays, are shown in Fig. 6.1. For simplicity, we take snapshots of the internal state, composed of *only* two arrays, at two close rounds denoted by round t and round $t' = t + \delta$. However, our analysis is still valid with more arrays and rounds than just two. Now we delineate the rudiments of the PRBG of such ciphers.

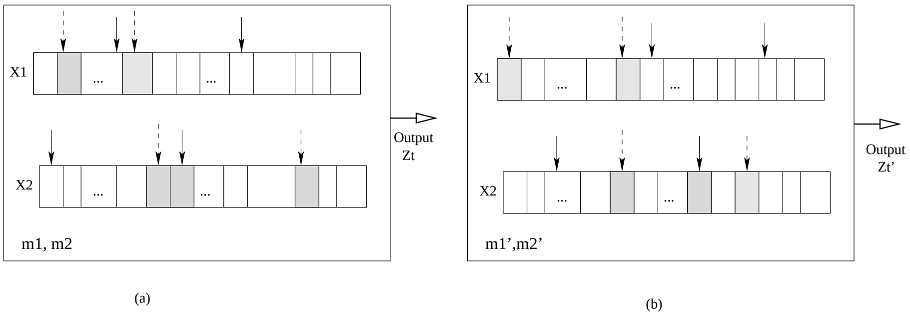

Figure 6.1: Internal State at (a) round t and (b) round $t' = t + \delta$

- **Components:** the internal state of the cipher comprises all or part of the following components.
    1. One or more arrays of n-bit elements (X_1 and X_2 in Fig. 6.1).
    2. One or more variables for indexing into the arrays, i.e., the index-pointers (down arrows in Fig. 6.1).
    3. One or more random variables usually of n-bit length (m_1, m_2, m'_1, m'_2 in Fig. 6.1).
- **Modification to the Internal State at a round.**
    1. *Index Pointers:* the most notable feature of such ciphers is that it has two sets of index pointers. (i) Some of them are fixed or updated in a *known way*, i.e., independent of the secret part of the state (solid arrows in Fig. 6.1) and (ii) the other set of pointers are updated *pseudorandomly*, i.e., based on one or more secret components of the internal state (dotted arrows in Fig. 6.1).
    2. *Arrays*: a few elements of the arrays are updated pseudorandomly based on one or more components of the internal state (the shaded cells of the arrays in Fig. 6.1). Note that, in two successive rounds, only a small number of array-elements (e.g. one or two in each array) are updated. Therefore, most of the array-elements remain identical in consecutive rounds.
    3. *Other variables if any*: they are updated using several components of the internal state.
- **Output generation:** the output generation function at a round is a non-linear combination of different components described above.

6.2.2 Weaknesses and General Attack Scenario

Before assessing the security of array-based ciphers in general, for easy understanding, we first deal with a simple toy-cipher with certain properties (or weaknesses) which induce distinguishing attack on it.

Remark 6.1 *The basis for the attacks, described throughout the chapter including the one in the following example, is searching for internal states for which the outputs can be predicted with bias. This strategy is inspired by the* fortuitous states *attacks by Fluhrer and McGrew on the RC4 stream cipher [31].*

EXAMPLE 6.2 *Let the size of the* internal state *of a stream cipher with the following properties be k bits.*

Property 1 *The outputs Z_{t_1}, Z_{t_2} are as follows.*

$$Z_{t_1} = (X \oplus Y) + (A \lll B)\,, \tag{6.1}$$

$$Z_{t_2} = (M + N) \oplus (C \lll D) \tag{6.2}$$

where X, Y, A, B, M, N, C, D are uniformly distributed and independent.

Property 2 *[Bias-inducing State] If certain k' bits $(0 < k' \leq k)$ of the* internal state *are set to all 0's (denote the occurrence of such state by event E) at round t_1, then the following equations hold.*

$$X = M,\ Y = N,\ B = D = 0,\ A = C.$$

Therefore, (6.1) and (6.2) become

$$Z_{t_1} = (X \oplus Y) + A\,,\ \ Z_{t_2} = (X + Y) \oplus A\,.$$

Now, it follows directly from the above equations that, for a fraction of $2^{-k'}$ of all internal states*,*

$$P\left[Z_{(0)} = (Z_{t_1} \oplus Z_{t_2})_{(0)} = 0 | E\right] = 1. \tag{6.3}$$

Property 3 *If the* internal state *is chosen randomly from the rest of the states, then*

$$P[Z_{(0)} = 0|E^c] = \frac{1}{2}. \tag{6.4}$$

To calculate the overall bias we shall use the following fact known as law of total probability.

Fact 6.3 *Given two mutually exclusive events E and E^c whose probabilities sum to unity,*

$$P[A] = P[A|E] \cdot P[E] + P[A|E^c] \cdot P[E^c],$$

where A is an arbitrary event, and $P[A|B]$ is the conditional probability of A given B (where B is either E or E^c).

Combining (6.3) and (6.4) we get the overall bias for $Z_{(0)}$ using Fact 6.3,

$$\begin{aligned} P[Z_{(0)} = 0] &= \frac{1}{2^{k'}} \cdot 1 + (1 - \frac{1}{2^{k'}}) \cdot \frac{1}{2} \\ &= \frac{1}{2}(1 + \frac{1}{2^{k'}}) . \end{aligned} \tag{6.5}$$

Note that, if the cipher were a secure PRBG then $P[Z_{(0)} = 0] = \frac{1}{2}$. □

Discussion. Now we argue that an array-based cipher has all the three properties of the above example; therefore, the style of attack presented in the example can possibly be applied to an array-based cipher too. First, we discuss the operations involved in the output generation of the PRBG. Let the internal state consist of N arrays and M other variables. At round t, the arrays are denoted by $S_{1,t}[\cdot]$, $S_{2,t}[\cdot], \cdots$, $S_{N,t}[\cdot]$ and the variables by $m_{1,t}$, $m_{2,t}, \cdots$, $m_{M,t}$. We observe that the output Z_t is of the following form,

$$\begin{aligned} Z_t \quad = \quad & \mathsf{ROT}[\cdots \mathsf{ROT}[\mathsf{ROT}[\mathsf{ROT}[V_{1,t}] \circledast \mathsf{ROT}[V_{2,t}]] \\ & \circledast \mathsf{ROT}[V_{3,t}]] \circledast \cdots \circledast \mathsf{ROT}[V_{k,t}]] \end{aligned} \tag{6.6}$$

where $V_{i,t} = m_{g,t}$ or $S_{j,t}[I_l]$; $\mathsf{ROT}[\cdot]$ is the cyclic rotation function either constant or variable depending on the secret state; the function $\circledast[\cdot,\cdot]$ is either *bit-wise* XOR or *addition* modulo 2^n.

Now we describe a general technique to establish a distinguishing attack on an array-based cipher (or PRBG) from the above information. We recall that, at the first round (round t_1 in the present context), the internal state is assumed to be uniformly distributed (see Sect. 6.1.1).

Step 1. [Analogy with Property 1 of Example 6.2] Observe the elements of the internal state which are involved in the outputs Z_{t_1}, $Z_{t_2}, \cdots$ (i.e., the $V_{i,t}$'s in (6.6)) when the rounds in question are close ($t_1 < t_2 < \cdots$).

Step 2. [Bias-inducing state, Analogy with Property 2 of Example 6.2] Fix a few bits of some array elements (or fix a relation among them) at the initial round t_1 such that *indices* of array-elements in later rounds can be predicted with probability 1 or close to it. More specifically, we search for a *partially specified internal state* such that one or both of the following cases occur due to predictable *index-pointers*.

1. The $V_{i,t}$'s involved in Z_{t_1}, $Z_{t_2}, \cdots$ are those array-elements whose bits are already fixed.

2. Each $V_{i,t}$ is dependent on one or more other variables in Z_{t_1}, $Z_{t_2}, \cdots$.

Now, for this case, we compute the bias in the output bits. Below we identify the reasons why an array-based cipher can potentially fall into the above scenarios.

Reason 1 Usually, an array-based cipher uses a number of pseudorandom index-pointers which are updated by the elements of the array. This fact turns out to be a weakness, as fixed values (or a relation) can be assigned to the array-elements such that the index-pointers fetch values from known locations. In other words, the weakness results from the correlation between index-pointers and array-elements which are uniformly distributed individually but not independent of each other.

Reason 2 Barring a few, most of the array-elements do not change in rounds which are close to each other. Therefore, by fixing bits, it is sometimes easy to force the pseudorandom index-pointers to fetch certain elements from the arrays in successive rounds.

Reason 3 The size of an index-pointer is small, usually 8 bits irrespective of the size of an array-element which is either 16 bits or 32 bits or 64 bits. Therefore, fixing a small number of bits of the array-elements, it is possible to assign appropriate values to the index-pointers. The fewer the number of fixed bits, the greater is the bias (note the parameter k' in (6.5)).

Reason 4 If the rotation operations in the output function are determined by pseudorandom array elements (see (6.6)) then fixing a few bits of internal state can simplify the function by freeing it from rotation operations. In many cases rotation operations are not present in the function. In any case the output function takes the following form.

$$Z_t = V_{1,t} \circledast V_{2,t} \circledast V_{3,t} \circledast \cdots \circledast V_{k,t}.$$

Irrespective of whether '$\circledast$' denotes '$\oplus$' or '$+$', the following equation holds for the *lsb* of Z_t.

$$Z_{t(0)} = V_{1,t_{(0)}} \oplus V_{2,t_{(0)}} \oplus V_{3,t_{(0)}} \oplus \cdots \oplus V_{k,t_{(0)}}.$$

Now by adjusting the index-pointers through fixing bits, if certain equalities among the $V_{i,t}$'s are ensured then $\bigoplus^{t} Z_{t(0)} = 0$ occurs with probability 1 rather than probability 1/2.

Step 3. [Analogy with Property 3 of EXAMPLE 6.2] Prove or provide strong evidence that, for the rest of the states other than the bias-inducing state, the bias generated in the previous step is not counterbalanced.

REASON The internal state of such cipher is huge and uniformly distributed at the initial round. The correlation, detected among the indices and array-elements in Step 2, is fortuitous although not entirely surprising because the variables are not independent. Therefore, the possibility that a bias, produced by an accidental *state*, is *totally* counterbalanced by another accidental state is negligible. In other words, if the bias-inducing state, as explained in Step 2, does not occur, it is likely that at least one of the $V_{i,t}$'s in (6.6) is uniformly distributed and independent; this fact ensures that the outputs are also uniformly distributed and independent.

Step 4. [Analogy with (6.5) of EXAMPLE 6.2] Estimate the overall bias from the results in Step 2 and Step 3. □

In the next section, we attack several array-based ciphers (or PRBGs) following the methods described in this section.

6.3 Distinguishing Attacks on Array-based Ciphers

This section describes distinguishing attacks on the ciphers (or PRBGs) Py, Py6, IA, ISAAC, NGG and GGHN – each of which is based on *arrays* and *modular addition*. A full description of the ciphers is omitted; the reader is kindly referred to the corresponding design papers for details. For each of the ciphers, our task is essentially two-forked as summed up below.

1. **Identification of a Bias-inducing State.** This state is denoted by the event E which adjusts the *index-pointers* in such a way that the *lsb*s of the outputs are biased. The *lsb*s of the outputs are potentially vulnerable as they are generated without any carry bits which are nonlinear combinations of input bits (see Step 2 of the general technique described in Sect. 6.2.2).

2. **Computation of the Probability of Overall Bias.** The probability is calculated considering both E and E^c. As suggested in Step 3 of Sect. 6.2.2, for each cipher (or PRBG), the *lsb*s of the outputs are uniformly distributed if the event E does not occur under the assumption mentioned in Sect. 6.1.1.

Note. For each of the five ciphers attacked in the subsequent sections, simple observations show that, if E (i.e., the bias-inducing state) does not occur then the variable under investigation is uniformly distributed under the assumption of uniformly distributed *internal state* after the key-setup algorithm. We omit the formal proofs.

6.3.1 Bias in the Outputs of Py6

The stream cipher Py6, designed especially for fast software applications by Biham and Seberry in 2005, is one of the modern ciphers that are based on arrays [6, 8].[2] Although the cipher Py, a variant of Py6, was successfully attacked in Chapter 5 and [17], Py6 has so far remained alive. The PRBG of Py6 is described in Algorithm 8 (see [6, 8] for a detailed discussion).

Algorithm 8 Single Round of Py6

Input: $Y[-3, ..., 64]$, $P[0, ..., 63]$, a 32-bit variable s
Output: 64-bit random output
/*Update and rotate P*/
swap $(P[0], P[Y[43]\&63])$;
rotate (P);
/* Update s*/
$s+ = Y[P[18]] - Y[P[57]]$;
$s = ROTL32(s, ((P[26] + 18)\&31))$;
/* Output 8 bytes (least significant byte first)*/
output $((ROTL32(s, 25) \oplus Y[64]) + Y[P[8]])$;
output $((\quad s \quad \oplus Y[-1]) + Y[P[21]])$;
/* Update and rotate Y*/
$Y[-3] = (ROTL32(s, 14) \oplus Y[-3]) + Y[P[48]]$;
rotate(Y);

Bias-producing State of Py6. Below we identify five conditions among the elements of the S-box P, for which the distribution of $Z_{1,1} \oplus Z_{2,3}$ is biased ($Z_{1,t}$ and $Z_{2,t}$ denote the lower and upper 32 bits of output respectively, at round t).

C1. $P_2[26] \equiv -18 (\bmod 32)$; **C2.** $P_3[26] \equiv 7 (\bmod 32)$; **C3.** $P_2[18] = P_3[57] + 1$; **C4.** $P_2[57] = P_3[18] + 1$; **C5.** $P_1[8] = 1$; **C6.** $P_3[21] = 62$.

Let the event E denote the simultaneous occurrence of the above conditions ($P[E] \approx 2^{-33.86}$). Theorem 6.4 shows that, if E occurs then $Z_{(0)} = 0$ where Z denotes $Z_{1,1} \oplus Z_{2,3}$.

[2]The cipher has been submitted to the ECRYPT Project [25].

Theorem 6.4 $Z_{1,1(0)} = Z_{2,3(0)}$ *if the following six conditions on the elements of the S-box P are simultaneously satisfied.*

1. $P_2[26] \equiv -18 \bmod 32$,
2. $P_3[26] \equiv 7 \bmod 32$,
3. $P_2[18] = P_3[57] + 1$,
4. $P_2[57] = P_3[18] + 1$,
5. $P_1[8] = 1$,
6. $P_3[21] = 62$.

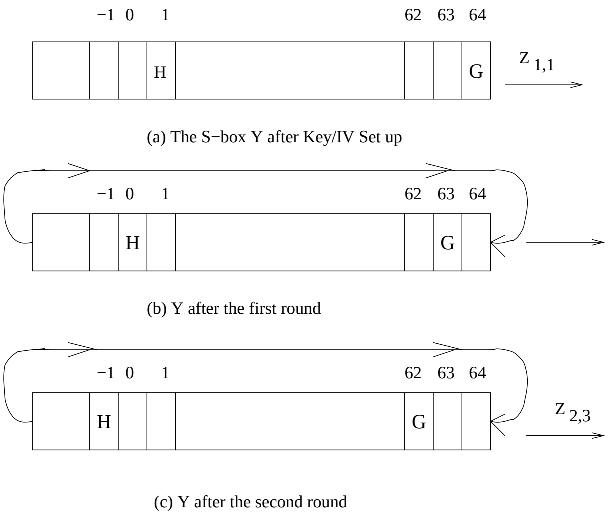

Figure 6.2: Py6: (a) $P_1[8] = 1$ (condition 5): G and H are used in $Z_{1,1}$, (b) Y_2 (i.e., Y after the 1^{st} round), (c) $P_3[21] = 62$ (condition 6): G and H are used in $Z_{2,3}$

Proof. The formulas for the $Z_{1,1}$, $Z_{2,3}$ and s_2 are given below (see Algorithm 8).

$$Z_{1,1} = (ROTL32(s_1, 25) \oplus Y_1[64]) + Y_1[P_1[8]]\,, \tag{6.7}$$

$$Z_{2,3} = (s_3 \oplus Y_3[-1]) + Y_3[P_3[21]]\,, \tag{6.8}$$

$$\begin{aligned} s_2 = {} & ROTL32(s_1 + Y_2[P_2[18]] - Y_2[P_2[57]], \\ & P_2[26] + 18 \bmod 32)\,. \end{aligned} \tag{6.9}$$

• Condition 1 (i.e., $P_2[26] \equiv -18 \bmod 32$) reduces (6.9) to

$$s_2 = s_1 + Y_2[P_2[18]] - Y_2[P_2[57]]\,.$$

• Condition 2 (i.e., $P_3[26] \equiv 7 \bmod 32$) together with Condition 1 implies

$$s_3 = ROTL32((s_1 + Y_2[P_2[18]] - Y_2[P_2[57]] + Y_3[P_3[18]] - Y_3[P_3[57]]), 25)\,.$$

• Condition 3 and Condition 4 (that is, $P_2[18] = P_3[57]+1$ and $P_2[57] = P_3[18]+1$) reduce the previous equation to

$$s_3 = ROTL32(s_1, 25)\,. \tag{6.10}$$

From (6.7), (6.8), (6.10) we get:

$$\begin{aligned} Z_{1,1} &= (ROTL32(s_1, 25) \oplus Y_1[64]) + Y_1[P_1[8]]\,, && (6.11)\\ Z_{2,3} &= (ROTL32(s_1, 25) \oplus Y_3[-1]) + Y_3[P_3[21]]\,. && (6.12)\end{aligned}$$

In Fig. 6.2, condition 5 and 6 are described. According to the figure,

$$\begin{aligned} H &= Y_1[P_1[8]] = Y_3[-1]\,, && (6.13)\\ G &= Y_1[64] = Y_3[P_3[21]]\,. && (6.14)\end{aligned}$$

Applying (6.13) and (6.14) in (6.11) and (6.12) we get,

$$Z_{1,1(0)} \oplus Z_{2,3(0)} = Y_1[64]_{(0)} \oplus Y_1[P_1[8]]_{(0)} \oplus Y_3[-1]_{(0)} \oplus Y_3[P_3[21]]_{(0)} = 0\,.$$

This completes the proof. $\square$

Now, using Fact 6.3, we calculate $P[Z_{(0)} = 0]$. Note that $P[E] = 2^{-33.86}$ and $P[Z_{(0)} = 0|E^c] = 0.5$.

$$\begin{aligned} P[Z_{(0)} = 0] &= P[Z_{(0)} = 0|E] \cdot P[E] + P[Z_{(0)} = 0|E^c] \cdot P[E^c]\\ &= 1 \cdot 2^{-33.86} + \frac{1}{2} \cdot (1 - 2^{-33.86})\\ &= \frac{1}{2} \cdot (1 + 2^{-33.86})\,. && (6.15)\end{aligned}$$

Note that, if Py6 had been an ideal PRBG then the above probability would have been exactly $\frac{1}{2}$.

Remark 6.5 *The above bias can be generalized for rounds t and $t+2$ ($t > 0$) rather than only rounds 1 and 3.*

Remark 6.6 *The main difference between Py and Py6 is that the locations of S-box elements used by one cipher is different from those by the other. The significance of the above results is that it shows that changing the locations of array-elements is futile if the cipher retains some intrinsic weaknesses as explained in Sect. 6.2.2. We shall later see that Py was attacked with $2^{84.7}$ data while Py6 can be attacked with $2^{68.61}$ (explained in Sect. 6.4)*

6.3.2 Biased Outputs in IA and ISAAC

At FSE 1996, R. Jenkins Jr. proposed two fast PRBGs, namely IA and ISAAC, along the lines of the RC4 stream cipher [41]. The round functions of IA and ISAAC are shown in Algorithm 9 and Algorithm 10. Each of them uses an array of 256 elements. The size of an array-element is 16 bits for IA and 32 bits for ISAAC. However, IA and ISAAC can be adapted to work with array-elements of larger size too. For ISAAC, the earlier attack was by Pudovkina who claimed to have deduced its internal state with time $4 \cdot 67 \cdot 10^{1240}$ which was way more than the exhaustive search through the keys of usual size of 256-bit or 128-bit [73]. On the other hand, we shall see later in Sect. 6.4 that our distinguishing attacks can be built with much lower time complexities.

Algorithm 9 PRBG of IA

Input: $m[0, 1, ...255]$, 16-bit random variable b
Output: 16-bit random output

1: $i = 0$;
2: $x = m[i]$;
3: $m[i] = y = m[ind(x)] + b \bmod 2^{16}$; /* $ind(x) = x_{(7,0)}$ */
4: Output= $b = m[ind(y \gg 8)] + x \bmod 2^{16}$;
5: $i = i + 1 \bmod 256$;
6: Go to step 2;

Bias-inducing State of IA. Let $m_t[i_t + 1 \bmod 256] = a$. If the following condition

$$ind((a + Z_t) \gg 8) = ind(a) = i_{t+1} \tag{6.16}$$

is satisfied then

$$Z_{(0)}(= Z_{t(0)} \oplus Z_{t+1(0)}) = 0$$

A proof is in Theorem 6.7.

Theorem 6.7 *Let $m_t[i_t + 1 \bmod 256] = a$. If the following condition*

$$ind((a + Z_t) \gg 8) = ind(a) = i_{t+1}$$

is satisfied then

$$Z_{t+1} = 2 \cdot a + Z_t \bmod 2^{16} \Rightarrow Z_{t(0)} \oplus Z_{t+1(0)} = 0\,.$$

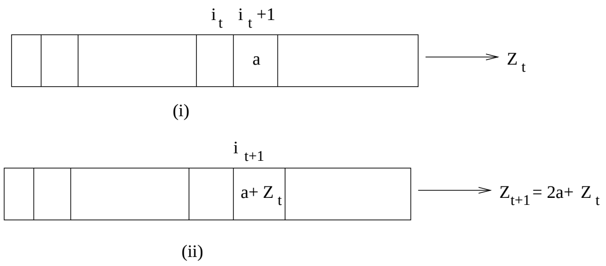

Figure 6.3: IA: (i) at round t when $m_t[i_t + 1] = a$, (ii) at round $t + 1$

Proof. This follows directly from Algorithm 9. In Fig. 6.3, the configuration of the array m and the output are shown for two consecutive rounds when the condition is satisfied. □

Let event E occur when (6.16) holds good. Note that $P[E] = 2^{-16}$ and $P[Z_{(0)} = 0|E^c] = 0.5$ assuming a and Z_t are independent and uniformly distributed. Now, using Fact 6.3, we calculate $P[Z_{(0)} = 0]$.

$$P[Z_{(0)} = 0] = \frac{1}{2} \cdot (1 + 2^{-16}) . \tag{6.17}$$

Algorithm 10 PRBG of ISAAC

Input: $m[0, 1, ...255]$, two 32-bit random variables a and b
Output: 32-bit random output

1: $i = 0$;
2: $x = m[i]$;
3: $a = a \oplus (a \ll R) + m[i + K \text{ mod } 256] \text{ mod } 2^{32}$;
4: $m[i + 1] = y = m[ind(x)] + a + b \text{ mod } 2^{32}$;/* $ind(x) = x_{(7,0)}$ */
5: Output= $b = m[ind(y \gg 8)] + x \text{ mod } 2^{32}$;
6: $i = i + 1 \text{ mod } 256$;
7: Go to Step 2.

Bias-inducing State of ISAAC. For ease of understanding, we rewrite the PRBG of the ISAAC in a simplified manner in Algorithm 10. The variables R and K, described in step 3 of Algorithm 10, depend on the parameter i (see [41] for details); however, we show that our attack can be built independent of those variables.

Let $m_{t-1}[i_t] = x$. Let event E occur when the following equation is satisfied.

$$ind((m_{t-1}[ind(x)] + a_t + b_{t-1}) \gg 8) = i_t. \tag{6.18}$$

If E occurs then $Z_t = x + x \bmod 2^{32}$, i.e., $Z_{t(0)} = 0$ (see Theorem 6.8 for a proof).

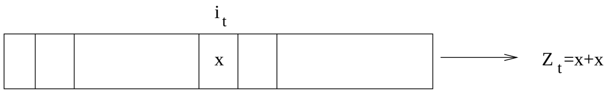

Figure 6.4: ISAAC: at round t

Theorem 6.8 *Let $m_{t-1}[i_t] = x$. If the following condition*

$$ind((m_{t-1}[ind(x)] + a_t + b_{t-1}) \gg 8) = i_t \tag{6.19}$$

is satisfied then

$$Z_t = x + x \bmod 2^{32} \Rightarrow Z_{t(0)} = 0\,. \tag{6.20}$$

PROOF. The claim can be easily verified from Algorithm 10. In Fig. 6.4, the configuration of the *internal state* at round t is shown. □

As a_t, b_{t-1} and x are independent and each of them is uniformly distributed over $\mathbb{Z}_{2^{32}}$, the following equation captures the bias in the output using Fact 6.3.

$$P[Z_{t(0)} = 0] = \frac{1}{2} \cdot (1 + 2^{-8}) \tag{6.21}$$

where $P[E] = 2^{-8}$.

6.3.3 Biases in the Outputs of NGG and GGHN

Gong *et al.* very recently have proposed two array-based ciphers NGG and GGHN with 32/64-bit word-length [60, 36] for very fast software applications. The PRBGs of the ciphers are described in Algorithm 11 and 12. Both the ciphers are claimed to be more than three times faster RC4. Due to the introduction of an extra 32-bit random variable k, the GGHN is evidently a stronger version of NGG. We propose attacks on both the ciphers based on the general technique described in Sect. 6.2.2. Note that the NGG cipher was already experimentally attacked by Wu without theoretical quantification of the attack parameters such as bias, required outputs [96]. For NGG, our attack is new, theoretically justifiable and most importantly, conforms to the basic weaknesses of an array-based cipher, as explained in Sect. 6.2.2. For GGHN, our attack is the first attack on the cipher.

Algorithm 11 Pseudorandom Bit Generation of NGG

Input: $S[0, 1, ...255]$
Output: 32-bit random output
1: $i = 0,\ j = 0$;
2: $i = i + 1 \bmod 256$;
3: $j = j + S[i] \bmod 256$;
4: Swap $(S[i], S[j])$;
5: Output= $S[S[i] + S[j] \bmod 256]$;
6: $S[S[i] + S[j] \bmod 256] = S[i] + S[j] \bmod 2^{32}$
7: Go to step 2;

Bias-inducing State of NGG. Let the event E occur, if $i_t = j_t$ and $S_{t+1}[i_{t+1}] + S_{t+1}[j_{t+1}] = 2 \cdot S_t[i_t] \bmod 256$. We observe that, if E occurs then $Z_{t+1(0)} = 0$ (see Theorem 6.9).

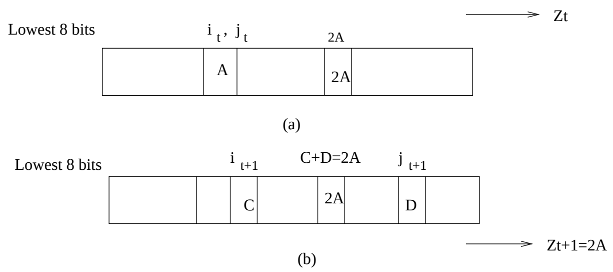

Figure 6.5: NGG: (a) the array S at the end of round t, (b) the array S just before output generation at round $t + 1$

Theorem 6.9 *If (i) $i_t = j_t$ and (ii) $S_{t+1}[i_{t+1}] + S_{t+1}[j_{t+1}] = 2 \cdot S_t[i_t] \bmod 256$, then*

$$Z_{t+1} = 2 \cdot S_t[i_t] \bmod 2^{16} \Rightarrow Z_{t+1(0)} = 0$$

assuming that $i_{t+1} \neq 2 \cdot S_t[i_t] \bmod 256$ and $j_{t+1} \neq 2 \cdot S_t[i_t] \bmod 256$.[3]

PROOF. The proof can be easily followed from Fig. 6.5. □

Now we compute $P[Z_{t+1(0)} = 0]$ where $P[E] = 2^{-16}$ in a similar manner as before.

$$P[Z_{t+1(0)} = 0] = \frac{1}{2} \cdot (1 + 2^{-16}) \,. \tag{6.22}$$

Algorithm 12 Pseudorandom Bit Generation of GGHN

Input: $S[0, 1, ...255]$, k

Output: 16-bit random output

1: $i = 0$, $j = 0$;
2: $i = i + 1 \bmod 256$;
3: $j = j + S[i] \bmod 256$;
4: $k = k + S[j] \bmod 2^{32}$;
5: Output= $S[S[i] + S[j] \bmod 256] + k \bmod 2^{32}$;
6: $S[S[i] + S[j] \bmod 256] = k + S[i] \bmod 2^{32}$;
7: Go to step 2;

Bias-producing State of GGHN. In Theorem 6.10 it is shown that, if $S_t[i_t] = S_{t+1}[j_{t+1}]$ and $S_t[j_t] = S_{t+1}[i_{t+1}]$ (denote it by event E) then $Z_{t+1(0)} = 0$.

Theorem 6.10 *If $S_t[i_t] = S_{t+1}[j_{t+1}]$ and $S_t[j_t] = S_{t+1}[i_{t+1}]$ then*

$$Z_{t+1} = 2 \cdot (k + S_t[i_t]) \bmod 2^{32} \Rightarrow Z_{t+1(0)} = 0 \,.$$

PROOF. From Algorithm 12 and Fig. 6.6 the proof can be ascertained. □

Now, applying Fact 6.3, we compute $P[Z_{t+1(0)} = 0]$ where $P[E] = 2^{-16}$.

$$P[Z_{t+1(0)} = 0] = \frac{1}{2} \cdot (1 + 2^{-16}) \,. \tag{6.23}$$

[3]The assumption has negligible effect on the computed probability.

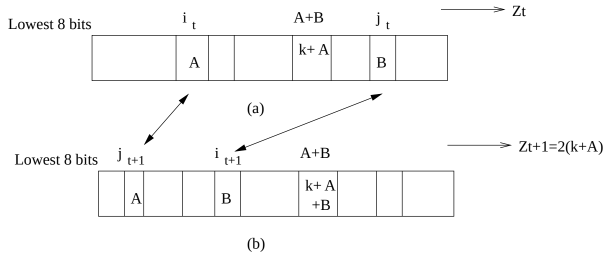

Figure 6.6: GGHN: (a) the array S at the end of round t, (b) the array S at the end of round $t+1$

6.4 Data and Time of the Distinguishing Attacks

In the section we compute the data and time complexities of the distinguishers derived from the biases computed in the previous sections. See Sect. 1.3.3 for a detailed discussion on distinguishers and the advantage of a distinguisher.

Let there be n binary random variables z_1, z_2, $\cdots$, z_n which are independent of each other and each of them follows the distribution $\mathsf{D_{BIAS}}$. Let the uniform distribution on alphabet $\mathbb{Z}_2$ be denoted by $\mathsf{D_{UNI}}$. A method to construct an *optimal distinguisher* with a fixed number of samples has been described by Baignères *et al.* [3].[4] While the detailed description of an *optimal distinguisher* is omitted, the following theorem determines the number of samples required by an *optimal distinguisher* to attain an *advantage* of 0.5 which is considered a reasonable goal.

Table 6.2: Data and time of the distinguishers with advantage exceeding 0.5

PRBG	M	Bytes of a single stream $= 0.4624 \cdot M^2$	Time
Py6	$2^{34.86}$	$2^{68.61}$	$\mathcal{O}(2^{68.61})$
IA	2^{17}	$2^{32.89}$	$\mathcal{O}(2^{32.89})$
ISAAC	2^{9}	$2^{16.89}$	$\mathcal{O}(2^{16.89})$
NGG	2^{17}	$2^{32.89}$	$\mathcal{O}(2^{32.89})$
GGHN	2^{17}	$2^{32.89}$	$\mathcal{O}(2^{32.89})$

[4]Given a fixed number of samples, an *optimal distinguisher* attains the maximum advantage.

Theorem 6.11 *Let the input to an* optimal distinguisher *be a realization of the binary random variables z_1, z_2, $z_3, \cdots, z_n$ where each z_i follows $\mathsf{D}_{\mathit{BIAS}}$. To attain an advantage of more than 0.5, the least number of samples required by the* optimal distinguisher *is given by the following formula*

$$n = 0.4624 \cdot M^2 \quad \textit{where}$$

$$P_{\mathsf{D}_{BIAS}}[z_i = 0] - P_{\mathsf{D}_{UNI}}[z_i = 0] = \frac{1}{M}\,.$$

Proof. See Sect. 5.5 for the proof. □

Now $\mathsf{D}_{\mathrm{UNI}}$ is known and $\mathsf{D}_{\mathrm{BIAS}}$ can be determined from (6.15) for Py6, (6.17) for IA, (6.21) for ISAAC, (6.22) for NGG, (6.23) for GGHN. In Table 6.2, we list the data and time complexities of the distinguishers. Our experiments agree well with the theoretical results. The constant in $\mathcal{O}(m)$ is determined by the time taken by single round of the corresponding cipher; note that this is an abuse of notation.

6.5 A Note on IBAA, Pypy and HC-256

We are still unable to mount distinguishing attacks on the array-based PRBGs IBAA, Pypy and HC-256, using the method described in this chapter. The IBAA works in a similar way as the ISAAC works, except for the variable a which plays an important role in the output generation of IBAA [41]. It seems that a relation has to be discovered among the values of the parameter a at different rounds to successfully attack IBAA. Pypy is a slower variant of Py and Py6 [7]. Pypy produces 32 bits per round when each of Py and Py6 produces 64 bits. To attack Pypy, in a fashion described in this chapter, a relation needs to be found among the elements which are separated by at least three rounds. However, please note that very recently Wu and Preneel have reported key recovery attacks on Pypy using chosen IVs [97]. To attack HC-256 [95], some correlations need to be known among the elements which are cyclically rotated by constant number of bits.

6.6 Conclusion

In this chapter, we have studied array-based stream ciphers in a general framework to assess their resistance against certain distinguishing attacks originating from the correlation between index-pointers and array-elements. We show that the weakness becomes more profound because of the usage of simple modular additions in the output generation function. In the unified framework we have attacked five modern array-based stream ciphers (or PRBGs) Py6, IA, ISAAC, NGG, GGHN with data complexities $2^{68.61}$, $2^{32.89}$, $2^{16.89}$, $2^{32.89}$ and $2^{32.89}$ respectively. We also note that some other array-based ciphers IBAA, Pypy, HC-256 still do not come under any attacks, however, the algorithms need to be analyzed more carefully in order to be considered secure. We believe that our investigation will throw light on the security of array-based stream ciphers in general and can possibly be extended to analyze other types of ciphers.

Chapter 7

Conclusions and Future Work

Nature's music is never over; her silences are pauses, not conclusions.
-Mary Webb (1881-1927)

7.1 Results of the Thesis: In a Nutshell

In the first part of the thesis we dealt with a certain class of equations that combined modular additions over two different algebraic groups. These equations are known as *differential equations of addition* (DEA). DEA are of the following forms:

$$(x+y) \oplus ((x \oplus \alpha) + (y \oplus \beta)) = \gamma,$$
$$(x+y) \oplus (x + (y \oplus \beta)) = \gamma,$$

where x and y are the only unknown variables. Combination of additions over various algebraic groups is considered a fundamental building block in the designs of symmetric cipher primitives. We first establish that the satisfiability of such equations is in the complexity class $\mathcal{P}$. Going further, we also solve those equations using a novel technique which is based on simple combinatorial observations rather than usual algebraic methods such as Buchberger's algorithm for computing Gröbner bases [2], F_4 [26] and F_5 [27]. Using these results, we are able to recover the secret key of the Helix cipher with both adaptive chosen plaintexts and chosen plaintexts.

Next we comprehensively studied a class of stream ciphers connected by the properties that each used (i) arrays as main components of their *internal state* and (ii) modular additions in the PRBG to generate output streams. We devised a unified framework to evaluate the resistance of these ciphers against certain kinds of distinguishing attacks. The attacks can be interpreted as the extension, modification and further insightful explanation of the *fortuitous states attacks* invented by Fluhrer and McGrew to cryptanalyze the RC4 cipher in 2000. Our investigations affirm that, without certain precautions, any array-based stream cipher (or PRBG) can come under distinguishing attacks originating from the relations between the indices and the corresponding elements of the array. To establish the fact, exploiting the above mentioned weaknesses, we describe distinguishing attacks on 8 practical array-based ciphers namely RC4, RC4A, Py, Py6, GGHN, NGG, ISAAC and IA.

7.2 Open Problems

We identify a number of ways to improve and extend our work.

- Many modern ciphers, such as IDEA [49, 50], RC6 [79], use *modular multiplications* in addition to *modular additions* over various groups. Therefore, the next question that immediately comes to our mind is: how to solve the following types of equations in the manners described in Chapter 2.

$$(x \circledast y) \circledast ((x \circledast \alpha) \circledast (y \circledast \beta)) = \gamma,$$
$$(x \circledast y) \circledast (x \circledast (y \circledast \beta)) = \gamma,$$

  where the function $\circledast[\cdot, \cdot]$ can be any of the following binary operations:

  - *addition* modulo 2^n $(+)$,
  - *multiplication* modulo 2^n $(\odot)$,
  - bitwise *XOR* $(\oplus)$,
  - bitwise *or* $(\vee)$,
  - bitwise *and* $(\wedge)$.

  The above equations as well as the ones explored in the thesis can be broken into a set of multivariate polynomial equations over GF(2). It is a well established fact that the satisfiability of an arbitrary set of multivariate polynomial equations over any field is an $\mathcal{NP}$-complete problem [94]. Recently, substantial research effort has been spent on inventing faster techniques to solve multivariate polynomial equations (e.g. XL algorithm [16], XSL algorithm [15]). However, the efficiency of those algorithms has also been disputed by many [13, 21]. It is important to note that, to attack a symmetric cipher component, it may not be necessary to search for an algorithm which solves any arbitrary set of multivariate polynomial equations. As the components of symmetric ciphers are mainly composed of operations such as $+$, $-$, $\oplus$ and $\odot$, where each of them combines n-bit integers, it is reasonable to assume that a weaker algorithm may suffice to attack the cipher, if we are able to discover some fortuitous patterns among the bits of the variables as we did for the DEA in the thesis. Therefore, a possible direction to carry on our work is to design algorithms to solve the above equations and then take on more complex equations based on them.

- Secondly, in Chapter 5 and 6 most of the distinguishing attacks were based on the biased *lsb*s where the carry bits of the respective expressions were each zero. However, it is apparent that the higher order carry bits also occur with biases with diminishing magnitudes (see Sect. 5.6 for an example). It is not yet known how to combine all the biased bits *optimally* to construct an *optimal distinguisher*. Paul Crowley has made an attempt to combine the biases of several bits of Py outputs using a Hidden Markov Model [17], however, it seems plausible that a better distinguisher can be built.

- Next, what appears to be the most challenging question at present is to express the array-based ciphers as a set of algebraic equations in terms of key, IV, plaintext and ciphertext and then to solve the equations for the secret key. Till now no such attempts have been made for such types of ciphers.

- The thesis dealt with a large number of ciphers which used one or more arrays as the main components of their internal states. In the literature, many more sophisticated data structures are also available rather than just arrays. Heap, various types of trees, pushdown automata, stacks, queues are some of them that seem to be potential candidates to be used in stream ciphers. Therefore, analyzing different data structures for the purpose of constructing symmetric ciphers may be a useful way to extend our work.

- Lastly, it is still a challenging open problem to recover the secret key of the RC4 cipher (with 128-bit key) with running time which is less than the exhaustive search through all possible keys.

Bibliography

[1] A. Aho, J. Hopcroft, J. Ullman, "The Design and Analysis of Computer Algorithms," Addison-Wesley, 1974.

[2] I. A. Ajwa, Z. Liu, P. S. Wang, "Gröbner Bases Algorithm," *ICM Technical Report*, February 1995, Available Online at `http://icm.mcs.kent.edu/reports/1995/gb.pdf`.

[3] T. Baignères, P. Junod and S. Vaudenay, "How Far Can We Go Beyond Linear Cryptanalysis?," *Asiacrypt 2004* (P. Lee, ed.), vol. 3329 of *LNCS*, pp. 432-450, Springer-Verlag, 2004.

[4] T. A. Berson, "Differential Cryptanalysis Mod 2^{32} with Applications to MD5," *Eurocrypt 1992* (R. A. Rueppel, ed.), vol. 658 of *LNCS*, pp. 71-80, Springer-Verlag, 1993.

[5] E. Biham, Personal Communication, Dec. 2005.

[6] E. Biham, J. Seberry, "Py (Roo): A Fast and Secure Stream Cipher using Rolling Arrays," eSTREAM, ECRYPT Stream Cipher Project, Report 2005/023, 2005.

[7] E. Biham, J. Seberry, "Pypy: Another Version of Py," eSTREAM, ECRYPT Stream Cipher Project, Report 2006/038, 2006.

[8] E. Biham, J. Seberry, "C Code of Py6," as available from `http://www.ecrypt.eu.org/stream/py.html`, eSTREAM, ECRYPT Stream Cipher Project, 2005.

[9] E. Biham, A. Shamir, "Differential Cryptanalysis of DES-like Cryptosystems," *Crypto '90* (A. Menezes, S. A. Vanstone, eds.), vol. 537 of *LNCS*, pp. 2-21, Springer-Verlag, 1991.

[10] A. Biryukov, D. Wagner, “Slide Attacks,” *Fast Software Encryption 1999*, (Lars R. Knudsen, ed.), vol. 1636 of *LNCS*, pp. 245-259, Springer-Verlag, 1999.

[11] M. Blum, S. Micali, “How to Generate Cyptographically Strong Sequence of Psudo-random Bits,” *Siam Journal of Computing*, vol. 13, No. 4, pp. 850-864, November 1984.

[12] C. Burwick, D. Coppersmith, E. D’Avignon, Y. Gennaro, S. Halevi, C. Jutla, S. M. Matyas Jr., L. O’Connor, M. Peyravian, D. Safford and N. Zunic, “MARS – A Candidate Cipher for AES,” Available Online at `http://www.research.ibm.com/security/mars.html`, June 1998.

[13] C. Cid, G. Leurent, “An analysis of the XSL algorithm”, *Advances in Cryptology–Asiacrypt 2005* (B. Roy, ed.), vol. 3788, pp. 333-352, *LNCS*, 2005.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *“Introduction to Algorithms,”* MIT Press.

[15] N. Courtois, J. Pieprzyk, “Cryptanalysis of Block Ciphers with Overdefined Systems of Equations,” *Asiacrypt 2002* (Yuliang Zheng, ed.), vol. of *LNCS*, pp. 267-287, Springer-Verlag, 2002.

[16] N. Courtois, A. Klimov, J. Patarin, and A. Shamir, “Efficient algorithms for solving overdefined systems of multivariate polynomial equations,” *Advances in Cryptology – EUROCRYPT 2000* (B. Preneel, ed.), vol. 1807, Lecture Notes in Computer Science, pages 392-407. Springer-Verlag, 2000.

[17] P. Crowley, “Improved Cryptanalysis of Py,” *Workshop Record of SASC 2006 – Stream Ciphers Revisited*, ECRYPT Network of Excellence in Cryptology, February 2006, Leuven (Belgium), pp. 52-60.

[18] J. Daemen, J. Lano, and B. Preneel, “Chosen ciphertext attack on SSS,” In SASC 2006 - Stream Ciphers Revisited, pages 45-51. ECRYPT, 2006.

[19] J. Daemen, V. Rijmen, “The Design of Rijndael: AES – The Advanced Encryption Standard,” Springer-Verlag, 2002.

[20] D. J. Bernstein, “Comparison of 256-bit stream ciphers at the beginning of 2006,” *Workshop Record of SASC 2006 – Stream Ciphers Revisited*, ECRYPT Network of Excellence in Cryptology, pp. 70-83.

[21] C. Diem, “The XL-algorithm and a conjecture from commutative algebra,” *Advances in Cryptology - ASIACRYPT 2004* (P. Lee, ed.), vol. 3329, Lecture Notes in Computer Science, pp. 323-337, Springer-Verlag, 2004.

[22] W. Diffie and M. E. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory,* vol. IT–22, Nov. 1976, pp. 644-654.

[23] H. Dobbertin, A. Bosselaers and B. Preneel, *RIPEMD-160: A Strengthened Version of RIPEMD*, *Proceedings of Fast Software Encryption 1996* (D. Gollmann, ed.), vol. 1039, *LNCS*, pp. 71-82, Springer-Verlag, 1996.

[24] FIPS 186-2, Digital Signature Standard (DSS), National Institute of Standards and Technology, Jan. 2000.

[25] ECRYPT, `http://www.ecrypt.eu.org`.

[26] J. Faugère, "A new effecient algorithm for computing Gröbner bases (F_4)," *Journal of Pure and Applied Algebra*, vol. 139, pp. 61-88, 1999, Available Online at `http://www.elsevier.com/locate/jpaa`.

[27] J. Faugère, "A new effecient algorithm for computing Gröbner bases without reduction to zero (F_5)," *International Symposium on Symbolic and Algebraic Computation– ISSAC 2002*, pp. 75-83, ACM Press, 2002.

[28] A. Fiat and A. Shamir, *How to Prove Yourself: Practical Solutions to Identification and Signature Problems*, Advances in Cryptology – Proceedings of CRYPTO 1986, LNCS 263, pp. 186-194, Springer-Verlag, 1987.

[29] H. Finney, "An RC4 cycle that can't happen," Post in `sci.crypt`, September 1994.

[30] S. Fluhrer, I. Mantin, A. Shamir, "Weaknesses in the Key Scheduling Algorithm of RC4," *SAC 2001* (S. Vaudenay, A. Youssef, eds.), vol. 2259 of *LNCS*, pp. 1-24, Springer-Verlag, 2001.

[31] S. Fluhrer, D. McGrew, "Statistical Analysis of the Alleged RC4 Keystream Generator," *Fast Software Encryption 2000* (B. Schneier, ed.), vol. 1978 of *LNCS*, pp. 19-30, Springer-Verlag, 2000.

[32] R. Floyd, R. Beigel, "The Language of Machines," W. H. Freeman, 1994.

[33] N. Ferguson, D. Whiting, B. Schneier, J. Kelsey, S. Lucks, T. Kohno, "Helix: Fast Encryption and Authentication in a Single Cryptographic Primitive," *Fast Software Encryption 2003* (T. Johansson, ed.), vol. 2887 of *LNCS*, pp. 330-346, Springer-Verlag, 2003.

[34] O. Goldreich, "Lecture Notes on Pseudorandomness–Part-I," Department of Computer Science, Weizmann Institute of Science, Rehovot, Israel, January 23, 2001.

[35] J. Golić, "Linear Statistical Weakness of Alleged RC4 Keystream Generator," *Eurocrypt '97* (W. Fumy, ed.), vol. 1233 of *LNCS*, pp. 226-238, Springer-Verlag, 1997.

[36] G. Gong, K. C. Gupta, M. Hell, Y. Nawaz, "Towards a General RC4-Like Keystream Generator," *First SKLOIS Conference, CISC 2005* (D. Feng, D. Lin, M. Yung, eds.), vol. 3822 of *LNCS*, pp. 162-174, Springer-Verlag, 2005.

[37] A. Grosul, D. Wallach, "A related key cryptanalysis of RC4," *Department of Computer Science, Rice University, Technical Report TR-00-358,* June 2000.

[38] S. Halevi, D. Coppersmith, C. S. Jutla, "Scream: A Software-Efficient Stream Cipher," *Fast Software Encryption 2002* (J. Daemen and V. Rijmen, eds.), vol. 2365 of *LNCS*, pp. 195-209, Springer-Verlag, 2002.

[39] M. E. Hellman, "A Cryptanalytic Time-Memory Trade-off," IEEE Transaction on Information Theory, vol. IT-26, No. 4, July, 1980.

[40] J. E. Hopcroft, R. Motwani, J. D. Ullman, *"Introduction to Automata Theory, Languages and Computation,"* Second Edition, Pearson Education, 2004.

[41] R. J. Jenkins Jr., "ISAAC," *Fast Software Encryption 1996* (D. Gollmann, ed.), vol. 1039 of *LNCS*, pp. 41-49, Springer-Verlag, 1996.

[42] A. Joux and F. Muller, "Chosen-ciphertext attacks against MOS- QUITO, *"Fast Software Encryption, FSE 2006,* (M. Robshaw, ed.) Lecture Notes in Computer Science. Springer-Verlag, 2006.

[43] D. Kahn, "The Codebreakers: the story of Secret Writings," MacMillan Publishing Co. Inc., 1967.

[44] A. Kipnis, A. Shamir, "Cryptanalysis of the HFE Public Key Cryptosystems by Relinearization," *Crypto 1999* (M. Wiener, ed.), vol. 1666 of *LNCS*, pp. 19-30, Springer-Verlag, 1999.

[45] A. Klimov, A. Shamir, "Cryptographic Applications of T-Functions," *Selected Areas in Cryptography 2003* (M. Matsui, R. J. Zuccherato, eds.), vol. 3006 of *LNCS*, pp. 248-261, Springer-Verlag, 2004.

[46] A. Klimov, A. Shamir, "New Cryptographic Primitives Based on Multiword T-Functions," *Fast Software Encryption 2004* (B. Roy, W. Meier, eds.), vol. 3017 of *LNCS*, pp. 1-15, Springer-Verlag, 2004.

[47] L. Knudsen, W. Meier, B. Preneel, V. Rijmen, S. Verdoolaege, "Analysis Methods for (Alleged) RC4," *Asiacrypt '98* (K. Ohta, D. Pei, eds.), vol. 1514 of *LNCS*, pp. 327-341, Springer-Verlag, 1998.

[48] D. E. Knuth, *"The Art of Computer Programming," vol. 2, Seminumerical Algorithms,* Addison-Wesley Publishing Company, 1981.

[49] X. Lai, J. L. Massey, "A Proposal for a New Block Encryption Standard," *EUROCRYPT 1990* (I. Damgård, ed.), vol. 473 of *LNCS*, pp. 389-404, Springer-Verlag, 1990.

[50] X. Lai, J. L. Massey, "Markov Ciphers and Differential Cryptoanalysis," *EUROCRYPT 1991* (D. W. Davies, ed.), vol. 547 of *LNCS*, pp. 17-38, Springer-Verlag, 1991.

[51] H. Lipmaa, S. Moriai, "Efficient Algorithms for Computing Differential Properties of Addition," *FSE 2001* (M. Matsui, ed.), vol. 2355 of *LNCS*, pp. 336-350, Springer-Verlag, 2002.

[52] H. Lipmaa, J. Wallén, P. Dumas, "On the Additive Differential Probability of Exclusive-Or," *Fast Software Encryption 2004* (B. Roy, W. Meier, eds.), vol. 3017 of *LNCS*, pp. 317-331, Springer-Verlag, 2004.

[53] I. Mantin, A. Shamir, "A Practical Attack on Broadcast RC4," *Fast Software Encryption 2001* (M. Matsui, ed.), vol. 2355 of *LNCS*, pp. 152-164, Springer-Verlag, 2001.

[54] M. Matsui, "Linear Cryptoanalysis Method for DES Cipher," *Eurocrypt 1993* (T. Helleseth, ed.), vol. 2355 of *LNCS*, pp. 386-397, Springer-Verlag, 1993.

[55] A. Maximov, "Two Linear Distinguishing Attacks on VMPC and RC4A and Weakness of RC4 Family of Stream Ciphers," *Fast Software Encryption 2005* (H. Gilbert and H. Handschuh, eds.), vol. 3557 of *LNCS*, pp. 342-358, Springer-Verlag, 2005.

[56] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone, "Handbook of Applied Cryptography,"CRC Press, Available Online at `http://www.cacr.math.uwaterloo.ca/hac/`, 1996.

[57] I. Mironov, "Not (So) Random Shuffle of RC4," *Crypto 2002* (M. Yung, ed.), vol. 2442 of *LNCS*, pp. 304-319, Springer-Verlag, 2002.

[58] S. Mister, S. Tavares, "Cryptanalysis of RC4-like Ciphers," *SAC '98* (S. Tavares, H. Meijer, eds.), vol. 1556 of *LNCS*, pp. 131-143, Springer-Verlag, 1999.

[59] F. Muller, "Differential Attacks against the Helix Stream Cipher," *Fast Software Encryption 2004* (B. Roy, W. Meier, eds.), vol. 3017 of *LNCS*, pp. 94-108, Springer-Verlag, 2004.

[60] Y. Nawaz, K. C. Gupta, and G. Gong, "A 32-bit RC4-like Keystream Generator," Cryptology ePrint Archive, 2005/175.

[61] NESSIE: New European Schemes for Signature, Integrity and Encryption, `http://www.cryptonessie.org`.

[62] B.C. Neuman and T. Ts'o, *Kerberos: An Authentication Service for Computer Networks*, IEEE Communications, 32(9), pp. 33-38, 1994.

[63] K. Nyberg, L. Knudsen, "Provable Security Against a Differential Attack," *Journal of Cryptology*, 8(1):27-37, 1991.

[64] S. Paul, B. Preneel, "Analysis of Non-fortuitous Predictive States of the RC4 Keystream Generator," *Indocrypt 2003* (T. Johansson, S. Maitra, eds.), vol. 2904 of *LNCS*, pp. 52-67, Springer-Verlag, 2003.

[65] S. Paul, B. Preneel, "A New Weakness in the RC4 Keystream Generator and an Approach to Improve the Security of the Cipher," *Fast Software Encryption 2004* (B. Roy, ed.), vol. 3017 of *LNCS*, pp. 245-259, Springer-Verlag, 2004.

[66] S. Paul, B. Preneel, "Solving Systems of Differential Equations of Addition (Extended Abstract)," *10th Australasian Conference on Information Security and Privacy, ACISP 2005* (Colin Boyd and Juan Gonzalez, eds.), vol. 3574 of *LNCS*, pp. 75-88, Springer-Verlag, 2005, Extended Version available online on IACR ePrint Archive as Report 2004/294 at `http://eprint.iacr.org/2004/294`, April 2005.

[67] S. Paul, B. Preneel, "Near Optimal Algorithms for Solving Differential Equations of Addition with Batch Queries," *Indocrypt 2005* (Subhamoy Maitra, C. E. Veni Madhavan and R. Venkatesan, eds.), vol. 3797 of *LNCS*, Springer-Verlag, pp. 90-103, 2005.

[68] S. Paul, B. Preneel, G. Sekar, "Distinguishing Attacks on the Stream Cipher Py," *Fast Software Encryption 2006* (M. Robshaw, ed.), vol. 4047 of *LNCS*, Springer-Verlag, pp. 405-421, 2006.

[69] S. Paul, B. Preneel, "On the (In)security of Stream Ciphers Based on Arrays and Modular Additions," *Asiacrypt 2006* (X. Lai, ed.), *LNCS*, Springer-Verlag, 2006 (to appear).

[70] B. Preneel *et al.*, "NESSIE Security Report," Version 2.0, IST-1999-12324, February 19, 2003, `http://www.cryptonessie.org`.

[71] B. Preneel, P. C. van Oorschot, "MDx-MAC and Building Fast MACs from Hash Functions," *Advances in Cryptology – Proceedings of CRYPTO 1995* (D. Coppersmith, ed.), LNCS 963, pp. 1-14, Springer-Verlag, 1995.

[72] M. Pudovkina, "Statistical Weaknesses in the Alleged RC4 keystream generator," *Cryptology ePrint Archive 2002-171,* IACR, 2002.

[73] M. Pudovkina, "A known plaintext attack on the ISAAC keystream generator," *Cryptology ePrint Archive: Report 2001/049,* IACR, 2001.

[74] R. L. Rivest, "The MD4 Message Digest Algorithm," *Advances in Cryptology – Proceedings of CRYPTO 1990* (A. Menezes, S. A. Vanstone, eds.), pp. 303-311, Springer-Verlag, 1991.

[75] R. L. Rivest, "The MD5 Message Digest Algorithm," Request for Comments (RFC 1320), Internet Activities Board, Internet Privacy Task Force, 1992.

[76] R. L. Rivest, "The RC4 Encryption Algorithm," RSA Data Security, Inc., March 12, 1992.

[77] R. L. Rivest, M. Robshaw, R. Sidney, Y. L. Yin, "The RC6 Block Cipher," Available Online at `http://theory.lcs.mit.edu/ rivest/rc6.ps`, June 1998.

[78] R. L. Rivest, A. Shamir, L. M. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," Commun. ACM., vol. 21, no. 2, pp. 120-126, 1978.

[79] R. L. Rivest, M. Robshaw, Y. L. Yin, "RC6 as the AES," *AES Candidate Conference 2000*, National Institute of Standards and Technology, pp. 337-342, 2000.

[80] A. Roos, "Class of weak keys in the RC4 stream cipher," Post in `sci.crypt`, September 1995.

[81] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C.Hall, N. Ferguson, "The Twofish Encryption Algorithm: A 128-Bit Block Cipher," John Wiley & Sons, April 1999, ISBN: 0471353817.

[82] A. Shamir, "Stream Ciphers: Dead or Alive?," *Advances in Cryptology - ASIACRYPT 2004* (P. Lee, ed.), vol. 3329, Lecture Notes in Computer Science, pp. 78, Springer-Verlag, 2004. .

[83] U.S. Department of Commerce. *FIPS 180*: Secure Hash Standard, SHA-0, Federal Information Processing Standards Publication, N.I.S.T., May 1993.

[84] U.S. Department of Commerce. *FIPS 180-1*: Secure Hash Standard, SHA-1, Federal Information Processing Standards Publication, N.I.S.T., April 1995.

[85] U.S. Department of Commerce. *FIPS 180-2*: Secure Hash Standard, SHA-2, Federal Information Processing Standards Publication, N.I.S.T., August 2002.

[86] C. E. Shannon, "Communication Theory of Secrecy Systems," *Bell System Technical Journal*, vol. 28-4, pp. 656-715, 1949. Available online at `http://www.cs.ucla.edu/~jkong/research/security/shannon1949.pdf`.

[87] O. Staffelbach, W. Meier, "Cryptographic Significance of the Carry for Ciphers Based on Integer Addition," *Crypto '90* (A. Menezes, S. A. Vanstone, eds.), vol. 537 of *LNCS*, pp. 601-614, Springer-Verlag, 1991.

[88] D. R. Stinson, "Cryptography: Theory and Practice," CRC Press, Third Edition, November 2005.

[89] A. Stubblefield, J. Ioannidis, A. Rubin, "Using the Fluhrer, Mantin and Shamir attack to break WEP," *Proceedings of the 2002 Network and Distributed Systems Security Symposium,* pp. 17-22, 2002.

[90] Y. Tsunoo, T. Saito, H. Kubo, M. Shigeri, T. Suzaki, and T. Kawabata, "The Most Efficient Distinguishing Attack on VMPC and RC4A," eStream project, Available Online at `http://www.ecrypt.eu.org/stream/papersdir/037.pdf`, 2005.

[91] H. Yoshida, A. Biryukov, C. D. Cannière, J. Lano, B. Preneel, "Non-randomness of the Full 4 and 5-Pass HAVAL," *SCN 2004* (C. Blundo and S. Cimato, eds.), vol. 3352 of *LNCS*, pp. 324-336, Springer-Verlag, 2004.

[92] D. Wagner, "The Boomerang Attack," *Fast Software Encryption 1999*, (L. R. Knudsen, ed.), vol. 1636 of *LNCS*, pp. 156-170, Springer-Verlag, 1999.

[93] J. Wallén, "Linear Approximations of Addition Modulo 2^n," *Fast Software Encryption 2003* (T. Johansson, ed.), vol. 2887 of *LNCS*, pp. 261-273, Springer-Verlag, 2003.

[94] C. Wolf, "Multivariate Quadratic Polynomials in Public Key Cryptography," Ph.D. Thesis, Katholieke Universiteit Leuven, Belgium, 2005.

[95] H. Wu, "A New Stream Cipher HC-256," *Fast Software Encryption 2004* (B. Roy, ed.), vol. 3017 of *LNCS*, pp. 226-244, Springer-Verlag, 2004.

[96] H. Wu, "Cryptanalysis of a 32-bit RC4-like Stream Cipher," Cryptology ePrint Archive, 2005/219.

[97] H. Wu, B. Preneel, "Key Recovery Attack on Py and Pypy with Chosen IVs," eSTREAM, ECRYPT Stream Cipher Project, Report 2006/052, 2006.

[98] B. Zoltak, "VMPC One-Way Function and Stream Cipher," *Fast Software Encryption 2004* (B. Roy, ed.), vol. 3017 of *LNCS*, pp. 210-225, Springer-Verlag, 2004.

Appendix A

Helix

A.1 Proofs of Lemma 2.6 and Lemma 2.7

Claim A.1 *For all* $(0,\, \beta,\, \tilde{\gamma}) \in \tilde{D}$, $\tilde{\gamma}_{(i)} = 0\ \forall\, i \in [0,\, t]$.

Proof. If the position of the least significant '1' of x is t then $c_{(i)} = \tilde{c}_{(i)} = 0$ $\forall\, i \in [0,\, t]$ and $\forall\, \beta \in \mathbb{Z}_2^n$ (see (2.4)). Recall $\tilde{\gamma}_{(i)} = c_{(i)} \oplus \tilde{c}_{(i)}$. This proves the lemma. □

Claim A.2 *For each* $i \in [t+1,\, n-1]$, *there exists* $(0,\, \beta,\, \tilde{\gamma}) \in \tilde{D}$ *with* $\tilde{\gamma}_{(i)} = 1$.

Proof. We prove the lemma by induction on i. The statement is true if $i = t+1$. Suppose, the statement is true if $i = k$ for some $k \in [t+1,\, n-2]$, that is, there exists $(0,\, a,\, b) \in \tilde{D}$ with $b_{(k)} = 1$ (induction hypothesis). We construct three n-bit integers from a,

1. $a' = (a_{(n-1)},\, a_{(n-2)},\, \cdots,\, a_{(k+1)},\, 0,\, a_{(k-1)},\, \cdots,\, a_{(0)})$
2. $a'' = (a_{(n-1)},\, a_{(n-2)},\, \cdots,\, a_{(k+1)},\, 1,\, a_{(k-1)},\, \cdots,\, a_{(0)})$
3. $a''' = (a_{(n-1)},\, a_{(n-2)},\, \cdots,\, a_{(k+1)},\, 1,\, 0,\, 0,\, \cdots,\, 0)$.

Now we select three elements $(0,\, a',\, b')$, $(0,\, a'',\, b'')$, $(0,\, a''',\, b''') \in \tilde{D}$ (such elements exist since, for all $p \in \mathbb{Z}_2^n$, there exists $(0,\, p,\, q) \in \tilde{D}$ for some $q \in \mathbb{Z}_2^n$). Note that $b'_{(k)} = b''_{(k)} = b_{(k)} = 1$ and $b'''_{(k)} = 0$. From Table 2.1, at least one of $b'_{(k+1)}$, $b''_{(k+1)}$ and $b'''_{(k+1)}$ is 1. This proves the lemma. □

A.2 Proof of Lemma 2.11

Claim A.3 *Let $n-4 \geq t \geq 0$. For any algorithm $\mathcal{A}$ there exists a seed $(x, y) \in V_{n,\,t}$ such that the adaptively selected sequence of two queries by $\mathcal{A}$ for that particular seed produces oracle outputs $\tilde{\gamma}$'s with $\tilde{\gamma}_{(i+1)} = 0, \forall\, i \in [t+1,\, n-2]$.*

Proof. Let the first two queries and the corresponding oracle outputs be $(0, \beta)$, $(0, \beta')$, $\tilde{\gamma}$ and $\tilde{\gamma}'$. Depending *only* on the tth bit of β and β', the oracle returns outputs (i.e., $\tilde{\gamma}$ and $\tilde{\gamma}'$) according to the following rules.

1. If $\beta_{(t)} = 0$ then the oracle returns $\tilde{\gamma} = (0, 0, \cdots, 0)_n$.
2. If $\beta_{(t)} = 1$ then $\tilde{\gamma}_{(t+1)} = 1$ and all other bits of $\tilde{\gamma}$ are zero.
3. If $\beta'_{(t)} = 0$ then the oracle returns $\tilde{\gamma}' = (0, 0, \cdots, 0)_n$.
4. If $\beta'_{(t)} = 1$ then $\tilde{\gamma}'_{(t+1)} = 1$ and all other bits of $\tilde{\gamma}'$ are zero.

Under any of the above input-output combinations there exists a seed and $\tilde{\gamma}_{(i+1)} = 0, \forall\, i \in [t+1,\, n-2]$ for all outputs $\tilde{\gamma}$. This proves the lemma. $\square$

A.3 Construction of M from the L_i's

Let G_i be known $\forall\, i \in [0,\, n-2]$ $(n > 1)$ for a nonempty *useful set* $\tilde{A}$. Let $L_i = \{(x_{(i)}, y_{(i)}, c_{(i)}) \,|\, G_i \Rightarrow (x_{(i)}, y_{(i)}, c_{(i)})\} \forall i \in [0,\, n-2]$. Let a set M be constructed from the L_i's in the following way,

$$\begin{aligned} M \;=\; & \{((x_{(n-1)}, x_{(n-2)}, \cdots, x_{(0)}), (y_{(n-1)}, y_{(n-2)}, \cdots, y_{(0)})) \\ & |(x_{(n-1)}, y_{(n-1)}) \in \mathbb{Z}_2^2, (x_{(i)}, y_{(i)}, c_{(i)}) \in L_i, i \in [0,\, n-2], c_{(0)} = 0\,, \\ & c_{(i+1)} = x_{(i)}y_{(i)} \oplus x_{(i)}c_{(i)} \oplus y_{(i)}c_{(i)}\}\,. \end{aligned}$$

We present an algorithm to construct M from the L_i's with memory $\mathcal{O}(n \cdot S)$ and time $\mathcal{O}(S)$ where $S = |\tilde{A}\text{-consistent}|$.

First we set

$$M_1 = \{((c_{(1)}), (x_{(0)}), (y_{(0)}))|(x_{(0)}, y_{(0)}, 0) \in L_0, c_{(1)} = x_{(0)}y_{(0)}\}\,.$$

Now we construct a set M_k $\forall\, k \in [2,\, n-1]$ using the following recursion.

$$\begin{aligned} M_k \;=\; & \{((c_{(k)}), (x_{(k-1)}, \cdots, x_{(0)}), (y_{(k-1)}, \cdots, y_{(0)}))| \\ & (x_{(k-1)}, y_{(k-1)}, c_{(k-1)}) \in L_{k-1}\,, \\ & ((c_{(k-1)}), (x_{(k-2)}, \cdots, x_{(0)}), (y_{(k-2)}, \cdots, y_{(0)})) \in M_{k-1}, \\ & c_{(k)} = x_{(k-1)}y_{(k-1)} \oplus x_{(k-1)}c_{(k-1)} \oplus y_{(k-1)}c_{(k-1)}\}\,. \end{aligned}$$

Now, we construct

$$\begin{aligned} M_n \;&=\; \{((x_{(n-1)}, \cdots, x_{(0)}), (y_{(n-1)}, \cdots, y_{(0)}))|(x_{(n-1)}, y_{(n-1)}) \in \mathbb{Z}_2^2\,, \\ &\quad ((c_{(n-1)}), (x_{(n-2)}, \cdots, x_{(0)}), (y_{(n-2)}, \cdots, y_{(0)})) \in M_{n-1}\}\,. \end{aligned}$$

Using Proposition 2.3 and Theorem 2.2 it is easy to show that $M = M_n$. Note that the size of each L_i is $\mathcal{O}(1)$ since the size of the Table 2.1 is $\mathcal{O}(1)$. Also note that $|M_n| = S$ and therefore the asymptotic memory requirement to construct M_n recursively following the above algorithm is $\mathcal{O}(n \cdot S)$ since $k = \mathcal{O}(n)$ and M_{k+1} can be constructed from M_k only. It is trivial to show that the time to construct M_n (i.e., M) from the L_i's is $\mathcal{O}(S)$. Thus, the set M can be constructed from the L_i's with memory $\mathcal{O}(n \cdot S)$ and time $\mathcal{O}(S)$.

Appendix B

RC4

B.1 Criteria for i to Reach an Index to Produce an Output

The fact that the i pointer can move from the index x to the index y implies that the value of j is always available in each of the intermediate $(y - x + 1)$ rounds. The S-Box region between the indices x and y is all the $(y - x + 1)$ indices from x in the direction of the movement of i.

Proposition B.1 *If, at a particular round r (when $i = i_r$), j_r and some elements of the S-Box are known, then the fact that i can reach the index $i_r + k$ (where $0 < k \leq N$) from the round r depends only on j_r and the known S-Box elements between the indices $i_r + 1$ and $i_r + k$ at round r.*

Proposition B.2 *Let the number of known elements of the S-Box at the rth round between the indices $i_r + 1$ and $i_r + k$ (where $0 < k \leq N$) be m and at the rth round the tth element to the right of i be indexed by p_t $(0 < t \leq m)$. Then, starting from the rth round, the pointer i must reach at least p_t to predict the tth output.*

Proposition B.3 *If the number of rounds, at which $S[j]$ is known during the passage of i from $i = i_r$ to $i = i_r + k$ (where $0 < k \leq N$), is m, then the number of known elements, between the indices $i_r + 1$ and i_{r+k} at round $r + k$, is also m.*

B.2 Evaluation of the Maximum Value of d_2

Theorem B.4 *For any* non-fortuitous state *of length 3, $p_3 - p_2 < 3$ where the tth element is indexed by p_t.*

PROOF. Let us assume $p_3 - p_2 = 3$. Now we try to generate 3 *non-consecutive* outputs, in a similar manner as in Theorem **??**. The execution of the first three rounds are shown in Figure B.1. At the 2nd and the 3rd rounds j must point to $S_2[g_1]$ and $S_3[g_2]$ respectively, in order for j to be available at the third and the fourth rounds. But such conditions lead to Finney's forbidden state at round three. So our assumption is wrong. Hence, $p_3 - p_2 \neq 3$. It is easy to see that the same situation arises for $p_3 - p_2 > 3$. Therefore, $p_3 - p_2 < 3$. We know that $d_2 = p_3 - p_2 - 1$. Hence, $d_2^{max} = 1$. □

One can see that if we relax the condition of the 1st round producing output always, then the maximum *inter-element gap* between the first two elements of the S-Box is also 1. This basic fact will be used in the determination of d_t^{max} when $t > 2$.

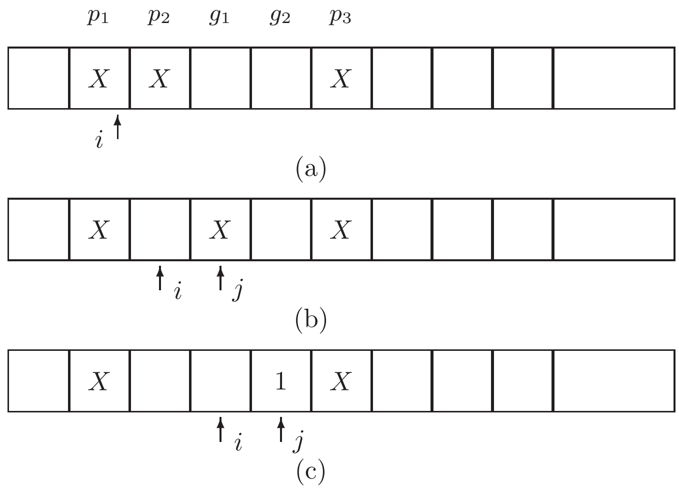

Figure B.1: *A non-fortuitous state* of length 3 with $d_2 = 2$: (a) Round 1: after production of the 1st output, X indicates known value; (b) Round 2: no output; (c) Round 3: we reach Finney's forbidden state as $j_3 = i_3 + 1$ and $S_3[j_3] = 1$

Appendix C

Py

C.1 Uniformity of Bits If L Does Not Occur

We first write the general formula to calculate $Z = O_{1,1} \oplus O_{2,3}$.

$$\begin{aligned} O_{1,1} &= (ROT32(s, 25) \oplus G) + H\,, && \text{(C.1)} \\ O_{2,3} &= (ROT32(ROT32(s + I - J\,, r) + K - L\,, l) \oplus M) + N\,, && \text{(C.2)} \end{aligned}$$

where
$s = s_1$, $G = Y_1[256]$, $H = Y_1[P_1[26]]$, $I = Y_2[P_2[72]]$, $J = Y_2[P_2[239]]$, $r = P_2[116] + 18 \bmod 32$, $K = Y_3[P_3[72]]$, $L = Y_3[P_3[239]]$, $l = P_3[116] + 18 \bmod 32$, $M = Y_3[-1]$, $N = Y_3[P_3[208]]$.

Below we isolate 18 cases, divided into 4 groups, where the relation between internal and external states is not trivial. The notation $A \leftrightarrow B$ signifies that the A and B are identical elements in two different rounds of the S-box Y (i.e., $A = B$ but their indices may be changed in different rounds). The symbol '/' is used to mean 'or'. Note that the following relations in each group are satisfied if they do not violate the condition of uniqueness of permutation elements of S-box P.

1. $I \leftrightarrow N/M$, $J \leftrightarrow M/N$, $K \leftrightarrow G/H$, $L \leftrightarrow H/G$ (a total of 4 cases). See Fig.C.1.

2. $I \leftrightarrow K/L$, $J \leftrightarrow L/K$, $M \leftrightarrow H$, $N \leftrightarrow G$ (a total of 2 cases). See Fig.C.2.

3. $I \leftrightarrow N/M$, $J \leftrightarrow K/L$. The G is identical to one of the remaining two elements (so is the H) (a total of 6 cases). See Fig.C.1.

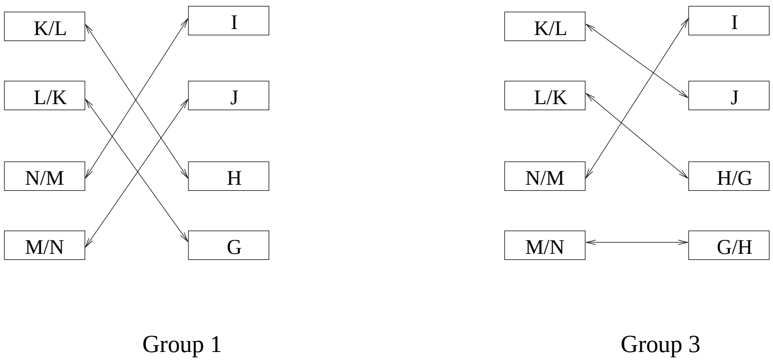

Figure C.1: Representations of Group 1 and Group 3

4. Similar to the above, $J \leftrightarrow M/N$, $I \leftrightarrow L/K$. The G is identical to one of the remaining two elements (so is the H) (a total of 6 cases).

Fact C.1 *After the key/IV setup, if the permutation P falls outside all of the 18 cases described above then $O_{1,1}$ and $O_{2,3}$ are independent and uniformly distributed over* $[2^{32}-1, 0]$.

Now we denote $O_{1,1(0)} \oplus O_{2,3(0)}$ by R_0.

Theorem C.2 *After the key/IV setup, let the S-box P be one of the 16 cases described in Groups 1,3 and 4. Then*

$$P[R_0 = 0 \,|\, P] = \frac{1}{2}.$$

Proof. Now we prove the theorem by considering Groups 1, 3 and 4 separately.
Group 1 (see Fig. C.1). For this group, R_0 can be written in the following form,

$$\begin{aligned} R_0 \quad = \quad & s_{(7)} \oplus s_{(w)} \oplus (G_{(0)} \oplus H_{(0)} \oplus K_{(u)} \oplus L_{(u)}) \\ & \oplus (I_{(w)} \oplus J_{(w)} \oplus M_{(0)} \oplus N_{(0)}) \oplus C. \end{aligned}$$

Note that the C is a nonlinear function of several bits of s, M, N, H, G. Now we take three possible subcases.

1. If $u \neq 0$ then R_0 is uniformly distributed since C is independent of $K_{(u)}$ and $L_{(u)}$.

2. If $u = 0$, $w \neq 0$ then R_0 is uniformly distributed since C is independent of $I_{(w)}$ and $J_{(w)}$.

3. If $u = 0$, $w = 0$ then $R_0 = s_{(7)} \oplus s_{(0)}$. Therefore, R_0 is uniformly distributed.

Group 3 (see Fig. C.1). R_0 can be written in the following form,

$$\begin{aligned} R_0 &= s_{(7)} \oplus s_{(w)} \oplus (G_{(0)} \oplus N_{(0)}) \oplus (K_{(u)} \oplus J_{(w)}) \\ &\quad \oplus (H_{(0)} \oplus L_{(u)}) \oplus (M_{(0)} \oplus I_{(w)}) \oplus C. \end{aligned}$$

Of the 6 cases in Group 3, we are considering *only* the following case where $I \leftrightarrow M$, $J \leftrightarrow K$, $G \leftrightarrow N$ and $H \leftrightarrow L$. In a similar way as above we divide this case into three subcases.

1. If $u \neq 0$ then C is independent of $L_{(u)}$ and thus R_0 is uniformly distributed.

2. If $u = 0$, $w \neq 0$ then R_0 is uniformly distributed since C is independent of $J_{(w)}$.

3. If $u = 0$, $w = 0$ then $R_0 = s_{(7)} \oplus s_{(0)}$. Therefore, R_0 is uniformly distributed.

All the other 5 cases of this group can be proved in a similar fashion.
Group 4. Proof for this group is similar to that for Group 3. □

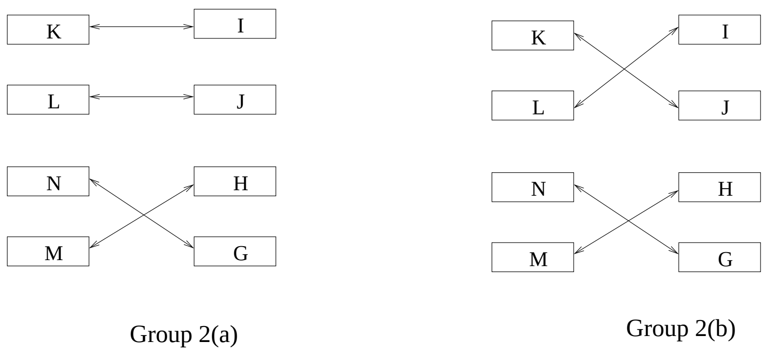

Figure C.2: Group 2(a): $I \leftrightarrow K$, $J \leftrightarrow L$, $M \leftrightarrow H$, $N \leftrightarrow G$; Group 2(b): $I \leftrightarrow L$, $J \leftrightarrow K$, $M \leftrightarrow H$, $N \leftrightarrow G$

Discussion. From Fact C.1 and Theorem C.2, it is clear that, if P does not fall within Group 2 then $P[R_0 = 0|P] = \frac{1}{2}$. The probability of the occurrence of Group 2 is approximately 2^{-31}. Therefore, for a fraction of $(1-2^{-31})$ of all cases,

R_0 is uniformly distributed. Sect. 5.4 shows that, for the event L occurring with probability $2^{-41.9}$, $P[R_0 = 0 \mid P = L] = 1$.

Therefore, we are able to prove that, for a fraction of $(1-2^{-31.001})$ of all cases, there exists a bias in R_0 toward zero. It is, however, nontrivial to determine the distribution of R_0 for the remaining fraction of $2^{-31.001}$ of the cases, because of vigorous mixing of bits in a nonlinear way. Our experiments suggest that it is very unlikely that the positive bias generated in the large fraction of $(1-2^{-31.01})$ can be compensated by a very minuscule fraction of $2^{-31.001}$. According to a small number of experiments that we carried out, a slight bias toward zero was detected for that remaining fraction of $2^{-31.001}$ also. However, we ignored that bias and assumed R_0 to be uniformly distributed for those cases in building the distinguishers described in the chapter.

In addition to the event L, for which R_i is biased toward zero $\forall i \in [1, 31]$ (see Sect. 5.6), we also identify another event L', for which R_{25} is again biased toward zero (all other R_i's are uniformly distributed individually). The event L' occurs when $P_2[116] \equiv -18 (\bmod 32)$, $P_3[116] \equiv 7 (\bmod 32)$, $P_2[72] = P_3[72] + 1$, $P_2[239] = P_3[239]+1$, $P_1[26] = 1$, $P_3[208] = 254$ (see Group 2(a) of Fig. C.2). Using similar arguments as above, it can be shown that R_i is uniformly distributed over [0, 1] for the rest of the cases.

List of Publications

Lecture Notes in Computer Science

1. Souradyuti Paul, Bart Preneel, "On the (In)security of Stream Ciphers Based on Arrays and Modular Additions," *Asiacrypt 2006* (X. Lai, ed.), *LNCS*, Springer-Verlag, 2006 (to appear).

2. Souradyuti Paul, Bart Preneel, Gautham Sekar, "Distinguishing Attacks on the Stream Cipher Py," *Fast Software Encryption 2006* (M. Robshaw, ed.), vol. 4047 of *LNCS*, Springer-Verlag, pp. 405-421, 2006.

3. Souradyuti Paul, Bart Preneel, "Near Optimal Algorithms for Solving Differential Equations of Addition with Batch Queries," *Indocrypt 2005* (Subhamoy Maitra, C. E. Veni Madhavan and Ramarathnam Venkatesan, eds.), vol. 3797 of *LNCS*, Springer-Verlag, pp. 90-103, 2005.

4. Souradyuti Paul, Bart Preneel, "Solving Systems of Differential Equations of Addition (Extended Abstract)," *10th Australasian Conference on Information Security and Privacy, ACISP 2005* (Colin Boyd and Juan Gonzalez, eds.), vol. 3574 of *LNCS*, pp. 75-88, Springer-Verlag, 2005, Extended Version available online on IACR ePrint Archive as Report 2004/294 at `http://eprint.iacr.org/2004/294`, April 2005.

5. Souradyuti Paul, Bart Preneel, "A New Weakness in the RC4 Keystream Generator and an Approach to Improve the Security of the Cipher," *Fast Software Encryption 2004* (B. Roy, ed.), vol. 3017 of *LNCS*, pp. 245–259, Springer-Verlag, 2004.

6. Souradyuti Paul, Bart Preneel, "Analysis of Non-fortuitous Predictive States of the RC4 Keystream Generator," *Indocrypt 2003* (T. Johansson, S. Maitra, eds.), vol. 2904 of *LNCS*, pp. 52-67, Springer-Verlag, 2003.

Journals (national level)

1. Souradyuti Paul, “Cryptology: A Mathematician’s Quest for Making and Breaking the Code,” *Point: Journal of Department of Mathematics, Sree Chaitanya College, Habra, India* (Sanatan Paul, Saroj Kumar Chattopadhyay, Utpal Dasgupta, Uttam Das, eds.), No. 1, 2005.

Short CV

Souradyuti Paul was born on January 13, 1976 in Kolkata (formerly known as Calcutta), India, to Dr. Sanatan Paul (a former professor of mathematics) and Mrs. Ratna Paul as the eldest son of their two children. He received his Bachelor of Engineering degree (B.E.) in Mechanical Engineering from Jadavpur University, Kolkata, India, in 1998. Thereafter he was caught up in the whirlpool of burgeoning software industry in India for one year, only to eventually realize that it was not his cuppa. So he again put on the mantle of a student and secured an admission for a Master's degree in computer science to the Indian Statistical Institute, Kolkata, where he was awarded a Master of Technology degree (M.Tech) in Computer Science in July 2001. His master's thesis was on the analysis of non-linearities of a special class of Boolean functions under the supervision of Dr. Subhamoy Maitra. After that, he worked as a Junior Research Fellow at the CVPRU (Computer Vision and Pattern Recognition Unit) at the Indian Statistical Institute for 6 months. In January 2002, he joined the research group COSIC (Computer Security and Industrial Cryptography) at the Department of Electrical Engineering (ESAT) of the K. U. Leuven, Belgium, as a pre-doctoral student. Since October 2003, he has been working in the same research group as a doctoral student.