A STUDY ON TIME/MEMORY TRADE-OFF CRYPTANALYSIS

Thesis submitted to Indian Statistical Institute

by

Sourav Mukhopadhyay
Applied Statistics Unit
Indian Statistical Institute
March 2006

A Study on Time/Memory Trade-Off Cryptanalysis

Thesis submitted to Indian Statistical Institute in partial fulfillment
of the requirements for the award of the degree of
Doctor of Philosophy

by

Sourav Mukhopadhyay
Applied Statistics Unit
Indian Statistical Institute
203, B. T. Road, Calcutta 700 108, INDIA

under the supervision of

Prof. Palash Sarkar
Applied Statistics Unit
Indian Statistical Institute
203, B. T. Road, Calcutta 700 108, INDIA
e-mail : palash@isical.ac.in

To my Mother

Preface

At the beginning of 2003, I was searching for a suitable area to do my PhD. My respected guide Prof. Palash Sarkar assigned me to do a survey work on block cipher cryptanalysis. I found myself very interested in this area. I am very lucky that I joined into a golden group – Cryptology Research Group of India formed by our beloved teacher Prof. Bimal Roy and had the opportunity to work with Prof. Palash Sarkar, Dr. Subhomoy Maitra, Prof. Rana Barua and many others. I have been inspired by many persons during my PhD work.

I would like to begin by thanking my advisor Prof. Palash Sarkar for his unlimited involvement to proceed with my research work. He always helped and tried to motivate me in my research. Without his solid and innovative effort, I do not think I would have reached at my current position. I also remember him as a good teacher for his interactive classes during my M.Tech course from 1999-2001.

The name of the next person that comes to my mind is Prof. Bimal Roy. He is my teacher in both of my M.Stat and M.Tech courses. I found him the most student-friendly teacher in my entire student life. He introduced me to the area of Cryptography and Coding Theory. He always provided his helping hand in several problems (academic and also non-academic) during the last couple of years. I feel proud myself that I got an opportunity to do a joint paper with him.

I convey my thank to Prof. Rana Barua, Dr. Subhomoy Maitra and Dr. Pinakpani Pal for their inspiration and all kinds of supports to achieve my goal.

During my long journey at ISI, I came in contact with many scientists like Prof. Sankar Kumar Pal, Prof. Rajeev Karandhikar, Prof. Arijit Choudhary, Dr. Mandar Mitra, Prof. C. A. Murty and many more. I am grateful to them for their teaching and guidance.

I must thank all anonymous referees of my published papers, whose comments have always added a new dimension to my works. I would like to convey my hearty thanks to Prof. Alex Biryukov for his valuable comments and suggestion during my preparation of the final draft and presentation slide for SAC 2005 to present our joint paper. I am also thankful to Mr. Michael Wiener for providing me a valuable research paper. I thank Prof. Harald Niederreiter, Prof. Willi Meier, Prof. Vincent Rijmen and Prof. Amr Youssef for carefully

reading an initial draft of Chapter 5 in this dissertation.

I would like to thank the anonymous reviewer of this thesis for his valuable comments.

I have spent my graduation period at Ramakrishna Mission Vidyamandira, Belur Math, where I got a fantastic spiritual environment to explore myself in every positive area of life provided by the Maharajs (Monks). My hostel superintendent Swapan Maharaj showed me the right direction for higher studies. I am also grateful to Biswarup Maharaj for his spiritual inspiration. There I got some real knowledgeable and good teachers like Sachin Bakshi, Late Mohanlal Singharoy, and others.

I am thankful to the faculty and staff of the Applied Statistics Unit for all the help during my PhD study in ISI. I would also like to express my special thanks to all members of Cryptology Research Group of ISI and my room mates: Bappa, Jayanta, Anupam, Debasis and Tarun.

Last, but not the least I must express my thanks to my family. My mother has always been a great source of strength. My father, brothers, sister and our family friend Radhabinod Pal are always with me in my journey with their well wishes. Finally, I must thank Ratna for her inspiration to complete this thesis.

List of Symbols

The following is a list of important symbols used throughout the thesis. In this list, we do not provide the symbols used in Chapter 7 to describe the hardware architecture. Instead the corresponding list is given at the start of Chapter 7 itself.

$E()$	:	encryption function
f	:	one way function, $f : \{0,1\}^s \rightarrow \{0,1\}^s$
y	:	s-bit string given to the attacker
N	:	number of all possible keys $= 2^s$
D	:	number of targets available to the attacker $= 2^d$
r	:	number of tables
m	:	number of rows
t	:	number of columns
T	:	online time
T_t	:	number of memory accesses
P	:	pre-computation time
M	:	memory
PS	:	success probability

Contents

Chapter 1

Introduction

Cryptography is the science or art of secret writing. The fundamental objective of cryptography is to enable two people to communicate over an insecure channel (a telephone line or a computer network for example) in such a way that an opponent can not understand what is being communicated. This privacy is achieved either by using *secret key cryptography* (also called symmetric key cryptography), or by adopting *public key cryptography* (also called asymmetric key cryptography). We concentrate on discussing only secret key cryptography for this dissertation.

In secret key cryptography, a *plaintext* is transformed to a *ciphertext* by means of an *encryption function* and a *secret key*. The ciphertext is communicated and the receiver who has the secret key, recovers the plaintext by using a *decryption function*. Study of cryptography concentrates on designing *secure* encryption and decryption function. The basic mathematical tools used for constructing such functions are algebra, number theory, combinatorics etc. Block cipher and stream cipher are two basic primitives of secret key cryptography.

Cryptanalysis (popularly known as code breaking) is the other side of the coin. It is assumed that ciphertexts and the model for the encryption are known to the attacker. In addition some plaintexts may be available. There are different ways of modeling the adversary's behaviour, some of which allow the adversary to be adaptive.

Cryptographic algorithms usually require the use of a secret key to ensure confidentiality of transmitted messages. The basic goal of a cryptanalytic attack is to recover the secret key from publicly available information. Very often a successful attack exploits weakness in the design of the specific algorithm being considered. Two common attacks on secret key algorithms are linear cryptanalysis [64, 65] and differential cryptanalysis [21]. There are several variants of differential attacks, namely truncated and higher order differential [61] attack, impossible differential attack [19], boomerang attack [96]. The literature contains a

wide variety of attacks on secret key cryptosystems, some of which are related key attack [18], algebraic attack [36, 37], slide attack [27], correlation attack [90].

A generic approach to cryptanalysis views the encryption function as a black box, i.e., it does not utilize information about how the function is constructed. The simplest generic attack is to try every possible key until the correct one is found. This is called an exhaustive search attack. The importance of such an approach arises from the fact that if a cryptographic algorithm is not secure against exhaustive search, then it cannot be considered secure at all. The effectiveness of exhaustive search depends on the size of the key space and the following factors:

(a) implementation in software or special purpose hardware,

(b) the number of parallel processors available,

(c) the speed at which each key can be processed,

(d) the cost of each processor and the overall cost of implementing the attack.

In 1998, the Electronics Frontier Foundation [3] built a machine called *DES Cracker* for breaking DES at a cost of US $200,000 and which solved a DES problem in 3 and 1/2 days.

The main disadvantage of using exhaustive search is that it has to be repeated separately for each target. To address this problem, Hellman [56] introduced *time/memory trade-off* (TMTO) attack that enables one to perform an exhaustive search *once* in an offline precomputation phase. The actual attack, i.e., *finding the key corresponding to a target* is done in an online phase with table lookups and is significantly faster than exhaustive search. Also, one can repeat the attack on different targets without going through the precomputation each time. A TMTO attack is a generic attack which can be carried out against any one-way function f. The online target consists of an image y and the goal of the attacker is to find a x, such that $f(x) = y$, x being the secret key (preimage) from a key space of size N corresponding to the target y. In the multiple data version of TMTO, the adversary is given search targets (also called data) $y_1, y_2, \ldots, y_D$ and has to recover at least one x_i, such that, $f(x_i) = y_i$.

Since the publication of Hellman's result, there has been a spurt in research on TMTO. Hellman's method can recover a key in time T using memory M with the trade-off curve $TM^2 = N^2$ for $1 \leq T \leq N$, N being the number of all possible keys. Rivest [39] introduced the idea of distinguished points (DP) to reduce the number of table lookups in Hellman's method. Oechslin [76] proposed the rainbow method to reduce runtime cost to one-half of the Hellman method. Barkan et al. [14] pointed out that the cost of storage in the rainbow method is substantially higher than for the Hellman + DP. Oechslin also described in [76] the implementation of *rainbowcrack* which is a software implementation of the rainbow method.

TMTO was applied to stream ciphers independently by Babbage [12] and Golić [46]. This attack is jointly known as the BG attack. Later, Biryukov and Shamir [25] combined the Hellman attack and the BG attack to obtain TMTO with multiple data and trade-off curve $TM^2D^2 = N^2$, where D is the number of available targets to the attacker. There is an elegant application of TMTO in [26], which uses a special type of sampling technique called the BSW sampling.

Hellman's method assumes certain randomness property of f and it fails for some construction of f. Fiat-Naor [45] proposed a TMTO construction that works for any function f. Unfortunately, the trade-offs obtained in [45] are worse than the Hellman trade-offs.

In 1988, Amirazizi and Hellman [10] proposed *time/memory/processor trade-off* where several processors execute in parallel, sharing a large memory through a *switching/sorting* network. They assumed that the cost of the wires is less than $n \log n$ and left this as an open problem for further study. Wiener [98] investigated the problem and proved that if an algorithm has a very high memory access rate then the wiring cost is the dominating cost for any *switching/sorting* network and showed this cost to be $\Theta(n^{\frac{3}{2}})$ to connect n processors with n memory blocks. On the other hand, it is shown in [98] that the wiring cost is negligible if the memory access rate is low.

Quisquater and Standaert [81] provided a generic architecture based on their two previous works [79, 82]. They suggested a pipelined architecture for implementing a multi-round function f. This is built on Wiener's design [99] of implementing DES in his exhaustive search attack on DES. Mentens et al. [69] proposed a hardware architecture for key search based on the rainbow method. Bernstein [16] described the precomputation phase of the rainbow method as well as the Hellman+DP method as parallel brute force search algorithms.

The goal of this thesis is to study the design and analysis of TMTO attacks. We study in great detail the exhaustive search attack and TMTO attacks and present a survey of these generic attacks in Chapters 2 and 3. Chapter 2 talks about exhaustive search attack and Chapter 3 covers TMTO methodology. The effect of multiple data in the Hellman TMTO was first studied by Biryukov and Shamir (BS) [25]. We continue the analysis of the general multiple data TMTO started by BS in Chapter 4 and design new single table trade-offs. In Chapter 5, we introduce the use of LFSR as function generator and parallel start point generator in TMTO attacks. We investigate in detail other applications of LFSR-based parallel sequence generation in cryptologic algorithms. These are given in the appendix. In Chapter 6, we develop a systematic architecture for implementing TMTO attack. Finally, Chapter 7 discusses the effectiveness of TMTO and exhaustive search attacks. Concluding remarks are given in Chapter 8.

1.1 Outline and Main Contribution

The material that we include in this thesis are based on our six articles [24, 71, 72, 73, 74, 75] and is organized as follows.

In Chapters 2 and 3, we provide a survey on generic attacks and also a summary of state of the art on implementation of TMTO and exhaustive search attacks. Chapter 2 covers a survey on exhaustive search attack. In this part we also describe the work done on special purpose hardware for exhaustive search with special emphasis on DES. We present an overview of TMTO algorithms and summarize the work done so far on software and hardware implementations of such algorithms in Chapter 3.

In Chapter 4, we present a unifying framework for the analysis of multiple data trade-offs. Both Babbage-Golić (BG) and Biryukov-Shamir (BS) can be obtained as special cases of this framework. Moreover, we identify a new class of *single table* multiple data trade-offs which cannot be obtained either as BG or BS trade-off. We consider the analysis of the rainbow method of Oechslin and show that for multiple data, the TMTO curve of the rainbow method is inferior to the TMTO curve of the Hellman method. Finally, to increase the success probability in the presence of multiple data, we apply Kim and Matsumoto [58] style parametrization.

Chapter 5 is devoted to the study of function generation and start point generation in TMTO attacks. We suggest the use of LFSR sequences for function generation to be used in the rainbow TMTO. Properties of LFSR sequences such as long period, pseudo-randomness properties and efficient forward and backward generation make such sequences useful for the intended application. The time, memory and success probability of the multiple table rainbow method is analyzed in detail. All TMTO attacks require efficient generation of start points. For hardware implementation, we show that parallel generation of LFSR sequences are well suited as parallel start point generators.

Chapter 6 describes a systematic architecture for implementing TMTO. We break down the offline and online phases into simpler tasks and identify opportunities for pipelining and parallelism. This provides a detailed top level architecture.

In Chapter 7, we develop a cost-time-data trade-off model. This model is used to analyze the effectiveness of the exhaustive search and the TMTO precomputation for s-bit keys with $s \leq 128$. The analysis shows that $s \leq 96$ does not afford comfortable security while $s = 128$ appear to be secure in the foreseeable future. We apply our trade-off model to stream ciphers and find that 80-bit stream ciphers do not provide adequate protection against TMTO attacks. Further, we show that the A5/3 encryption algorithm used in GSM mobile phone [7] also does not provide adequate security. Finally, we compare the effectiveness of TMTO and exhaustive search approaches.

In Appendix, we describe two other applications of LFSR sequences. The first application is to improve the design of DES Cracker built by the Electronic Frontier Foundation in 1998 [3]. The second application is to present a variant of the counter mode of operation of a block cipher. (We thank a reviewer for pointing out an earlier suggestion by McGrew [67] for using LFSR sequences in the counter mode of operation. Compared to [67], we provide more details.) In an earlier work, Wiener [99] had presented a careful and detailed design of a hardware architecture for performing exhaustive search on DES. We do not improve upon this design.

Chapter 2

Definitions and Background Concepts

2.1 Introduction

The strongest goal of cryptanalysis is to recover the unknown key which has been used in an encryption process. Weaker goals, such as distinguishing ciphertexts from random bit strings have also been considered in the literature. It is assumed that the model of encryption and the ciphertexts are known to the adversary. Such an attack is called ciphertext only attack. Additionally, the adversary may know (or can choose) some plaintexts and obtain the corresponding ciphertexts. These attacks are called known (or chosen) plaintext attacks. In attacking certain types of encryption algorithms, it may be advantageous for the adversary to choose some ciphertexts and obtain the corresponding plaintexts. This is called a chosen ciphertext attack.

The above approaches can be combined leading to attacks such as chosen plaintext, chosen ciphertext attacks. Further, in the chosen plaintext or ciphertext attack, the adversary can behave adaptively, i.e., it obtains a response before choosing the next input. The weakest attack is the ciphertext only attack, while the adaptive attacks are considered to be the strongest attacks.

Block ciphers and stream ciphers are two types of symmetric key cryptosystems. See [22] for a survey of the state of the art in symmetric key encryption. In block ciphers, a plaintext is first partitioned into blocks and then each block is encrypted using the same key. Stream ciphers generate a keystream from the given secret key and use it to encrypt a plaintext string. See [68] for more details about block ciphers and stream ciphers and their differences. The literature has a vast discussion on block and stream ciphers [5, 44, 6, 83] and their cryptanalysis [65, 34, 85, 97]. Very often a successful attack exploits weaknesses in the design of the specific algorithm being considered. For example, linear and differential attacks try to

find the linear and differential characteristics between the plaintext and the corresponding ciphertext for a given encryption algorithm.

A generic approach for cryptanalysis views the encryption function as a black box, i.e., it does not utilize information about how the function is constructed. A simplest generic attack is to try every possible key until the correct one is found. This is called an exhaustive search attack. The importance of such an approach arises from the fact that if a cryptographic algorithm is not secure against exhaustive search, then it cannot be considered secure at all.

2.2 Preliminaries

2.2.1 Block Cipher

In block ciphers, the plaintext is divided into blocks of a fixed length and encrypted into blocks of ciphertexts using the same key. In Figure 2.1, we show the process of encrypting the plaintext X_0 under a typical r round block cipher to obtain the ciphertext X_r. Here X_i denotes the intermediate value of the block after i rounds of the encryption, so that $X_i = F_i(X_{i-1}\ ,\ k_i)$, where ($k_1$,$k_2$, ...,$k_r$) is the list of round keys derived from the secret key using a publicly known key scheduling algorithm. In modern ciphers, the secret key is between 128 and 256 bits long and for an r round iterated cipher, this is expanded into r round keys.

The Data Encryption Standard (DES) [5] has been the most widely used iterated block cipher since it was published in 1977. It has now been replaced by the Advanced Encryption Standard (AES) [6]. AES is a 128-bit block cipher with one of the three different key sizes, 128, 192 or 256 bits.

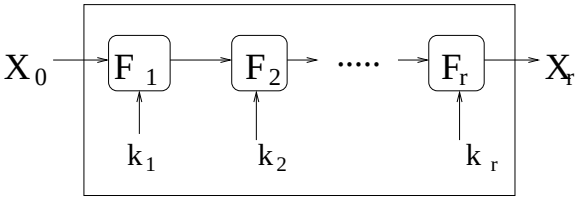

Figure 2.1: A typical r-round block cipher

2.2.2 Stream Cipher

Rueppel [86] described a key stream generator (see Figure 2.2) as follows: Let $\mathsf{S} = \{0,1\}^s$ be the set of internal states and $s_t \in \mathsf{S}$ be the internal state at time t. The total number of possible internal states is $N = 2^s$. Let $g : \mathsf{S} \to \mathsf{S}$ be the function that modifies the internal state at each clock (time) and $h : \{0,1\}^s \to \{0,1\}^m$ be the function which takes the internal state as input and computes an m-bit string at each clock. The method of encryption is also shown in Figure 2.2.

Modern stream ciphers use a nonce or an initial vector (IV) in addition to the secret key. A (key, IV) pair is loaded into the internal state of the cipher after which a number of cipher rounds (initialization algorithm) are executed without producing any output. After initialization is over, encryption proceeds as in Figure 2.2.

Turing [84], Scream [52], Rabbit [28], Sober [55], Snow [44] etc. are some well known stream ciphers. For the new stream ciphers one can see eSTREAM, the ECRYPT stream cipher project (http://www.ecrypt.eu.org/stream).

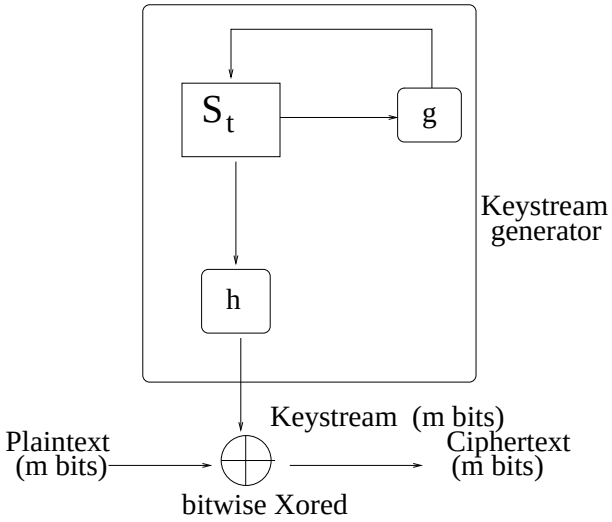

Figure 2.2: Keystream generator

2.2.3 One-Way Function

Informally speaking, a one-way function $f : A \to B$ satisfies the following two properties (see Figure 2.3) where A and B are two finite set,

- f is easy to compute.
- f is hard to invert, i.e., it is difficult to get x from $f(x)$.

See [49] for a formal definition of one-way function.

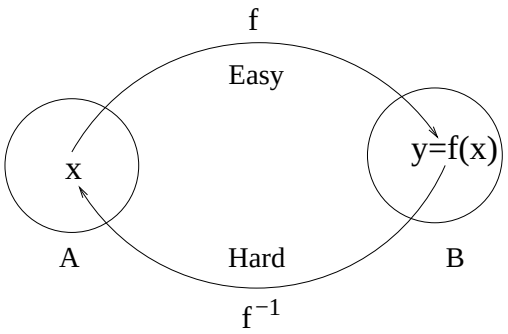

Figure 2.3: A typical one-way function

Block ciphers and stream ciphers can be viewed as one-way functions.

Block cipher as one-way function: Let $V_s = \{0,1\}^s$ be the set of all possible bit strings of length s. We take V_{s_1} and V_{s_2} to be the plaintext space and the ciphertext space respectively. Let $V = V_s$ be the key space (set of all possible keys). Then $N = 2^s$ is the total number of possible keys. An s_1-bit block cipher is a function $E : V_{s_1} \times V \to V_{s_2}$ and $\mathsf{cpr} = E_x(\mathsf{msg})$ denotes the ciphertext cpr for plaintext msg under key x. Let $\psi : V_{s_2} \to V_s$ be a function from ciphertexts to keys. One way of constructing the function ψ is as follow. If $s_2 > s$ (DES has $s_1 = s_2 = 64$ and $s = 56$) then we remove the first $(s_2 - s)$ bits. If $s_2 \leq s$ (AES has $s_1 = s_2 = 128$ and there are three allowable key lengths, $s = 128, 192$ and 256 bits) then we append $(s - s_2)$ constant bits.

For a fixed message msg, we define a function $f : V_s \to V_s$ as,

$$f(x) = \psi(E_x(\mathsf{msg})).$$

To get $y = f(x)$ from x we need to apply the encryption function under the known key x followed by a function ψ which is easy to compute. But to get x from $f(x)$ is hard as one has to decrypt the known plaintext msg under the unknown key x which is equivalent to the chosen plaintext attack to the cipher. Hence the function f can be viewed as a one-way function.

There can be other possible ways of constructing f. For example if $s = 2s_2$, i.e., the key length is twice the length of the ciphertext, then one can define $f(x) = E_x(\mathsf{msg})||E_x(E_x(\mathsf{msg}))$. The particular construction of f from $E_x()$ can influence the success of a cryptanalytic method. In this thesis, we do not consider the different possible constructions of f. Our focus is on generic methods to invert f.

Stream cipher as one-way function: There are two ways of defining a one-way function from a stream cipher.

(a) State to keystream map. $f : \mathsf{S} \to \{0,1\}^s$ as follow:

$$\text{state } S \mapsto \text{first } s \text{ bits generated by the stream cipher starting from state } S.$$

This f is said to be the state to prefix mapping. The function f is easy to compute in the forward direction. From a given internal state S to get $f(S)$, we just apply the keystream generator algorithm repeatedly until s key bits are generated. On the other hand, if it is possible to compute S from $f(S)$ then the stream cipher is broken. So for a secure stream cipher, inverting f should be hard. Hence f can be viewed as a one-way function.

(b) $(\text{key}, \text{IV}) \mapsto$ keystream map. For a k-bit stream cipher using an l-bit IV, the following $(k+l)$-bit one-way function f is constructed in [57]:

$$(k\text{-bit key}, l\text{-bit IV}) \mapsto (k+l)\text{-bit keystream prefix.}$$

To compute f in the forward direction, we first load the key and IV with the given value (k-bit key, l-bit IV) and then initialization algorithm is applied until the initial internal state is reached. After reaching the initial internal state, we repeatedly apply the keystream generator algorithm to get $(k+l)$ key bits. So the function f is easy to compute in the forward direction. Inverting this one-way function f will provide the secret key (note that the IV is public). Hence the function f can be viewed as a one-way function.

2.2.4 Problem Definition

Let $f : \{0,1\}^s \to \{0,1\}^s$ be the one-way function to be inverted. This function may be obtained from a block cipher by considering the map from the key space to the cipher space for a fixed message or from stream ciphers. Thus our problem will be: given a string y, we will have to find a string x (*preimage or key*) such that $f(x) = y$. We will denote $V = \{0,1\}^s$ as the search space and $N = 2^s$ as the size of the search space. Note that we require the range and domain of the function f to be the same because we will be computing iterates of the form $f^i(x)$ for $i \geq 1$.

2.3 Exhaustive Search

An exhaustive search attack tries out all possible keys one by one until the correct key is found.

1. $l \leftarrow 1$ to N do
2. $\quad X = \mathsf{next}(V)$;
3. $\quad$ if $(f(X) = y)$ return X as a key and *stop*;
4. end do
5. return failure;

For practical applications, the set V has an ordering such that $\mathsf{next}(V)$ returns the first unused element of V.

Suppose n search units (processors) $P_1, P_2, \ldots, P_n$ are available. The search space V is partitioned into n equal size disjoint subspaces, i.e., $V = V_1 \cup V_2 \cup \ldots \cup V_n$ such that $V_{i_1} \cap V_{i_2} = \phi$ for $i_1 \neq i_2$ and $|V_i| = \lfloor \frac{N}{n} \rfloor$ for $1 \leq i \leq n$. The processor P_i searches through the key subspace V_i for $i = 1, 2, \ldots, n$. Each processor P_i executes as follows.

P_i :
2. $j \leftarrow 1$ to $\lfloor \frac{N}{n} \rfloor$ do
3. $\quad X_i = \mathsf{next}(V_i)$;
4. $\quad$ if $(f(X_i) = y)$ return X_i as a key and *stop*;
6. end do
7. return failure;

The parallel search algorithm can be described as follows.

1. $i \leftarrow 1$ to n do in parallel
2. $\quad$ execute P_i;
3. if all P_is return failure then return failure.

The total number of f invocations required for the exhaustive search is N. The number of parallel rounds is $\frac{N}{n}$ when n processors are running in parallel. In the above, the subspaces $V_1, V_2, \ldots, V_n$ are disjoint. Quisquater and Desmedt [80] showed that using random subspaces corresponds to a loss of a factor 2 in the success probability.

Tabulation attack: Tabulation attack is the other extreme of the generic approach to invert a one-way function. It has two phases: offline and online. In the offline phase, a table consisting of all possible pairs $(x, y = f(x))$ is generated and stored sorted on the second

component. In the online phase, given y, a lookup into the table provides the corresponding x.

An exhaustive search requires 2^s invocations of f in the online phase and constant memory. On the other hand, a tabulation attack requires to store 2^s pairs of elements and requires negligible time in the online phase. The TMTO methodology described in Chapter 3 attempts to provide a trade-off between the online time and memory requirement.

2.3.1 Exhaustive Search on DES

In 1977, Diffie and Hellman [41] proposed an exhaustive search machine for DES using 10^6 DES chips. The cost of the machine was estimated to be around 20M USD and it was expected to find the key in 12 hours. In [40], Desmedt proposed a realistic exhaustive key search machine which breaks thousands of DES keys in an hour. A collision search technique was proposed by Quisquater and Delescaille [79] in 1989. Quisquater and Desmedt [80] demonstrated how to make a simple fault-tolerent exhaustive code-breaking machine using widely distributed processors.

In 1991, Eberle [43] showed an efficient DES implementation in GaAs hardware. A gate-level design of DES chip was proposed by Wiener [99] in 1993. Every chip uses 16 pipeline stages and can be operated at a clock frequency of 50 MHz. A machine consisting of 57600 DES chips was expected to recover the key in 3.5 hours. The total expected cost was 1M USD. The idea of using parallel LFSR sequence was suggested by Goldberg and Wagner [48] and was used by them in 1996 for an FPGA-based DES key search engine.

In 1997, a prize for cracking DES was announced at the annual RSA Cryptographic Trade Show and the prize was claimed in five months. Again in 1998 at the RSA show, the prize was offered. As a result, DES was broken in 39 days using exhaustive search on a network of PCs.

DES Cracker: In 1998, EFF (Electronics Frontier Foundation) built a machine for cracking DES [3] in 3.5 days with a total cost of 200,000 USD (80,000 USD for man power + 120,000 USD for production) by exhaustive search. The DES cracker is a ciphertext only attack where a PC drives many *search units* as follows. The key space is divided into segments and each search unit searches through one segment. A search unit takes a key and two ciphertexts as input, decrypts one ciphertext with the key and checks whether the resulting plaintext is interesting. If yes then it decrypts the second ciphertext using the same key and checks if it is also interesting. The search unit (hardware) selects a plaintext to be interesting if all its 8 bytes are ASCII, if not it ignores the plaintext. If both the plaintexts are found to be interesting (i.e., all 8 bytes of the plaintext are ASCII) then the (key, plaintext) pair is passed to PC to take the final decision. Otherwise, the search unit adds one to the key and

Table 2.1: Approximate time required for different architectures converted to the base year 2006 using Moore's law.

Architecture	Year of proposal	Required time	# 18 months periods upto 2006	Approximate time required in 2006
Wiener [99]	1993	3.5 hours	≈ 9	$\frac{3.5\times60\times60}{2^9} \approx 24.6$ seconds
EFF [3]	1998	3.5 days	≈ 5	$\frac{3.5\times24}{2^5} \approx 2.6$ hours
Kumar et al. [59]	2006	8.6 days	–	8.6 days

goes on searching for interesting plaintexts. After receiving a (key, plaintext) pair, the PC checks whether the plaintext which looked interesting to the hardware is an actual plaintext. If not, then PC returns a *false positive*.

A counter (adder) is used to add one to the key to get the next candidate key. A 32-bit adder is used so that it can count the bottom 32-bit of the key. Fixing the top 24-bit of the key, the search unit takes 1717 seconds to search all the possible keys having the same top 24 bits. At the end of this search the software stops the chip and resets the adder and places a new value in the top 24 bits and starts the search again. Each search unit is a DES chip and searches 2.5 million keys per second. A total of 24 search units fit inside a custom chip and search 60 million keys per second. A large circuit board contains 64 chips which searches 3.8 billion keys per second and 12 such boards are mounted into a chassis which searches 46 billion keys per second. Two chassis to search 92 billion keys per second, i.e., covering half of the key space in about 3.5 days.

Quisquater and Standaert [81] gave an estimate that 12000 USD search machine could break DES in 3 and 1/2 days by exhaustive search. This is a TMTO architecture. Kumar et al. [59] have built an FPGA based exhaustive search machine which has broken DES. This is a general purpose FPGA based programmable machine supporting a high degree of parallelism.

Moore's law: Moore's law states that for the same cost, the processing power approximately doubles every 18 months. Using this formulation, we can scale the time of different architectures to the base year 2006. Table 2.1 provides this comparison.

Chapter 3

Time/Memory Trade-Off Methodology

3.1 Introduction

Time/memory trade-off (TMTO) is a generic approach which can be carried out to invert any one-way function. It also appears in many search problems such as knapsack problem [70, 88] and discrete log problem [78]. TMTO is a trade-off between the exhaustive search attack and the tabular attack. In this chapter, we describe the various TMTO algorithms and related works.

3.2 A Historical Perspective of TMTO

The material in this Section is due to Palash Sarkar.

Enigma represents an important and significant step in mechanization of encryption procedure. A simplified description of the Enigma encryption algorithm can be found in [91]. At a conceptual level, it has two components – a scrambler and a plugboard with different settings. The total number of keys is the product of the number of possible scrambler settings and number of possible plugboard settings. This number is fairly large (around 2^{50}) and exhaustive search was not possible in the 1930's.

Keys were changed on a daily basis. Hence, all Enigma operators got the same Enigma day Key. Actual encryption of a message is done using a message key. The operator first encrypts the msg key using the day key and then the actual msg using the msg key. The msg key is encrypted *twice* to eliminate transmission errors.

The double encryption of the msg key proved to be weakness. By observing several such double encryptions of msg keys, Rejewski was able to construct a permutation of the Latin alphabet. Rejewski's main observation was that the cycle structure of the permutation depends only on the scrambler settings and is independent of the plugboard settings. The number of scrambler settings is around 100,000 for the early Enigma machines.

Rejewski's team spent one year in preparing a table of all possible scrambler settings and the cycle structure of the associated permutations. The table was indexed on the cycle structure. During the actual attack, the double encrypted msg keys were used to obtain a cycle structure. Using the precomputed table and the cycle structure the scrambler settings of the day key was obtained. The plugboard settings were then obtained by a separate procedure. This attack can be viewed as a TMTO where the table preparation time corresponds to precomputation; the tables correspond to memory; the processing of the msg keys and the lookup into the table correspond to the online phase of the attack.

Note that this is not a generic inversion of one-way function. In fact, this early twentieth century attack combines TMTO ideas with divide-and-conquer technique (of separating the attack on scrambler settings from that of plugboard settings).

As the design of the Enigma machine changed over the years, the hand calculation of the tables became infeasible. Instead special purpose electro-mechanical machine called **bombes** were built which ran in parallel to perform similar tasks. These ideas were later significantly developed by Turing during World War-II. In fact, Turing designed significantly more complex cryptanalytic machines for breaking more sophisticated versions of Enigma.

Another encryption algorithm used by the Germans was called the Lorentz machine. According to Simon Singh [91], the world's first general purpose computer was Colossus, which was actually conceived as a special purpose cryptanalytic machine. In more recent times, several proposals of special purpose hardware for performing the sieving step of the number field sieve algorithm were made. See [94] for a compilation of literature on special purpose cryptanalytic hardware.

3.3 Time/Memory Trade-Off Methodology

TMTO attack has basically two phases: a precomputation phase which is an offline activity followed by an online phase. In the precomputation phase, a (set of) table(s) is computed and only a part of the table(s) is stored which incurs a cost in the online phase. This leads to a trade-off between the memory (storage) and time required in the online phase.

3.3.1 Hellman Method

In the Hellman method, a chain of length t generated from a start point x_0 is defined as

$$x_0 \xrightarrow{f} x_1 \xrightarrow{f} x_2 \rightarrow \cdots \rightarrow x_{t-2} \xrightarrow{f} x_{t-1} \xrightarrow{f} x_t$$

To construct a Hellman table of size $m \times t$, we choose m start points uniformly at random from the key space and generate the chains of length t each (see Figure 3.1). The i^{th} start point is $x_{i,0}$. The start points and the end points of the matrix are stored in the table, sorted in the increasing order of the end points.

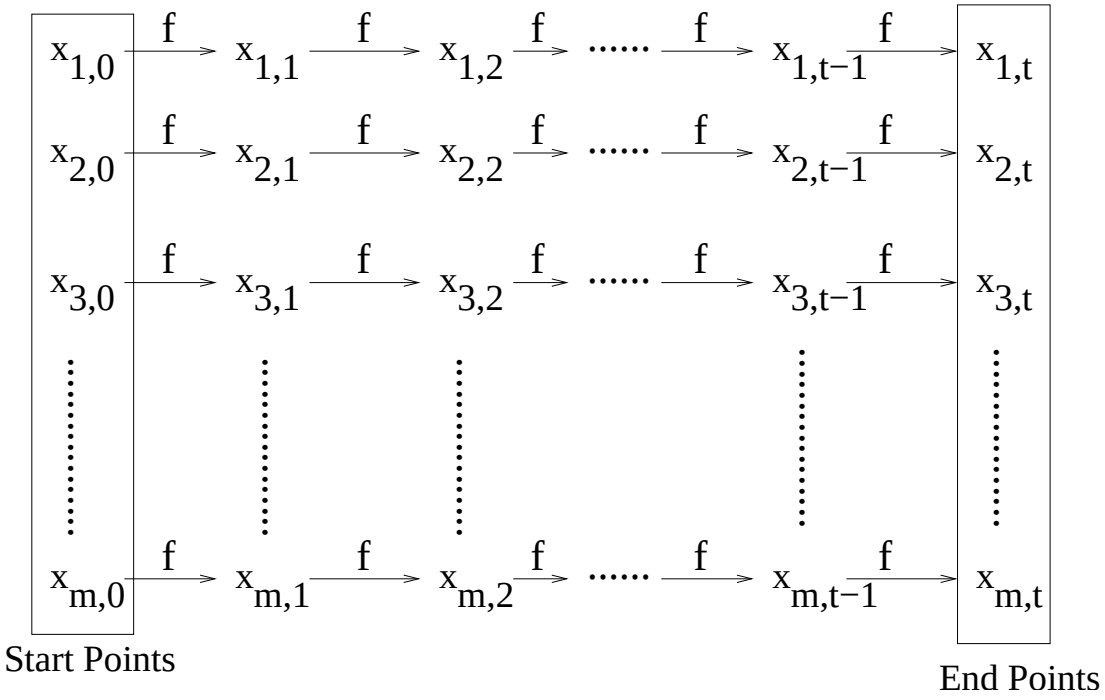

Figure 3.1: Single Hellman table with size $m \times t$

If we go on adding more rows to the matrix, then we reach a situation when there will be some repetition of the points in the matrix. To find the critical value of m, we assume that the matrix of size $m \times t$ contains exactly mt distinct points and another row is likely to contain exactly t distinct points. By the birthday paradox, these two sets are disjoint as long as $t \times mt \leq N$ and thus the critical values of m and t should satisfy $mt^2 = N$. This is called the *matrix stopping rule* [25]. One table can cover only a fraction $\frac{mt}{N} = \frac{1}{t}$ of the N

points. Hence t different (unrelated) tables are needed to cover all N keys. These are created as follows. For the t tables, t different functions $f_1, \ldots, f_t$ are used where each f_i is a simple output modification of the function f, i.e., $f_i(x) = \psi_i(f(x))$ where ψ_i is the output modification function. The success probability analysis assumes $f_1, f_2, \ldots, f_t$ to be independent random functions. The total memory requirement is $m \times t$ (start point, end point) pairs. To give the formal algorithm of the Hellman method we will use the following data structures.

Data structures:

- $\mathsf{SP}[1, 2, \ldots, t][1, 2, \ldots, m]$: The entry $\mathsf{SP}[i][j]$ is the j^{th} start point for the i^{th} table. The entries of $\mathsf{SP}[\,][\,]$ are random s-bit strings.
- $\mathsf{EP}[1, 2, \ldots, t][1, 2, \ldots, m]$: The entry $\mathsf{EP}[i][j]$ is the j^{th} end point for the i^{th} table.

We now give the formal algorithm to construct the tables.

Algorithm Precomputing Tables
Input: SP[ ][ ]
Output: EP[ ][ ]
1. $i \leftarrow 1$ to t do
2. $j \leftarrow 1$ to m do
3. $\mathsf{tmp} \leftarrow f_i(\mathsf{SP}[i][j])$
4. $l \leftarrow 1$ to $t - 2$ do
5. $\mathsf{tmp} \leftarrow f_i(\mathsf{tmp})$
6. end do
7. $\mathsf{EP}[i][j] \leftarrow \mathsf{tmp}$
8. end do
9. $\mathsf{Sort}(\mathsf{SP}[i], \mathsf{EP}[i])$
10. end do

The algorithm Sort (in Line 9) is described as follows: Note that $\mathsf{SP}[i]$ and $\mathsf{EP}[i]$ are arrays of length m each. We store the j^{th} elements of $\mathsf{SP}[i]$ and $\mathsf{EP}[i]$, i.e. $(\mathsf{SP}[i][j], \mathsf{EP}[i][j])$ into the j^{th} element of an array of ordered tuple for $j = 1, 2, \ldots, m$. Now we sort this array in increasing order of the second components. After sorting, the first and the second component of the j^{th} element are respectively copied into $\mathsf{SP}[i][j]$ and $\mathsf{EP}[i][j]$.

In the online phase, given y, it is required to find x such that $y = f(x)$. The t tables are searched one after another. The search for x in the i^{th} table is as follows. We repeatedly apply f_i to $\psi_i(y)$ at most t times and after each application we check whether the output of f_i is in the set of end points of the i^{th} table. The number of table lookups for this is at most t. If the output is an end point then we come to the corresponding start point and

repeatedly apply the function f_i until it reaches $\psi_i(y)$. The previous value it visited is x. The total runtime for searching in all the tables is $t^2 + t \approx t^2$ invocations of f and t^2 table lookups. Note that finding a match does not necessarily imply that the key is in the table. This could be a false alarm which we will discuss in Section 3.3.2. Hence after getting x we need to check whether $y = f(x)$.

Algorithm Search
Input: SP[ ][ ], EP[ ][ ], y.
Output: x such that $f(x) = y$, else failure.
1. $i \leftarrow 1$ to t do
2. tmp $\leftarrow \psi_i(y)$
3. $j \leftarrow 0$ to $t-1$ do
4. $q \leftarrow$ Find(EP$[i]$, tmp)
5. if ($q \neq$ NULL) then
6. val $\leftarrow$ SP[i][q]
7. $l \leftarrow 1$ to $t-1-j$ do
8. val $\leftarrow f_i$(val)
9. end do
10. if f(val) $= y$ then return val
11. else raise a false alarm
12. end if
13. tmp $\leftarrow f_i$(tmp)
14. end do
15. end do
16. return "failure".

The subroutine Find (which is used in Line 4) does the following: Note that EP$[i]$ is a sorted array of length m. We apply binary search technique to get the position of tmp and return the position if it is in the array, otherwise return NULL.

Trade-off curve: Let M and T be respectively the memory required and time required in the online phase. Then $M = mt$ and $T = t^2$. Also from the matrix stopping rule we have $m \times t^2 = N$. Eliminating m and t, the trade-off curve is obtained as $TM^2 = N^2$. The precomputation time $P = N$ since the total number of elements in all the matrices is N.

Merging Considerations: We define merging between two chains L_1 and L_2 (two rows of same or different tables) as follows:

Definition 1 (Merging between two chains L_1 and L_2): *Let us consider the two chains*

L_1 and L_2 as

$$x_{1,0} \longrightarrow x_{1,1} \longrightarrow x_{1,2} \rightarrow \ldots \rightarrow x_{1,t-2} \longrightarrow x_{1,t-1}$$

$$x_{2,0} \longrightarrow x_{2,1} \longrightarrow x_{2,2} \rightarrow \ldots \rightarrow x_{2,t-2} \longrightarrow x_{2,t-1}$$

We say that L_1 and L_2 merge if there exists i_1 and i_2 such that $x_{1,i_1} = x_{2,i_2}$ (which is called a collision) and $x_{1,i_1+j} = x_{2,i_2+j}$ for all $j \geq 1$.

There are two types of collisions possible in the Hellman tables.

- **Type 1:** Collision within a table, i.e., we have two chains C_1 and C_2 in a table having a common value a (say).

  $$\ldots \rightarrow x_{1,i-1} \xrightarrow{f_1} a \xrightarrow{f_1} x_{1,i+1} \rightarrow \ldots \rightarrow x_{1,t-2} \xrightarrow{f_1} x_{1,t-1}$$

  $$\ldots \rightarrow x_{2,j-1} \xrightarrow{f_1} a \xrightarrow{f_1} x_{2,j+1} \rightarrow \ldots \rightarrow x_{2,t-2} \xrightarrow{f_1} x_{2,t-1}$$

  So $x_{1,i+l} = x_{2,j+l}$ for $l \geq 0$. This leads to a merge.

- **Type 2:** Collision between different tables, i.e., we have two chains L_1 and L_2 in two different tables having a common value a (say).

  $$\ldots \rightarrow x_{1,i-1} \xrightarrow{f_1} a \xrightarrow{f_1} x_{1,i+1} \rightarrow \ldots \rightarrow x_{1,t-2} \xrightarrow{f_1} x_{1,t-1}$$

  $$\ldots \rightarrow x_{2,j-1} \xrightarrow{f_2} a \xrightarrow{f_2} x_{2,j+1} \rightarrow \ldots \rightarrow x_{2,t-2} \xrightarrow{f_2} x_{2,t-1}$$

  So $x_{1,i+1}$ is not necessarily equal to $x_{2,j+1}$. Thus Type 2 collision does not necessarily lead to a merge.

In the Hellman method, a merge can occur only with the Type 1 collisions. Less merging means the number of distinct keys covered by the tables is larger which increases the success probability.

Let us consider a chain with $x_{1,0}$ as start point as follows.

$$x_{1,0} \longrightarrow x_{1,1} \longrightarrow x_{1,2} \rightarrow \ldots \rightarrow x_{1,t-2} \longrightarrow x_{1,t-1} \ldots$$

We say that the chain is a *cycling chain* if there exists i_1 and i_2 ($> i_1$) such that $x_{1,i_2} = x_{1,i_1} = x^*$ (say), i.e.,

$$x_{1,0} \longrightarrow x_{1,1} \rightarrow \ldots \rightarrow x_{1,i_1-1} \longrightarrow x^* \rightarrow \ldots \rightarrow x_{1,i_2-1} \longrightarrow x^* \rightarrow \ldots \rightarrow x^* \ldots$$

Hence cycling chains in the tables reduce the total number of distinct keys covered by the tables which in turn reduces the probability of success.

Success Probability: Probability that a given unknown key x is in a table is the ratio

$$\frac{\text{number of distinct keys in all the tables}}{N}.$$

The numerator depends on the choice of the first column of the table which is random. So the numerator is a random variable X (say). Hence the success probability $\mathsf{PS} = \mathsf{E}(\frac{X}{N}) = \frac{\mathsf{E}(X)}{N}$. Hellman [56] suggested to choose the start points uniformly at random from the set of all possible start points. Hellman also considered the issue of choosing the start points to be distinct and noted that this situation is difficult to analyse.

The following result and its proof are from [56].

Lemma 2 *Assuming that the encryption function is a random function*

$$\mathsf{E}(X) \geq \sum_{i=1}^{m}\sum_{j=1}^{t}\left(1-\frac{it}{N}\right)^{j}.$$

Proof : Let the table be

$$\begin{pmatrix} x_{1,0} & x_{1,1} & \dots & x_{1,j} & \dots & x_{1,t-1} \\ x_{2,0} & x_{2,1} & \dots & x_{2,j} & \dots & x_{2,t-1} \\ \vdots & & & & & \\ x_{i,0} & x_{i,1} & \dots & x_{i,j} & \dots & x_{i,t-1} \\ \vdots & & & & & \\ x_{m,0} & x_{m,1} & \dots & x_{m,j} & \dots & x_{m,t-1} \end{pmatrix}$$

Let $Y_{i,j}$ be the event that $x_{i,j}$ is new, i.e.,

$$x_{i,j} \notin \left(\{x_{i_1,j_1} : i_1 = 1,2,\dots,i-1,; j_1 = 1,2,\dots,t\} \cup \{x_{i,j_2} : j_2 = 1,2,\dots,j-1\}\right).$$

Let $\mathsf{I}(Y_{i,j})$ be the indicator function of the event $Y_{i,j}$. Then

$$\begin{aligned} \mathsf{E}(X) &= \mathsf{E}\sum_{i=1}^{m}\sum_{j=1}^{t}\mathsf{I}(Y_{i,j}) \\ &= \sum_{i=1}^{m}\sum_{j=1}^{t}\mathsf{Prob}(Y_{i,j}). \end{aligned}$$

Now

$$\begin{aligned} \mathsf{Prob}(Y_{i,j}) &\geq \mathsf{Prob}(Y_{i,1} \cap Y_{i,2} \cap \ldots \cap Y_{i,j}) \\ &= \mathsf{Prob}(Y_{i,1}) \times \mathsf{Prob}(Y_{i,2}|Y_{i,1}) \times \ldots \times \mathsf{Prob}(Y_{i,j}|Y_{i,1} \cap Y_{i,2} \cap \ldots \cap Y_{i,j-1}) \\ &= \frac{N-n_{i,1}}{N} \times \frac{N-n_{i,1}-1}{N} \times \frac{N-n_{i,1}-2}{N} \times \ldots \times \frac{N-n_{i,1}-(j-1)}{N} \end{aligned}$$

where $n_{i,j}$ is the number of distinct elements covered so far. So $n_{i,1}$ is the number of keys covered upto row $(i-1)$. The total number of elements in $(i-1)$ rows is $(i-1)t$. Thus

$$n_{i,1} \leq (i-1)t,$$

i.e, $n_{i,1} + j \leq it$ for $j \leq t$ and $\mathsf{Prob}(Y_{i,j}) \geq (\frac{N-it}{N})^j$. Hence

$$\mathsf{E}(X) \geq \sum_{i=1}^{m}\sum_{j=1}^{t}(1-\frac{it}{N})^j.$$

This completes the proof. ■

Using Lemma 2 we get a lower bound for the success probability

$$\begin{aligned}\mathsf{PS} &= \frac{\mathsf{E}(X)}{N} \\ &\geq \frac{1}{N}\sum_{i=1}^{m}\sum_{j=1}^{t}\left(1-\frac{it}{N}\right)^j.\end{aligned}$$

Let PS_t be the success probability when t Hellman tables are used. Let A_i be the event that the key is not in the i^{th} table for $i = 1, 2, \ldots, t$. The events A_i are independent since the functions f_i are assumed to be independent random functions. Also $\mathsf{Prob}(A_i) = 1 - \mathsf{PS}$. Now

$$\begin{aligned}\mathsf{PS}_t &= 1 - \mathsf{Prob}(A_1 \cap A_2 \cap \ldots \cap A_t) \\ &= 1 - \prod_{i=1}^{t}\mathsf{Prob}(A_i) \\ &= 1 - (1-\mathsf{PS})^t.\end{aligned}$$

The following analysis is taken from [58, 60, 66]. Consider r Hellman tables of size $m \times t$ each. The success probability is then

$$\mathsf{Prob}(r, m, t) = 1 - (1-\mathsf{PS})^r$$

where PS is the success probability for a single table. From Lemma 2, we know

$$\begin{aligned}\mathsf{PS} &\geq \frac{1}{N}\sum_{i=1}^{m}\sum_{j=1}^{t}\left(1-\frac{it}{N}\right)^j \\ &\approx \frac{1}{t}\sum_{i=1}^{m}\frac{1-e^{\frac{-it^2}{N}}}{\frac{it}{N}}\frac{t}{N} \\ &\approx \frac{1}{t}\int_0^{\frac{mt}{N}}\frac{1-e^{-tx}}{x}dx \\ &\approx h(u)\frac{mt}{N}\end{aligned}$$

where $h(u) = \frac{1}{u}\int_0^u \frac{1-e^{-x}}{x}dx$ and $u = \frac{mt^2}{N}$. Hence

$$\begin{aligned}
\mathsf{Prob}(r,m,t) &= 1-(1-\mathsf{PS})^r \\
&\geq 1-(1-h(u)\frac{mt}{N})^r \\
&\approx 1-\mathsf{e}^{-h(u)\frac{rmt}{N}} \\
&= 1-\mathsf{e}^{-h(u)\times\lambda} \\
&= \mathsf{P}(u,\lambda)(say)
\end{aligned}$$

where

$$\lambda = \frac{rmt}{N}.$$

Now $h(u) = \frac{1}{u}\int_0^u \frac{1-e^{-x}}{x}dx < 1$ for all $u > 0$. This implies

$$\mathsf{P}(u,\lambda) < 1-e^{-\lambda}. \tag{3.1}$$

For $\lambda = 1$ we get

$$\mathsf{P}(u,1) < 1-e^{-1} \approx 0.63.$$

This shows that we cannot achieve success probability more than 0.63 with $\lambda = 1$. To attain higher success probability we have to use larger values of λ, for example,

$$\mathsf{P}(u,2) < 0.86; \mathsf{P}(u,3) < 0.95; \mathsf{P}(u,4) < 0.98.$$

Hence, to increase the success probability we have to increase the value of λ which in turn increases the size of the search space ($r \times m \times t = \lambda \times N$). But to increase the search space (amount of coverage), we have to increase either the number of tables (r) or the number of rows (m) or the number of columns (t). This increases the runtime ($r \times t$) or the amount of memory ($2rm$) and also the precomputation time ($P = r \times m \times t = \lambda \times N$). With fixed runtime and memory requirement, how to achieve high success probability is discussed in [58]. They set the amount of memory requirement $2rm = \frac{N}{a}$ and the runtime $rt = \frac{N}{b}$, where a and b are constants. Thus

$$m = \lambda b, t = 2\lambda a, r = \frac{N}{2\lambda ab}.$$

Equation (3.1) shows that in order to get the higher success probability, one has to increase the value of λ, which in turn increases m, t and decreases r. By the matrix stopping rule, $mt \times t \leq N$, i.e., $4\lambda^3 a^2 b \leq N$.

3.3.2 False Alarms

When we search a table for the key, finding a matching end point does not necessarily imply that the key is in the table. This is because the key may be a part of a chain that has the same end point but is not in the table. It is called a *false alarm*. In [56], Hellman proved that the expected number of false alarms for an $m \times t$ table can be bounded above by $\frac{mt(t+1)}{2N}$. Hellman also pointed out that when a false alarm occurs, at most t invocations of f are required to rule it out. If $mt^2 = N$ then the above bound becomes $\frac{1}{2}$. The number of false alarms can be as large as half the total number of f invocations required for the online phase. Hellman makes a worst case analysis of the number of false alarms. As pointed out by a reviewer of this thesis, the average case analysis of false alarms is an open problem.

3.3.3 DP Method

Rivest [39] introduced the idea of using distinguished points (DP) in time/memory trade-off cryptanalysis to reduce the number of table lookups.

Definition 3 (DP-property of order p [29]): *Let $p \in \{1, 2, \ldots, s-1\}$ where $V = \{0,1\}^s$ is the key space. Then DP-property of order p is an easily checked property which holds for 2^{s-p} different elements of V.*

Definition 4 (Distinguished Point(DP)): *Let $x \in V$ be a key. Then x is a DP of order p if the DP-property of order p defined above holds for x.*

For example, we may define a DP property on V as follows: a key $x \in V$ is a DP if its first p bits are zero. It is desirable for each chain to end in a DP within at most t iterations where t is the maximum chain length. This motivates a choice of p to be equal to t. In the precomputation phase, we generate r tables with maximum chain length t as follows. We choose r different functions $f_1, \ldots, f_r$, where each f_i is a simple output modification of the function f, i.e. $f_i(x) = \psi_i(f(x))$, where ψ_i is the i^{th} output modification function. For each table, we choose m start points uniformly at random from the key space. In the i^{th} table, for each start point we generate a chain by repeatedly applying f until we reach a DP or until length of the chain is t. If a DP is encountered in the chain, then we store the tuple (start point, DP point, length of the chain) in the table, otherwise the chain will be discarded. We sort the tuple in the increasing order of the end points (DP). If the same DP occurs in two different tuples, then the tuple with maximum chain length will be stored. Sort the tuples in the increasing order of the end points. To give the formal algorithm of the DP method we will use the following data structures.

Data structures:

- $\mathsf{SPD}[1, 2, \ldots, r][1, 2, \ldots, m]$: The entry $\mathsf{SPD}[i][j]$ is the j^{th} start point for the i^{th} table. The entries of $\mathsf{SPD}[\ ][\ ]$ are random s-bit strings.
- $\mathsf{EPD}[1, 2, \ldots, r][1, 2, \ldots, m]$: The entry $\mathsf{EPD}[i][j]$ is the j^{th} end point for the i^{th} table. The entries of $\mathsf{EPD}[\ ][\ ]$ are initialized by the empty string.
- $\mathsf{LEN}[1, 2, \ldots, r][1, 2, \ldots, m]$: The entry $\mathsf{LEN}[i][j]$ is the j^{th} chain length of the i^{th} table. The entries of $\mathsf{LEN}[\ ][\ ]$ are initialized by zero.

More formally, we have the following algorithm for table construction.

Algorithm Precomputing Tables
Input: $\mathsf{SPD}[\ ][\ ]$
Output: $\mathsf{EPD}[\ ][\ ]$ and $\mathsf{LEN}[\ ][\ ]$
1. $i \leftarrow 1$ to r do
2. $\quad j \leftarrow 1$ to m do
3. $\quad\quad \mathsf{val} \leftarrow f_i(\mathsf{SPD}[i][j])$
4. $\quad\quad l \leftarrow 1$ to t do
5. $\quad\quad\quad$ if val is a DP, then
6. $\quad\quad\quad\quad\quad \mathsf{EPD}[i][j] = \mathsf{val}$; $\mathsf{LEN}[i][j] = l$; break
7. $\quad\quad\quad \mathsf{val} \leftarrow f_i(\mathsf{val})$
8. $\quad\quad$ end do
9. $\quad$ end do
10. $\quad \mathsf{Sort}(\mathsf{SPD}[i], \mathsf{EPD}[i])$
$\quad$ (if several end points are same then the tuple with the largest l is stored)
11. end do

The above algorithm keeps the chain with the longest length. Other variants are possible.

In the search stage: given a ciphertext $y = f(x)$, we need to find the unknown key x, assuming that x is in one of the constructed tables. To search in the i^{th} table, we first apply ψ_i on y. We then repeatedly apply f_i until we encounter a DP or until the chain length is maximum for the table. If we reach a DP then we perform a table lookup to check whether this DP is in the table or not. If yes we come to the corresponding start point and repeatedly apply f_i (for the i^{th} table) until it reaches $\psi_i(y)$. Then the previous point it visited is x or it is a false alarm. For each table the number of table lookups is only 1, whereas for the Hellman method it is t (number of columns in a Hellman table). Thus the DP method reduces the memory requirement and substantially reduces the memory access rate. The search algorithm is formally described below.

Algorithm Search
Input: SPD[ ][ ], EPD[ ][ ], LEN[ ][ ] and y.
Output: x such that $f(x) = y$, else failure.
1. $i \leftarrow 1$ to r do
2. tmp $\leftarrow \psi_i(y)$
3. $j \leftarrow 0$ to $t-1$ do
4. if tmp is DP break
5. tmp $\leftarrow f_i$(tmp)
6. end do
7. if tmp is DP then
8. $q \leftarrow$ Find(EPD$[i]$, tmp)
9. if ($q \neq$ NULL) then
10. val $\leftarrow$ SPD[i][q]
11. $l \leftarrow 1$ to LEN[i][q] $- j$ do
12. val $\leftarrow f_i$(val)
13. end do
14. if f(val) $= y$ then return val
15. else raise a false alarm
16. end if
17. end if
18. end do
19. return "failure".

Analysis: In both the above algorithms (precomputation and online phases), chains are computed from a start point until a DP is encountered or until the chain length is maximum. So the probability that a chain started from a randomly chosen point will reach a DP after certain steps plays an important role in this method. In the literature, there is no precise guideline about how to choose the parameters (r, m, t) though some suggestions are made in [30, 31]. In [92], Standaert et al. provided a theoretical analysis of the DP method and gave some supporting experimental results. The expected number of chains and length of the chains play an important role in the memory requirement, runtime and success probability. Let α and β respectively be the expected number of chains and chain length in a table. Some empirical values of the parameters are given in [29]. Determining these exactly is a candidate for further research.

Let us now consider the case when the start points are DPs [31]. In a table the (expected) number of chains is α. For each table, we store the tuples containing two keys (DP of order p) which requires $2(s-p)$ bits storage. So for each table we need $2(s-p)\alpha$ bits storage. Hence the total memory requirement is $2r(1-\frac{p}{s})\alpha$ units (1 unit=s bits). For each table, the expected number of iterations is same as the chain length β, so the total runtime is $r \times \beta$.

Success Probability: Note that for the DP method there is no overlap (no cycling chain and no merge between two chains) within a single table. So the probability that the key is in a single table is:

$$\frac{E(\text{the number of points covered by the table})}{N} = \frac{\alpha \times \beta}{N}.$$

Hence the success probability for r tables is:

$$1 - \left(1 - \frac{\alpha\beta}{N}\right)^r \approx 1 - e^{-\frac{r\alpha\beta}{N}}.$$

3.3.4 Rainbow Method

Oechslin [76] proposed a variant of Hellman's attack called the rainbow method. Rainbow chains are used in the rainbow tables. To construct a rainbow chain of size t we choose t functions $f_0, f_1, \ldots, f_{t-1}$, which are again simple output modifications of f. As in Hellman's method, $f_0, f_1, \ldots, f_{t-1}$ are assumed to be independent random functions. Taking a start point $x_{1,0}$, a rainbow chain is generated as follows:

$$x_{1,0} \xrightarrow{f_0} x_{1,1} \xrightarrow{f_1} x_{1,2} \rightarrow \cdots \rightarrow x_{1,t-1} \xrightarrow{f_{t-1}} x_{1,t}$$

To construct a rainbow table of size $mt \times t$, we choose mt random points from the key space. Taking each key as a start point we generate a rainbow chain. So to get the j^{th} column, we apply f_j on each element in the $(j-1)^{th}$ column, i.e.

$$x_{i,j-1} \xrightarrow{f_j} x_{i,j}, \text{ for } 1 \leq i \leq mt.$$

Similar to Hellman's method, we store the start and the end points in the rainbow table sorted in the increasing order of end points. To describe the formal algorithm we will use the following data structures.

Data structures:

- $\mathsf{SPR}[1, 2, \ldots, mt]$: The entry $\mathsf{SPR}[i]$ is the i^{th} start point. The elements of SPR are random s-bit keys.
- $\mathsf{EPR}[1, 2, \ldots, mt]$: $\mathsf{EPR}[i]$ is the i^{th} end point.

We now give the formal algorithm to construct the tables. The algorithm Sort used in the following algorithm is the same as that in the Hellman method.

Algorithm Precomputing Tables
Input: SPR[ ]
Output: EPR[ ]
1. $i \leftarrow 1$ to mt do
2. $\quad \mathsf{tmp} \leftarrow f_0(\mathsf{SPR}[i])$
3. $\quad l \leftarrow 1$ to $t-1$ do
4. $\quad\quad \mathsf{tmp} \leftarrow f_l(\mathsf{tmp})$
5. $\quad$ end do
6. $\quad \mathsf{EPR}[i] \leftarrow \mathsf{tmp}$
7. end do
8. $\mathsf{Sort}(\mathsf{SPR}, \mathsf{EPR})$

In the online phase, we will be given $y = f(x)$ and have to find the key x. For $0 \leq j \leq t-1$, we apply ψ_j to y and compute $y_0 = f_{t-1}(f_{t-2}(\ldots(f_{j+1}(\psi_j(y))\ldots)$. If y_0 is in the last column of the table then let x_0 be the corresponding start point. This gives us the equations:

$$\left.\begin{array}{rcl} y_0 & = & f_{t-1}(f_{t-2}(\ldots(f_{j+1}(\psi_j(y))\ldots) \\ & = & f_{t-1}(f_{t-2}(\ldots(f_1(f_0(x_0))\ldots) \\ & = & f_{t-1}(f_{t-2}(\ldots(f_{j+1}(f_j(x))\ldots) \\ & = & f_{t-1}(f_{t-2}(\ldots(f_{j+1}(\psi_j(f(x)))\ldots) \end{array}\right\} \tag{3.2}$$

where $x = f_{j-1}(f_{j-2}(\ldots(f_1(f_0(x_0))\ldots)$. The first equality follows by the online search condition and the second equality follows from the table construction. From the first and last row of (3.2) we would like to infer that $y = f(x)$, i.e., x is a preimage of y. Note that this might not always hold, leading to a false alarm. During the actual attack, this needs to be verified. The total runtime is $\frac{t(t-1)+2t}{2} \approx \frac{t^2}{2}$ invocations of f.

We now describe the search algorithm in a more formal way. The algorithm Find used in this search algorithm is the same as that in the Hellman method.

Algorithm Search
Input: SPR[ ], EPR[ ] and y.
Output: x such that $f(x) = y$, else failure.
1. $j \leftarrow t-1$ down to 0 do
2. $\mathsf{tmp} \leftarrow \psi_j(y)$
3. $l \leftarrow j-1$ to $t-1$ do
4. $\quad \mathsf{tmp} \leftarrow f_l(\mathsf{tmp})$
5. end do
6. $q \leftarrow \mathsf{Find}(\mathsf{EPR}, \mathsf{tmp})$
7. if $(q \neq \mathsf{NULL})$ do
8. $\quad \mathsf{val} \leftarrow \mathsf{SPR}[\mathsf{q}]$

9. $l \leftarrow 0$ to $j-1$ do
10. $\mathsf{val} \leftarrow f_l(\mathsf{val})$
11. end do
12. if $f(\mathsf{val}) = y$ then return val
13. else raise a false alarm
14. end if
15. end do
16. return "failure".

The same technique can be applied to multiple rainbow tables, even though this is not explicitly mentioned in the paper [76] but appears in the implementation [9]. If r tables are used then the runtime increases to $rt^2/2$ and hence for practical implementation, r will be a small constant.

Merging consideration: In Figure 3.2, t Hellman tables with size $m \times t$ each are replaced by a single rainbow table of size $mt \times t$. In the rainbow table, having a common value in two different chains will lead to a merge only if the common value occurs in the same column, since the same sequence of functions are applied to the common value. Having a common value in two different columns will not lead to a merge due to the fact that different sequences of functions are applied to the common value. In the Hellman method, a merge can occur only for a collision within a table (i.e., within a set of mt points), whereas in a rainbow table, a merge can occur only with a collision within a column (i.e., again within a set of mt points). The number of tables in the Hellman method and the number of columns in the rainbow method are the same. So the merging effects and the success probabilities of both the methods are the same.

An approximate expression for the success probability of the rainbow method with table size $m_1 \times t_1$ is given [76] as follows: $\mathsf{Prob} = \left(1 - \prod_{i=1}^{t_1}(1 - \frac{m_i}{N})\right)$, where m_i denotes the number of distinct points in the i^{th} column of the table. The recursive expression for m_i is $m_{i+1} = N \times \left(1 - e^{-\frac{m_i}{N}}\right)$, where m_1 is the number points (assumed all distinct) in the first column.

Trade-off curve: In the rainbow table, we store mt pairs of the (start point, end point). Hence, the amount of memory required $M = mt$, which is same as the memory required for the Hellman method. The time required at the online stage is $T = \frac{t^2}{2}$, which is half of the online time required for the Hellman method. Total number of points covered by the rainbow table is $mt \times t$ which is equal to N, i.e., $mt^2 = N$. Eliminating m and t, we obtain the trade-off curve as $TM^2 = N^2$.

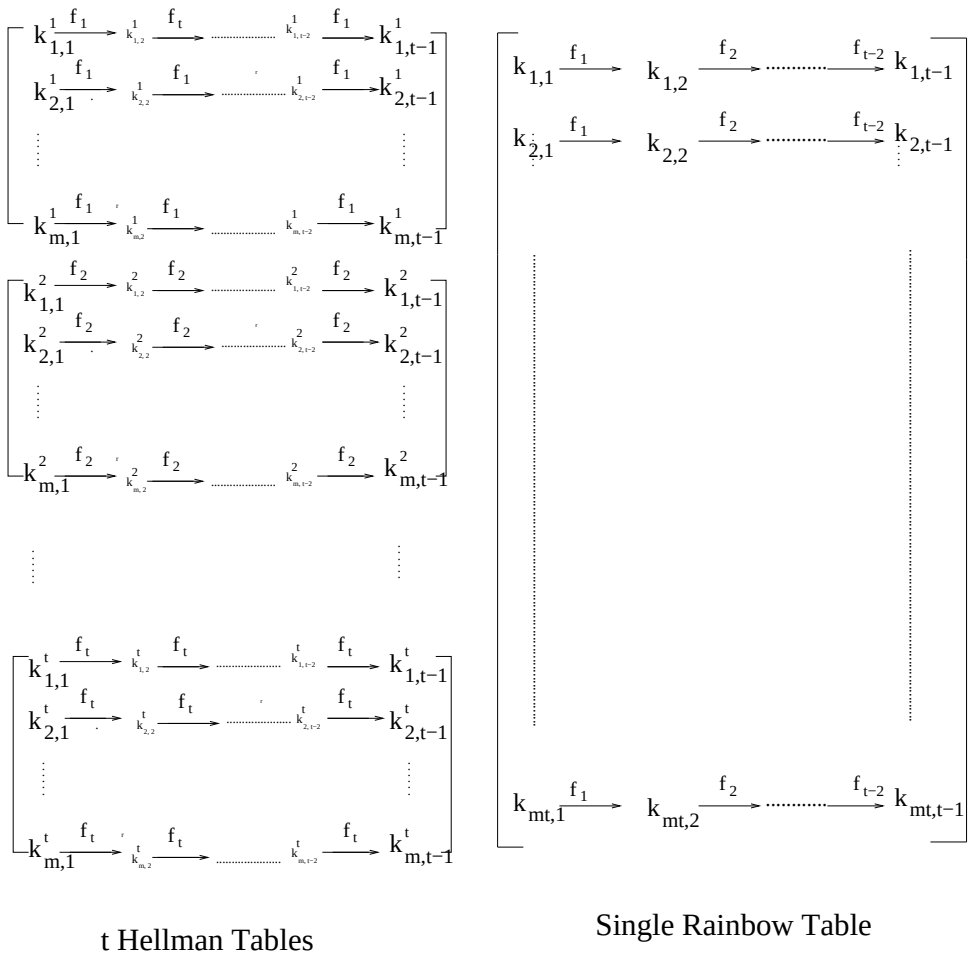

Figure 3.2: t Hellman tables each with size $m \times t$ are replaced by a single rainbow table with size $mt \times t$.

Barkan et al. [14] provided rigorous bounds on TMTO attacks. They formalized a general model of TMTO attack by introducing the new notion of stateful random graphs, where the evolution of the nodes depends on the hidden state, which is the table number in the Hellman method and the column number (this is also called color) in the rainbow method. It is pointed out in [14] that the cost of storage in the rainbow method is substantially higher than the Hellman + DP method.

Avoine et al. [11] studied the rainbow variant of TMTO attack to reduce the false alarms which may take 50% of the total attack time. By using "checkpoints", the probability for false alarms can be reduced and this reduces the overall complexity of the attack. The complexity improvement is less than a factor of 2. The main idea is to store intermediate "checkpoints" and to check the candidate against these, thus reducing the chance of a false alarm to survive the test.

3.3.5 Fiat-Noar Method

The analysis of the Hellman method assumes that the functions $f_1, f_2, \ldots, f_r$ are independent random function. It is not clear how to come up with such functions. Fiat and Naor [45] examined this requirement carefully and showed that there exist functions f which are polynomial time indistinguishable from a truly random function and for which the Hellman attack fails with overwhelming probability. They have given the following construction of f for which the Hellman method fails: consider a function f with the property that a certain set of $N^{1-\delta}$ domain points ($\delta < 1/3$) map to the same image. One can design a cryptographic scheme so that only $N - N^{1-\delta}$ of the keys induce a permutation and the other keys map all ciphertext values to zero. The Hellman attack fails for such an f. Though the existence of such a function f is of theoretical interest, it is of little practical consequence since functions obtained from crypto algorithms are extremely unlikely to be of such forms.

The indegree of a point y is the number of preimages for y under the function f. The Hellman method fails for the above f since the Hellman table may contain high indegree points. To avoid the high indegree points in the table, Fiat and Noar [45] proposed a method as follows.

The precomputation phase consists of the following two stages:

- Build a table A of high indegree points in the graph induced by f. This table is used to construct functions that bypass these points. Also, this allows a better cover of the domain by random chains. The table A is used to check if a point y has high indegree, i.e., if y has high indegree then there is high probability that the pair $(x, y = f(x))$ will appear in A. The preimage of y under f is only used to invert y itself but the main use of the table A is to avoid the high indegree points during the search. The table A is sorted in the increasing order of the end points.
- Use K-wise independent functions to substitute for the random functions assumed by the Hellman method, i.e., the output modification functions ψ_is are chosen to be K-wise independent functions. Construct the Hellman tables with appropriate choice of parameters so that there will be no high indegree points in the tables. Then store the (start point, end point) pairs in the tables with increasing order of the end points.

At the online phase, given $y = f(x)$ we perform the following to get the preimage x:

- check if $y \in A$. If found then an inverse is an x such that $\langle x, y \rangle$ is in A.
- Otherwise, perform a variant of the Hellman search technique in the Hellman tables.

Trade-off curve: Fiat-Noar method can invert any one-way function f in time T with the memory requirement M such that $TM^3 = N^3$ with the precomputation time $P = N$.

3.4 Time/Memory Trade-Off Cryptanalysis for Stream Ciphers

3.4.1 BG Attack

Babbage [12] and Golić [46] introduced a time/memory trade-off attack on a stream cipher, which is referred to as the BG attack (B for Babbage; G for Golić). The BG attack consists of two stages: precomputation and search. In the precomputation stage, we randomly choose M internal states $x_i \in S$, $i = 1, 2, \ldots, M$, compute $y_i = f(x_i)$ for $i = 1, 2, \ldots, M$ and store the pairs (x_i, y_i) into a table sorted in increasing order of the y values.

In the search stage, we are given a prefix of $D + s - 1$ generated bits, $c_1 c_2 c_3 \ldots\ldots c_{D+s-1}$. We generate D possible values $y_1, y_2, \ldots, y_D$ as follows: $y_1 = c_1 c_2 \ldots c_s$, $y_2 = c_2 c_3 \ldots c_{s+1}$, $y_3 = c_3 c_4 \ldots c_{s+2}$ and so on. Then we look up the table for each y_i; if at least one y_i is found in the table then its corresponding x_i is an internal state (or it might be a false alarm). The runtime is $T = D$, amount of memory required is M and preprocessing time is $P = M$. Now by the birthday paradox, we know that at least one of the y_i will be in the table with some significant probability if $DM = N$, i.e., if $TM = N$, which is the condition to get a constant success probability in the method. This method has been applied on the alleged A5 stream cipher by Golić in [46].

3.4.2 BS Attack

In 2000, Biryukov and Shamir [25] combined the Hellman method and the BG attack as follows: since data y_i, $i = 1, 2, \ldots, D$ can be viewed as unrelated random points, we can reduce the search space from N to N/D and we still get the same success probability. Hence the BS attack generates t/D Hellman tables with size $m \times t$ each in the precomputation stage. In the search stage, for each y_i, the Hellman search technique is applied to find the corresponding internal state. The attack is successful if at least one output value is found in any one of the tables. We take t/D ($t \geq D$) tables with size $m \times t$ each. Hence the amount of memory $M = \frac{mt}{D}$, runtime $T = \frac{t}{D} \times t \times D = t^2$. Then $TM^2D^2 = m^2t^4 = N^2$ (by the matrix stopping rule, i.e., $mt^2 = N$) which is the time/memory/data trade-off curve for this method. This relationship is valid if $t \geq D$ (otherwise the number of tables, i.e., $t/D < 1$) or $T = t^2 \geq D^2$. We refer to this attack as the BS attack. The number of table lookups for each y_i is $t \times \frac{t}{D} = \frac{t^2}{D}$, since for each table, t table lookups are required and there are t/D tables. Hence the total number of table lookups is $\frac{t^2}{D} \times D = t^2$.

3.4.3 Applying the DP Method in Stream Cipher Cryptanalysis

In the above methods, for stream cipher cryptanalysis we need to perform t table lookups for each table. To minimize the number of table lookups, the DP method is used in TMTO attacks on stream ciphers in [25]. The total number of possible DP states is 2^{s-p}. Let S be the probability that a random state will be a DP state. Then

$$S = \frac{\text{total number of possible DP state}}{N} = \frac{2^{s-p}}{2^s} = \frac{1}{2^p}.$$

BG attack with DP: In the precomputation stage, to store M DPs, we need to try $\frac{M}{S}$ random states, which increases the preprocessing time (search space) from $P = M$ to $P = \frac{M}{S}$. In the given data, we have $D \times S$ DPs. For each given DP, we perform the search technique in the actual attack. This reduces the runtime from $T = D$ to $T = DS$. By the birthday paradox, we know that there will be a collision (with high probability) between these DS DP states in the data and the stored M DP states in the tables if $M \times DS = NS$, since NS is the total number of possible DPs. Thus $TP = MD = N$ for $1 \leq T \leq D$. Hence, the trade-off curve $(TM = N, P = M, 1 \leq T \leq D)$ in the original BG method turns into two independent time/preprocessing $(TP = N)$ and memory/data $(MD = N)$ trade-off curves. This drastically changes the BG trade-off curve. If $N \geq 2^{80}$ then either M or D will be $> 2^{40}$ which is on the border line of feasibility. For higher values of N, we can not apply this method since either M or D become infeasible.

BS attack with DP: In the precomputation table stage, we select random start points and continue the chain until we reach a DP, so we do not need any trial and error (like in the BG method) to pick the random points. Hence the search space remains the same as N/D, the number of tables remains the same as t/D with same size $m \times t$ each, which makes the time/memory/data trade-off curve remain unchanged as $TM^2D^2 = N^2$ for $T \geq D^2$. But the number of table lookups reduces from t to 1 for a single table. So the number of table lookups required for t/D tables is t/D. Hence the total number table lookups for D data is $\frac{t}{D} \times D = t$ which makes a significant practical difference.

BSW Sampling: Biryukov, Shamir and Wagner [26] introduced a sampling technique to improve the BS attack. A state x is said to be a special state if it generates a DP output, i.e, if $f(x)$ is DP where f is the state to prefix mapping. A state x has a full name which is same as the original s-bit state and an output name of s bits. We associate a short name of $s - p$ bits to a special state x, which is used to define this special state by efficient enumeration procedures and a short output of $s - p$ bits, which is same as the output name $f(x)$ without the p leading zeros. Then the total space reduces to S_1 (say) where each point in S_1 can be viewed as either a short name or a short output and the size of S_1 is $N_1 = N \times S = 2^{s-p}$. We

Table 3.1: Trade-off curve for different methods (T is the runtime, M is the memory requirement, D is the number of available targets, P is the preprocessing time, S is the sampling resistance, t is the number of columns and N is the size of search space).

	Trade-Off curve	Number of table lookups
BG [12, 46]	$TM = N$ for $1 \leq T \leq D$	M
BG with DP [25]	$TP = MD = N$ for $1 \leq T \leq D$	M
BS [25]	$TM^2D^2 = N^2$ for $D^2 \leq T \leq N$	t^2
BS with DP [25]	$TM^2D^2 = N^2$ for $D^2 \leq T \leq N$	t
BSW [26]	$TM^2D^2 = N^2$ for $(DS)^2 \leq T \leq N$	tS

now define a function $f_s : S_1 \to S_1$ as follows: let $x_s \in S_1$ expand to full name x (say) which is a special point, then $f_s(x_s)$ is the last $(s-p)$ bit vector of $f(x)$ obtained by discarding the leading p zeros from $f(x)$ (since $f(x)$ is a DP). Thus $f_s(x_s)$ is a short output and inverting f_s is equivalent to inverting f restricted to special points. Then we apply the BS attack with the DP attack. Thus N and D will be replaced by NS and DS respectively and the trade-off curve become $T \times M^2 \times (DS)^2 = (NS)^2$, i.e., $TM^2D^2 = N^2$ for $T \geq (DS)^2$. Hence the curve remains unchanged, but the lower bound for T is reduced from D^2 (which is impractical for large value of D) to $(DS)^2$ (which is smaller) and also the number of table lookups is reduced from t to tS, since the search technique is applied only for the DS special points in the data. It is assumed that the available data contains at least one output as a special point, i.e., $DS \geq 1$. BSW sampling method is used in [26] to attack the stream cipher A5/1.

We compare the above methods in Table 3.1.

3.5 Applications of TMTO Algorithms

3.5.1 Application to Block Ciphers

A general problem of identifying suitable one-way functions in cryptographic algorithms with possible access to multiple data is addressed by Hong and Sarkar [57] where the possibility of TMTO application on various block cipher modes of operations are investigated. They showed that a suitable one-way function can be found for every mode of operations that they consider, to which chosen plaintext TMTO can be applied under appropriate conditions. Most interestingly, they showed how nontrivial multiple data TMTO can be applied to both the CBC and CFB modes of operations. However, Biham [17] pointed out that if CBC with

Table 3.2: TMTO attacks on UNIX password scheme.

Passwords attacked	State Size (bits)	Data	Time	Memory	Preprocessing
Alphanumeric	60	2^8	2^{34}	2^{34} (128 Gb)	2^{52}
Alphanumeric	60	2^{10}	2^{32}	2^{34}	2^{50}
Full keyboard	63	2^{10}	2^{36}	2^{35} (256 Gb)	2^{53}

a random and secret IV is used, a TMTO attack does not work unless the IV is considered to be a part of the secret. He also showed that the theoretical strength of ECB mode of block cipher cannot exceed the square root of the size of the key space.

3.5.2 Application to Stream Ciphers

Babbage [12], Golić [46], and Biryukov-Shamir [25] have considered applications of TMTO to the one-way function mapping internal state to a keystream segment. A possible countermeasure for resisting TMTO has been to use a state whose size is double that of the key size as suggested in [46].

Hong and Sarkar [57] revisited TMTO on stream ciphers and showed that a huge state size does not necessarily guarantee resistance to TMTO attack. Most stream ciphers use an initialization vector (IV) in addition to the secret key. They found that the function mapping (key, IV) to a keystream segment of suitable length is a candidate one-way function for TMTO application. If the IV length is less than the key length, then the cipher can be attacked using a TMTO algorithm whose online time is less than exhaustive search time. This will happen irrespective of the size of the internal state.

3.5.3 Application to Unix Password

The BS attack (described in 3.4.2) is used in [23] to analyze Unix password scheme. Suppose, the attacker has access to a file storing password hashes of a large organization ($D = 1000$ password hashes). The size of the unknown password is 56-bits and known salt is 12-bits. Suppose that the attacker knows that passwords are selected from a set of arbitrary 8-character alphanumeric passwords, including capital letters and two additional symbols like dot and comma which in total can be encoded in 48-bits. Thus together with a 12-bit salt the size of the search space is $N = 2^{60}$. Then the following TMTO attack is quite practical. Preprocessing is done once with $P = N/D = 2^{50}$ parallelizable Unix hash computations. A memory of $M = 2^{34}$, 8-byte entries (12+48 bits) which take one 128 Gbyte hard disk. This way we store 2^{34} start point and end point pairs. The attack time is then $T = 2^{32}$ Unix hash

evaluations which takes about an hour on a fast PC. The attack will recover one password from about every 1000 new password hashes supplied. Table 3.2 contains some of the TMTO attacks on UNIX password scheme.

3.6 Implementation of TMTO Attack

3.6.1 Software Implementation

In 2003, Oechslin [76] described the implementation of *rainbowcrack* which is a software implementation of rainbow method. Rainbowcrack can attack MS-Windows password hashes and crack 99.9% of all aphanumerical password hashes (out of 2^{37}) in 13.6 seconds using 1.4 GB memory. (The numerical figures are from [76], where exact platform is not mentioned.)

3.6.2 Hardware Implementation

In 1988, Amirazizi and Hellman [10] proposed *time/memory/processor trade-off* where more than one processors execute in parallel, sharing a large memory through a *switching/sorting* network. This requires $n \log n$ switching elements, n being both the number of processors and the blocks of memory. The emphasis of the work is to minimize the runtime of the cryptanalytic attacks in time/memory trade-off cryptanalysis by running the processors in parallel. The cost of the wires (number of wires required) is one of the dominating cost in the *switching/sorting* network. Amirazizi and Hellman [10] assumed that the cost of the wires is less than $n \log n$ and left this as an open problem for further study. Wiener [98] investigated the problem and proved that if an algorithm has a very high memory access rate then the wiring cost is the dominating cost for any *switching/sorting* network and showed this cost to be $\Theta(n^{\frac{3}{2}})$ to connect n processors with n memory blocks. It is shown that for the DP method where the memory access rate is low, the wiring cost is negligible. However this is not always the case for other methods (e.g. Hellman's method)

Quisquater and Standaert [81] provided a generic architecture based on their two previous works [79, 82]. They suggested a pipelined architecture for implementing a multi-round function f. This is built on Wiener's design [99] for exhaustive search attack on DES.

Mentens et al. [69] proposed a hardware architecture for key search based on the rainbow method. They have shown that an Virtex-4 FPGA implementation of the machine can recover an individual password within a few minutes. Their design targets Unix passwords of length 48 bits (out of 56 bits).

Chapter 4

TMTO With Multiple Data: Analysis and New Single Table Trade-offs

4.1 Introduction

In this chapter, we continue the analysis of the general multiple data TMTO started in Biryukov and Shamir (BS). The trade-offs of BG (Babbage and Golić) and BS are obtained as special cases. Our main contribution is to identify a new class of single table multiple data trade-offs which cannot be obtained either as BG or BS trade-off (some of these are shown in Table 4.1). In certain cases, these new trade-offs can provide more desirable parameters than the BG or the BS methods. We consider the analysis of the rainbow method of Oechslin and show that for multiple data, the TMTO curve of the rainbow method is inferior to the TMTO curve of the Hellman method. The material in this chapter is based on Sections 6 to 10 of our paper [24].

Table 4.1: Some new trade-offs (T is the runtime, M is the memory requirement, D is the available targets, P is the preprocessing time, N is the size of research space). Note that since all the trade-offs listed in the table are single table trade-offs, the number of table lookups in the Hellman + DP is going to be one in all the cases.

N	P	M	D	T
2^{64}	2^{48}	2^{40}	2^{16}	2^{24}
	2^{42}	2^{40}	2^{22}	2^{24}
2^{80}	2^{50}	2^{30}	2^{30}	2^{50}
	$2^{46.6}$	$2^{33.3}$	$2^{33.3}$	$2^{46.6}$
2^{100}	$2^{62.5}$	$2^{37.5}$	$2^{37.5}$	$2^{62.5}$
	$2^{58.3}$	$2^{41.6}$	$2^{41.6}$	$2^{58.3}$
* N	$N^{\frac{1+d}{3}}$	$N^{\frac{2-d}{3}}$	$N^{\frac{2-d}{3}}$	$N^{\frac{1+d}{3}}$

* Here d is a constant such that $\frac{1}{2} < d < 1$.

4.2 Hellman Attack

Suppose r tables each of dimension $m \times t$ are used and the online data consists of D points. Then from the Hellman attack we have the following relations.

$$\left.\begin{array}{lcll} T_f & = & r(t-1)D & (\#\ f \text{ invocations in the online phase}) \\ T_t & = & rtD & (\#\ \text{table lookups in the online phase}) \\ P & = & rmt & (\#\ f \text{ invocations in the precomputation phase}) \\ & = & \frac{N}{D} & (\text{coverage}) \\ M & = & rm & (\text{memory}) \\ mt^2 & \leq & N & (\text{birthday bound}) \\ \text{Success} & & & \\ \text{probability} & \approx & 1 - exp\left(-h(u)\frac{rmtD}{N}\right) & \end{array}\right\} \quad (4.1)$$

The number of memory access is same as the number of table lookups, i.e. T_t. Later in this chapter, we will derive the above success probability expression where $h(u) = \frac{1}{u}\int_0^u \frac{1-e^{-x}}{x}dx$ and $u = \frac{mt^2}{N}$.

If $t \gg 1$, we can assume $t - 1 \approx t$ and $T_f \approx rtD = T_t$. We will usually make this assumption except for the analysis of the BG attack where $t = 1$. Let γ be the ratio of the time required for performing one table lookup to the time required for one invocation of f, i.e.,

$$\gamma = \frac{\text{time for one table lookup}}{\text{time for one invocation of } f}. \quad (4.2)$$

We assume that one unit of time corresponds to one invocation of f (i.e., one unit of time is equal to the time required for completing one invocation of f). The time required for table lookups is then γrtD. Define $T = \max(T_f, \gamma T_t) = \gamma rtD$ when $\gamma \geq \frac{t-1}{t} \approx 1$. The parameter T is a measure of the time required during the online phase. The actual online time is proportional to $T_f + \gamma T_t$. However, this is only at most twice the value of T. Thus we will perform the analysis with T instead of $T_f + \gamma T_t$.

In modern technology, memory is organized in several levels. Depending on the memory organization γ can change. For an asymptotic analysis, we will assume that $\gamma = 1$ (and $T = T_t \approx T_f$), i.e., the cost of one invocation of f is equal to the cost of one table lookup. The value of γ need not actually be one. Even if it is a small constant (or a negligible fraction of N), we can assume it to be one and that will not affect the asymptotic analysis. On the other hand, [25] mentioned that γ may be as large as one million ($\approx 2^{20}$). If N is only moderately large (like 2^{64} for A5/1), then γ can be a significant proportion of N. In such a situation, we cannot assume $\gamma = 1$ and the cost of table lookup will dominate the total online cost. This case will be considered later.

Using (4.1) we can solve for r, m and t as follows.

$$\left.\begin{array}{lcll} t & = & \frac{N}{MD} \geq 1 & \text{(number of columns)} \\ m & = & \frac{N}{T} & \text{(number of rows)} \\ r & = & \frac{MT}{N} \geq 1 & \text{(number of tables)} \\ mt^2 & = & \frac{N^3}{TM^2D^2} \leq N & \text{(birthday bound)} \end{array}\right\} \tag{4.3}$$

Note that all three of r, m and t must be at least 1. Since $m = N/T$ and for a valid attack we must have $N > T$, the condition on m is trivially satisfied. The advantage of writing in the form of (4.3) is that given values for T, M and D satisfying the proper constraints, we can immediately design a table structure which achieves these values.

Let $D = N^a$ for some $0 \leq a < 1$. Since $PD = N$, we have $P = N^{1-a}$. The condition on r shows that $MT \geq N$. We write $MT = N^b$ for $b \geq 1$. Also let $M = N^c$. For a valid attack we must have $0 \leq c < 1$. Since $MT = N^b$ we have $T = N^{b-c}$ and again for a valid attack we must have $0 \leq b - c < 1$. The available online data D is a lower bound on M and T and hence we have $a \leq c, b - c$. Since the birthday bound tells us that $mt^2 \leq N$, we write $mt^2 = N^d$ for some d with $0 \leq d \leq 1$. Substituting in the last equation of (4.3), we obtain $2a + b + c + d = 3$. The condition on t shows that $MD \leq N$ which translates to $a + c \leq 1$. Thus any set of values for a, b, c and d which satisfies the following constraints constitute a valid attack.

$$\left.\begin{array}{ll} \textbf{C1:} & 2a + b + c + d = 3 \\ \textbf{C2:} & 0 \leq a < 1 \\ \textbf{C3:} & 0 \leq c, b - c < 1 \leq b \\ \textbf{C4:} & a + c \leq 1 \\ \textbf{C5:} & 0 \leq d \leq 1 \end{array}\right\} \tag{4.4}$$

The TMTO curve can be obtained as in the following relations.

$$\left.\begin{array}{lcl} TM^2D^2 & = & N^{3-d} \\ PD & = & N \\ MD \leq & N & \leq MT \\ M, D, T & < & N. \end{array}\right\} \tag{4.5}$$

Also we have the following values of r, m and t.

$$r = N^{b-1}, \quad m = N^{1-(b-c)}, \quad t = N^{1-a-c}. \tag{4.6}$$

Since $MT = N^b \geq N$ we have $r = 1$ if and only if $MT = N$. With $r = 1$ we have only one table and hence if there are more than one tables then MT is strictly greater than N.

BG Attack [12, 46]: In this case we have $r = t = 1$. This implies $T_f = 0$, i.e., the online phase does not require any invocation of f. The cost in the online phase is $T = T_t$ and we have $MD = N = MT$ and hence $T = D$, $M = N/D$. This corresponds to the conditions $a + c = 1, b = 1, d = 1 - a$.

BS Attack [25]: In [25], $r = t/D$ and $d = 1$ is used. Then $T = t^2$, $M = mt/D$ and hence $r = N^{-a+(b-c)/2}$. Since $r \geq 1$, we have the restriction $0 \leq 2a \leq b - c$ (i.e., $1 \leq D^2 \leq T$) in addition to (4.4).

The conditions $d = 1$ and $r = t/D$ are related (e.g., if $r = 1$ then $t = D$ and $T = t^2 = D^2$). In the following analysis we will proceed without these two conditions. Later we show the situation under which making these two assumptions is useful.

Condition $P = T$: Since both P and T represent time, the case $P = T$ puts equal emphasis on both the offline and the online times. This can be of theoretical interest and hence we briefly analyze it. (A reviewer of the thesis pointed out that $P = 10T$ to $100T$ is more practical.) The condition $P = T$ implies $P = N^{1-a} = T = N^{b-c}$ and so $m = N^{1-(b-c)} = N^a = D$. (On the other hand, $P = M$ is possible only if $t = 1$.) Since $PD = N$ we have $T = N/D$ and so the curve becomes $M^2D = N^{2-d}$. If $P = T$ then $r = M/D$. If further $M = D$ then $M = D = N^{(2-d)/3}$ and $P = T = N^{(1+d)/3}$.

Proposition 5 *If $P = T$ and $M = D$ in (4.5) then $M = D = N^{(2-d)/3}$ and $P = T = N^{(1+d)/3}$. Further, $r = 1$, i.e., exactly one table is required.*

Proposition 5 gives us a nice way to control the trade-off between time and data/memory requirement by varying d. Choosing $d = 1$ corresponds to $(P, D, M, T) = (N^{2/3}, N^{1/3}, N^{1/3}, N^{2/3})$ and has been observed in [25]. Choosing $d = 1/2$ corresponds to,
$(P, D, M, T) = (N^{1/2}, N^{1/2}, N^{1/2}, N^{1/2})$ which is the square root birthday (BG) attack.

Condition $T = M$: The condition $T = M$ was considered by Hellman [56] and we consider this to be of theoretical interest. Then $c = b - c$ and so $c = b/2$. Condition **C1** becomes $2a + 3c + d = 3$ and we have

$$\frac{b}{2} = c = 1 - \frac{2a + d}{3}. \tag{4.7}$$

Using $a + c \leq 1$ we obtain $a \leq d$. Also since $b \geq 1$, we have $c = b/2 \geq 1/2$. This along with (4.7) gives $d \leq 3/2 - 2a$. Since we already know $d \leq 1$ we obtain

$$a \leq d \leq \min(1, \frac{3}{2} - 2a). \tag{4.8}$$

Thus any non-negative solution in a and d to (4.8) gives a valid attack with $T = M = N^c$.

We are interested in minimizing the value of c. From (4.7) we see that the value of c is minimized by maximizing the value of d. In fact, using (4.8) we can choose $d = 1$ as long as $1 \leq \frac{3}{2} - 2a$, i.e., $2 - (1/2a) \leq 0$ or $a \leq 1/4$. Thus for $a \leq 1/4$ we obtain $T = M = N^{b/2} = N^{(2-2a)/3}$.

In the case $3/2 - 2a \leq 1$ we have $a \leq d \leq 3/2 - 2a$. For the gap to be non-empty we must have $a \leq 1/2$. For minimizing c, we use the upper bound, i.e., $d = 3/2 - 2a \leq 1$. Thus for $1/4 \leq a \leq 1/2$ we have $c = 1/2$ and $T = M = N^{1/2}$. Finally, we obtain the following result.

Theorem 6 *If $T = M$, then $D \leq N^{1/2}$ and the following conditions hold.*

1. $N^{1/2} \leq T = M = N^{(2-2a)/3} \leq N^{2/3}$, *for* $1/4 \geq a \geq 0$.
2. $T = M = N^{1/2}$, *for* $1/4 \leq a \leq 1/2$.

For the first case we have, $(a, b, c, d) = (a, 2(2-2a)/3, (2-2a)/3, 1)$ *and for the second case we have* $(a, b, c, d) = (a, 1, 1/2, 3/2 - 2a)$. *The corresponding values of* (r, m, t) *are* $(N^{(1-4a)/3}, N^{(1+2a)/3}, N^{(1-a)/3})$ *and* $(1, N^{1/2}, N^{1/2-a})$ *respectively.*

In the second case of Theorem 6, exactly one table is required. However, it is not the BG attack since the number of columns can be more than one. Also we have $T \leq P \leq N$. The situation with $T < P < N$ is interesting since the precomputation time is less than exhaustive search. Even though P is more than T since it is an offline activity, we might wish to spend more time in the precomputation part than in the online attack.

4.2.1 Distinguished Point Method

We now consider the case where $\gamma \gg 1$. In this case, a direct application of the Hellman method leads to $T = \gamma rtD$, i.e., the time required for table lookups dominate the online

time. It is useful to consider the distinguished point method of Rivest to reduce the number of table lookups. See section 3.3.2 for a description of the DP method.

Using the distinguished point method results in reducing the number of table lookups from rtD to rD, i.e., one table lookup per table per data. Then $T_t = rD = N^{a+b-1}$. (Note $T_t = N^a = D$, i.e., only one table lookup is required per data item if and only if $b = 1 = r$, i.e., $MT = N$.)

The total cost of table lookups is γrD whereas the cost of invoking the one-way function is rtD. In this case, the ratio of the two costs is γ/t. If $t \geq \gamma$, then the ratio is at most one. Hence we can again ignore the cost of table lookup and perform the analysis by considering simply the cost of invoking the one-way function. The actual runtime will be at most twice the runtime obtained by such an analysis.

Suppose $t < \gamma$. Then the analysis performed above does not hold. We now investigate the situation under which $t < \gamma$ holds. This certainly holds for $t = 1$ (the BG attack), but in the BG attack the entire online computation consists of table lookups and hence the general analysis is not required. Recall that $t = N^{1-(a+c)} = 2^{s(1-(a+c))}$, $D = N^a$ and $M = N^c$. Suppose $\gamma = 2^e$. Then $t \geq \gamma$ if and only if $a + c \leq 1 - (e/s)$. The value of e is a constant whereas s increases. Hence $(1 - e/s) \rightarrow 1$ as s grows. Thus we can have $a + c > 1 - e/s$ only for small values of s. If $e \approx 20$ (mentioned in [25]) then $1 - e/s \geq 2/3$ for $s \geq 64$.

Consider $a = c = 1/3$ as in the solution $(a, b, c, d) = (1/3, 1, 1/3, 1)$ corresponding to $P = T = N^{2/3}$, $M = D = N^{1/3}$, $r = 1$ of [25]. If $s \geq 64$ then $a + c = 2/3 \leq 1 - e/s$ and the time analysis assuming $T = rtD = tD$ holds. On the other hand, for the solution $(a, b, c, d) = (3/8, 1, 3/8, 7/8)$ corresponding to $P = T = N^{5/8}$, $M = D = N^{3/8}$, $r = 1$ considered in Section 4.3, we have $a + c = 3/4$. For $s = 64$, $a + c > 1 - e/s$ and we have to assume $T = \gamma rD = \gamma D$, whereas for $s = 100$, $a + c \leq 1 - e/s$ and we can assume $T = rtD = tD$. Thus for relatively small s we should solve (4.4) with the constraint $a + c \leq 1 - e/s$ instead of $a + c \leq 1$. This disallows some of the otherwise possible trade-offs.

There is another issue that needs to be considered. We have to ensure that t is large enough to ensure the occurrence of a DP in a chain. Let 2^{-p} be the probability of a point being a DP. Hence we can expect one DP in a random collection of 2^p points. Thus if $t \geq 2^p$ we can expect a DP in a chain of length t. This implies $p \leq \log_2 t$. Any attempt to design the tables with $t < 2^p$ will mean that several trials will be required to obtain a chain terminating in a DP. This will increase the precomputation time. In fact, [25] has shown that use of the DP method in the BG attack divides into two different trade-offs leading to unrealistic requirements on data and memory.

Using (4.6) we have $\frac{p}{s} \leq 1-(a+c)$. This leads to the condition $a+c \leq 1-\frac{p}{s}$ ($MD \leq N^{1-\frac{p}{s}}$) instead of the condition $a + c \leq 1$ (resp. $MD \leq N$) in (4.4) (resp. (4.5)). For small s, this condition has to be combined with $a + c \leq 1 - e/s$ and we should solve (4.4) with the

constraint $a + c \leq 1 - \frac{1}{s}\max(p, e)$ instead of the constraint $a + c \leq 1$. This puts further restrictions on otherwise allowed trade-offs.

4.3 Single Table Attack

The case $N = 2^{100}$ has been considered in [25]. It has been mentioned in [25] that the Hellman attack with $D = 1$, $T = M = N^{2/3} = 2^{66}$ requires unrealistic amount of disk space and the BG attack with $T = D = N^{2/3} = 2^{66}$, $M = N^{1/3} = 2^{33}$ requires unrealistic amount of data. (Note $T = M = D = N^{1/2} = 2^{50}$ also gives a BG attack but the data requirement is still unrealistic.) Further, [25] mentioned $P = T = 2^{66}$ and $D = M = 2^{33}$ to be a (barely) feasible attack. This corresponds to the parameters $(a, b, c, d) = (1/3, 1, 1/3, 1)$ and $(r, m, t) = (1, N^{1/3}, N^{1/3})$.

From Proposition 5 if we choose $d = 7/8$, then we obtain $M = D = N^{3/8} = 2^{37.5}$ and $P = T = N^{5/8} = 2^{62.5}$. The corresponding parameters are $(a, b, c, d) = (3/8, 1, 3/8, 7/8)$ and $(r, m, t) = (1, N^{3/8}, N^{1/4})$. This brings down the attack time while keeping the data and memory within feasible limits. Since $t > 1$, this cannot be obtained from the BG attack. Further, choosing $d = 7/8$ and $D^2 > T$ ensures that this attack cannot also be obtained from the BS attack. We would like to point out that [25] mentioned that choosing $d < 1$ is "wasteful". The above example shows that this is not necessarily the case and choosing $d < 1$ can lead to more flexible trade-offs. We show below the condition under which choosing $d < 1$ is indeed "wasteful".

The choice of the parameter $r = t/D$ is motivated in [25] by mentioning that this reduces the number of table lookups. The number of table lookups in the first case ($T = D = N^{2/3}, M = N^{1/3}$) is $rtD = tD = N^{2/3}$ whereas in the second case ($M = D = N^{3/8}, P = T = N^{5/8}$), it is $rtD = tD = N^{5/8}$. Thus the above example shows that the condition $r = t/D$ is not necessary for reducing the number of table lookups.

As mentioned earlier, we have one table (i.e., $r = 1$) if and only if $MT = N$. The reason for moving to more than one table is when $mt^2 > N$ and we begin to have more and more repetitions within a table.

Proposition 7 *There is a solution to (4.5) with $r = 1 = b$ (and hence $MT = N = PD$) if and only if $2a + c \geq 1$.*

Proof : Suppose $r = 1$. Then $b = 1$ and $2a + c + d = 2$. So $d = 2 - (2a + c)$. Since $d \leq 1$, we have $2a + c \geq 1$.

On the other hand, assume that $2a + c \geq 1$. Choose $b = 1$ and set $d = 2 - (2a + c) \leq 1$. This choice satisfies the conditions of (4.5). Further, since $b = 1$ we have $r = 1$. ■

Suppose $2a + c < 1$. Then $b + d > 2$ and $b > 2 - d$. Since $MT = N^b$, we would like to minimize b and hence we choose $d = 1$. We can now modify the suggestion of [25] and say that it is "wasteful" to choose $mt^2 < N$ *if there is more than one table.* Since $b > 1$, we have $2a + c < 1 < b$ and hence $2a < b - c$ which gives $D^2 < T$ and we are back to the situation described in [25].

Thus the analysis of [25] actually applies to the situation where the data is small enough to require more than one tables. On the other hand, for the case of one table, the restrictions of [25] are not required and removing these restrictions provide more flexible trade-offs. We would like to point out that there are interesting situations where a single table can be used. Apart from the examples $D = M = N^{1/3}$ and $D = M = N^{3/8}$ already considered, other possible examples are ($D = N^{0.3}$, $M = N^{0.4}$), ($D = N^{0.25}$, $M = N^{0.5}$) etc.

Going back to the example of $N = 2^{100}$, both $(P, D, M, T) = (N^{2/3}, N^{1/3}, N^{1/3}, N^{2/3})$ of [25] and $(P, D, M, T) = (N^{5/8}, N^{3/8}, N^{3/8}, N^{5/8})$ described above have $r = 1$. As mentioned above, the second one is better with respect to the number of table lookups. In conclusion, there are reasonable choices of data and memory requirements which lead to a single table. In such situations, the trade-off in [25] is not the only possible one. Other (and perhaps better) trade-offs can be obtained following the approach described here. We highlight some of the other interesting single table trade-offs that can be obtained.

Condition $P = T = N^{(1+d)/3}$, $M = D = N^{(2-d)/3}$: From Proposition 5, we have $r = 1$, i.e., all trade-offs attaining this condition use a single table. In the plausible situation, $M = D \leq P = T$, we have $1/2 \leq d \leq 1$. The case $d = 1$ can be obtained from the BS analysis. In the BG analysis, we have $d = 1 - a$. Since $a - (2 - d)/3$, this condition leads to $d = 1/2$. Thus the range $1/2 < d < 1$ for which the condition $P = T = N^{(1+d)/3}$, $M = D = N^{(2-d)/3}$ can be attained was not known earlier.

Condition $M = T$: In the second case of Theorem 6, we have $r = 1$ and $M = T = N^{1/2}$. The allowed range of a for this case is $1/4 \leq a \leq 1/2$. The case $a = 1/4$ can be obtained from the BS analysis and the case $a = 1/2$ can be obtained from the BG analysis. However, the range $1/4 < a < 1/2$ for which $T = M = N^{1/2}$ can be attained, cannot be obtained from either the BG or the BS analysis and provide previously unknown trade-offs. The advantage is that the data can be increased (thus lowering offline time) without increasing either time or memory.

Small N: Consider $N = 2^{64}$ as in A5/1. It is mentioned in [26] that $M \approx 2^{35}$ and $D \approx 2^{22}$ are reasonable choices. We consider two trade-offs corresponding to the second case of Theorem 6.

Trade-Off 1: $(P, D, M, T) = (2^{46}, 2^{18}, 2^{32}, 2^{32})$**:** The table parameters are $(r, m, t) = (1, 2^{32}, 2^{14})$.

Trade-Off 2: $(P, D, M, T) = (2^{42}, 2^{22}, 2^{32}, 2^{32})$**:** The table parameters are $(r, m, t) = (1, 2^{32}, 2^{10})$.

None of the above two trade-offs are obtainable as BG trade-offs, since in both cases $t > 1$. Also neither can be considered to be BS trade-offs since $D^2 > T$. For both trade-offs, the data and memory are within reasonable limits and the online times are the same. The offline time is lower for the second trade-off and is within doable limits (especially as an offline one-time activity), while for the first attack it is probably just outside the doable limit.

The total cost of online table lookup for both the attacks is γtD. Since the value of γ is a significant proportion of N the cost of table lookup dominates the online cost. Use of the DP method reduces the total cost of table lookups to γD. If γ is around 2^{20} as mentioned in [25], we have the table lookup costs to be 2^{38} and 2^{42} respectively. This pushes up the online cost for both the attacks to make them less of a threat. On the other hand, if γ can be brought down to around 2^{10} by the deployment of special purpose high speed memory, then the table lookup costs come down to 2^{28} and 2^{32} respectively. This will make both the attacks serious threats. We note that with $\gamma \approx 2^{20}$, the attack of [26] remains the most efficient one.

4.4 Rainbow Attack

In this section, we analyze the rainbow method in the presence of multiple data. In the rainbow attack, we use a table of size $m \times t$ and suppose there are D online data points. Then the total number of invocations of the one-way function is $t^2D/2$ while the cost of the table lookups is tD. Again, we will ignore the factor of two in the runtime since it does not affect the asymptotic analysis. Then the total number of invocations of f is t^2D and the total number of table lookups is tD. Also we have $mt = N/D$.

If we assume $\gamma \approx 1$, then the cost of invoking f dominates the online cost and we have $M = m$ and $T = t^2D$. Assume $D = N^a$ and $M = N^c$ as in the case of the Hellman analysis. Then since $mt = N/D = N^{1-a}$ we have $t = N^{1-a-c}$ and $T = t^2D = N^{2-a-2c}$. Also since $t \geq 1$, we must have $a + c \leq 1$. The TMTO curve for the rainbow method in the presence of multiple data is $TM^2D = N^2$ which is inferior to the Hellman TMTO curve when $D > 1$.

We now compare the rainbow parameters $(P, D, M, T) = (N^{1-a}, N^a, N^c, N^{2-a-2c})$ with the Hellman parameters for same data and memory. For multiple table Hellman, we choose

$d = 1$ and hence the corresponding Hellman parameters are
$(P, D, M, T) = (N^{1-a}, N^a, N^c, N^{2-2a-2c})$. If $a > 0$ i.e., if multiple data is available then clearly the Hellman time is less than the rainbow time. Thus, we conclude that in the presence of multiple data, the Hellman attack is in general better than the rainbow attack.

4.5 Increasing the Coverage Space

The success probability of the Hellman method is constant. It has been observed [58] that this value is around 60%. To increase the success probability, one can increase the coverage space of all the tables. The tables together cover a total of rmt points. We assume that $rmt = \lambda(N/D)$ for some $\lambda \geq 1$. By choosing $\lambda > 1$, it is possible to increase the success probability. Kim and Matsumoto [58] have described this technique for the basic Hellman attack with $D = 1$. Below we show that essentially the same technique also works for $D > 1$.

Let $y_1, \ldots, y_D$ be the D data points and let $x_1, \ldots, x_D$ be such that $f(x_i) = y_i$ where x_is are selected uniformly at random. We define the success probability $\mathsf{P_{Succ}}$ to be the probability that at least one of x_i is in the tables. Let P_1 be the probability that a random point is covered by the tables. Then $\mathsf{P_{Succ}} = 1 - (1 - P_1)^D$. To find P_1, we proceed as follows. Let $\mathsf{PS_{single}}$ be the probability that the randomly chosen point is in a single table. Then $P_1 = 1 - (1 - \mathsf{PS_{single}})^r$ and $\mathsf{P_{Succ}} = 1 - (1 - P_1)^D = 1 - (1 - \mathsf{PS_{single}})^{rD}$. In [56], Hellman provided the expression (see Section 3.3.1) $\mathsf{PS_{single}} \geq \frac{1}{N} \sum_{i=1}^{m} \sum_{j=1}^{t} \left(1 - \frac{it}{N}\right)^j$. Later Kim and Matsumoto [58] made the following simplification, $\mathsf{PS_{single}} \geq h(u)\frac{mt}{N}$ where $h(u) = \frac{1}{u}\int_0^u \frac{1-e^{-x}}{x}dx$ and $u = \frac{mt^2}{N}$. This gives, $\mathsf{P_{Succ}} \geq 1 - \mathsf{e}^{(-h(u)\times\lambda)}$ where $\lambda = \frac{rmtD}{N}$. Now $h(u) = \frac{1}{u}\int_0^u \frac{1-e^{-x}}{x}dx < 1$ for all $u > 0$. This implies that the success probability increases with the value of λ.

Chapter 5

Application of LFSRs in Time/Memory Trade-Off Cryptanalysis

5.1 Introduction

TMTO attacks require the generation of a sequence of functions which are obtained as minor modifications of a one-way function to be inverted. In this chapter, we carefully examine the requirements for such function generation. A counter-based method is used to generate the functions for the rainbow method. We show that there are functions for which the counter method fails. This is similar to the example given by Fiat and Naor [45] for the Hellman TMTO. Our main contribution is to suggest the use of LFSR sequences for function generation to be used in the rainbow TMTO. Properties of LFSR sequences such as long period, pseudo-randomness properties and efficient forward and backward generation make such sequences useful for the intended application. This part is based on material from [72].

We also consider the problem of efficiently generating sequences in hardware for use in parallel start point generation. The conventional method of doing this is to use a counter. We show that sequences generated by linear feedback shift registers (LFSRs) can be tailored to suit the parallel start points generation. For hardware implementation, this reduces both time and chip area. As a result, we are able to provide an efficient strategy for generating start points in TMTO attacks. This part is based on Sections 2 and 5 of [73].

One of the reviewers of this thesis has pointed out that LFSR is better than counter in absolute terms. But LFSR or counter will be implemented as a part of a bigger circuit which could be an AES block. The AES block will take up much more area in the chip compared

to an LFSR or a counter. So in this overall context, the savings may not be much when using LFSR instead of counter. On the other hand, we note that even a small reduction in the size of a chip can be of use when such chips are mass produced.

TMTO attacks require the generation of a sequence of functions $f_1, f_2, \ldots$ where f_is are obtained from f by a minor modification such as permuting the output bits of f. A crucial point in TMTO attack is to obtain the different functions f_i such that they can be assumed to be independent random functions.

In case of the rainbow tables, a counter based method is used to generate the rainbow chains. The function $f_i(x)$ is defined as $f(x) \oplus i$. If several rainbow tables are used, then the counter method is used in the following manner: if table one uses index 1 to 1000 (say), then table two uses index 1001 to 2000, and so on.

We carefully examine the different requirements for defining the functions f_i. We show that the counter method does not ensure uniform modification of the output and an adaptation of Fiat-Naor [45] counter example for the Hellman method also works for rainbow with counter method .

This leads us to the question of obtaining a method to define the f_i's such that the output modification is uniform. Our main contribution is to show that sequences produced by linear feedback shift registers (LFSRs) are a natural choice for such an application. LFSR sequences are very efficient to generate in the forward and backward directions, they satisfy certain nice pseudo-randomness properties, it is quite easy to generate very long non-repeating sequences of bit vectors. All these properties make LFSR sequences very suitable for defining the functions required in rainbow chains for one or more tables.

Details of an LFSR based multiple table rainbow method are presented and analyzed. It turns out that for the same precomputation time, the success probability of multiple table rainbow method is higher than that of Hellman method or the single table rainbow method. On the other hand, the runtime of the multiple table method is slightly higher. We show that a Kim-Matsumoto style parametrization is possible for the rainbow method and yields a higher success probability than the single table rainbow method without changing the runtime or the memory requirement.

For the parallel implementation of TMTO attack, we require the generation of parallel independent (pseudo)random sequences of s-bit values in the precomputation phase. Implementations of this use a counter to generate the required sequences. While this is intuitively simple, it is not the best possible option for hardware implementation. We show how LFSR sequences can be tailored for use in the parallel start point generation.

5.2 LFSR Preliminaries

A linear feedback shift register (LFSR) [62, 68] of length l consists of l stages $0, 1, 2, \ldots, l-1$, each capable of storing one bit. An l-bit LFSR is denoted by $(l, p(x))$, where $p(x) = 1 \oplus c_1 x \oplus \cdots \oplus c_{l-1}x^{l-1} \oplus x^l$ is called the connection polynomial [68]. LFSRs can produce sequences having large periods. If the initial content of stage i is $a_i \in \{0, 1\}$, for $0 \leq i \leq l-1$, then $(a_{l-1}, a_{l-2}, \ldots, a_0)$ is called the initial internal state of the LFSR. Let at time $t \geq 0$ the content of the stage i be $a_i^t \in \{0, 1\}$, for $0 \leq i \leq l-1$, then the internal state of the LFSR at time t is $(a_{l-1}^t, a_{l-2}^t, \ldots, a_0^t)$. Let $\{X_t{=}(a_{l-1}^t, \ldots, a_1^t, a_0^t)\}$ for $t \geq 0$, be a sequence of l-bit vectors. If $p(x)$ is a primitive polynomial, then each of the $2^l - 1$ non-zero initial states of the LFSR $(l, p(x))$ produces an output sequence with maximum possible period $2^l - 1$.

Let us consider the k^{th} and $(k+1)^{th}$ terms of the sequence, i.e., $X_k = (a_{l-1}^k, \ldots, a_1^k, a_0^k)$ and $X_{k+1} = (a_{l-1}^{k+1}, \ldots, a_1^{k+1}, a_0^{k+1})$ respectively, where

$$\left.\begin{array}{rcl} a_{l-1}^{k+1} & = & c_{l-1}a_{l-1}^k \oplus c_{l-2}a_{l-2}^k \oplus \cdots \oplus c_1 a_1^k \oplus a_0^k; \\ a_{l-2}^{k+1} & = & a_{l-1}^k; a_{l-3}^{k+1} = a_{l-2}^k; \ldots; a_0^{k+1} = a_1^k. \end{array}\right\} \tag{5.1}$$

From (5.1) we get

$$\left.\begin{array}{rcl} a_0^k & = & a_{l-1}^{k+1} \oplus c_{l-1}a_{l-2}^{k+1} \oplus c_{l-2}a_{l-3}^{k+1} \oplus \cdots \oplus c_1 a_0^{k+1}; \\ a_1^k & = & a_0^{k+1}; a_2^k = a_1^{k+1}; \ldots; a_{l-1}^k = a_{l-2}^{k+1}. \end{array}\right\} \tag{5.2}$$

Equations (5.1) and (5.2) show that forward and backward generation of LFSR sequences require at most l XOR operations on bits and can be done very fast in hardware and software (see for example [33]).

In this thesis we consider binary LFSRs. We note, however, that the techniques described in this thesis also hold for LFSRs over larger alphabets and for other linear sequence generators like cellular automata.

Below we highlight some features of LFSRs which are relevant to our work. See [62, 68] for more details and theory of LFSR sequences including non-binary LFSRs. Additionally, we would like to point out that even though we consider only LFSRs in this paper, our ideas carry over in a straightforward manner to other linear finite state machines like cellular automata.

Maximal length LFSR: It is well known that if $p(x)$ is a primitive polynomial, then for any non-zero s-bit vector S_0, the sequence $S_0, S_1, S_2, \ldots, S_{2^s-2}$ consists of all the $2^s - 1$ non-zero s-bit vectors. An LFSR which has this property is called a maximal length LFSR. The number of primitive polynomials of degree s over $GF(2)$ is given by the expression $\phi(2^s - 1)/s$, where $\phi(i)$ is the Euler totient function and is defined to be the number of

positive integers less than i and co-prime to i. The expression $\phi(2^s-1)/s$ is almost as large as 2^s and hence there are a large number of maximal length LFSRs of a certain degree. Further, maximal length LFSR sequences satisfy certain well defined pseudo-randomness properties and hence such sequences are used in generating test vectors.

Matrix representation: There is another way to view an LFSR sequence, which will be useful to us later. The next state S_{t+1} is obtained from the previous state S_t by a linear transformation and hence we can write $S_{t+1} = S_t M$, where M is an $s \times s$ matrix whose characteristic polynomial is $p(x)$. Extending this we can write $S_t = S_0 M^t$. Thus knowing M^t we can directly jump from S_0 to S_t without going through the intermediate states. For any fixed value of $t < 2^s - 1$, computing the matrix exponentiation M^t can be done using the usual square and multiply method and requires at most $2 \log t \leq 2s$ matrix multiplications. Appropriate addition chain heuristics can speed up the computation. Later we will apply this idea for parallel generation of subsequences of the sequence $S_0, S_1, \ldots, S_{2^s-2}$.

Implementation: Implementing an LFSR in hardware is particularly efficient. Such an implementation requires s flip-flops and $\mathsf{wt}(p(x)) - 1$ 2-input XOR gates, where $\mathsf{wt}(p(x))$ is the number of non-zero coefficients in $p(x)$. With this hardware cost, the next s-bit state is obtained in one clock. For maximal length LFSR, one requires $p(x)$ to be primitive. It is usually possible to choose $p(x)$ to be of very low weight, either a trinomial or a pentanomial. Thus, an s-bit maximal length LFSR provides a fast and low cost hardware based method for generating the set of all non-zero s-bit vectors. Software generation of an LFSR sequence is in general not as efficient as in hardware. On a machine which supports w-bit words, the next s-bit state of an LFSR can be obtained using $(\mathsf{wt}(p(x)) - 1)s/w$ XOR operations (see [33]).

Pseudo-randomness properties: The sequences generated by a maximal length LFSR satisfy the following properties. The number of 1's differs from the number of 0's by at most 1. In every period, half the runs have length 1, $\frac{1}{4}$th have length 2, $\frac{1}{8}$th have length 3, etc., as long as the number of runs so indicated exceeds 1. Moreover, for each of these lengths, there are (almost) equally many runs of 0's and of 1's. Frequency test, serial test, poker test, runs test, auto-correlation test are some basic tests to measure pseudo-randomness of a sequence. The sequences generated by a maximal length LFSR satisfy all these tests. For other statistical properties of LFSR based sequences, one can see [68].

5.2.1 Possible Advantages of LFSRs over Counters

Implementing a counter which can count from 0 upto $2^s - 1$ requires an s-bit register and an adder. At a top level, an adder circuit is more complicated than an LFSR, since carry propagation has to be handled. However, use of different techniques can provide efficient adder designs. In contrast, for LFSR sequences, apart from the s-bit register, we require only $\mathsf{wt}(p(x)) - 1$ 2-input XOR gates. The main cost of implementing an LFSR is the register and the interconnections. The number of XOR gates can usually be taken to be either two or four and can be assumed to be less than ten for all values of s. Thus, the cost of implementing an LFSR scales linearly with the value of s. Additionally, for some applications, the requirement is to generate a pseudo-random sequence of non-negative integers. In such cases, one cannot use a counter.

5.3 Function Generation

In this section, we first consider the requirements on the functions f_i's. The definition of f_i has been suggested by Hellman to be $f_i(x) = \psi_i(f(x))$. We will call this to be the output modification approach. One can similarly consider $f_i(x) = f(\psi_i(x))$, or the input modification approach.

We first consider the case of input modification and argue that this is actually the same as output modification. Consider the rainbow method and suppose $f_i(x)$ is defined as $f_i(x) = f(\psi_i(x))$. Consider the rainbow chain

$$(f_{t-1} \circ f_{t-2} \circ \cdots \circ f_1 \circ f_0)(x_0)$$

where x_0 is a start point and $x_i = f_{i-1}(x_{i-1})$ for $i \geq 1$. Expanding the above sequence, we can write

$$(f \circ \psi_{t-1} \circ f \circ \psi_{t-2} \circ \cdots \circ f \circ \psi_1 \circ f \circ \psi_0)(x_0).$$

Now for $1 \leq i \leq t-1$, if we define $g_i(x) = \psi_i(f(x))$, then we get the rainbow chain $x'_0, \ldots, x'_{t-1}$ where $x'_0 = \psi_0(x_0)$ and $x'_i = g_i(x'_{i-1})$. This gives a rainbow chain of output modified form of length one less than the original chain. Also, note that x_0 is chosen to be a random point and hence it does not matter whether we start from x_0 or from $\psi_0(x_0)$. This shows that we can convert a chain of input modified form into a chain of output modified form. A similar conversion will also convert a chain of output modified form into a chain of input modified form. Further, the technique also works for the original Hellman method.

The literature considers only output modification. To the best of our knowledge, the above argument regarding input modification does not appear in the literature. In view of this argument, like previous works, we will consider only output modification.

5.3.1 Invertibility

Consider the search technique of the rainbow method. From equation (3.2) of Section 3.3.3, we assume $f_j(x) = \psi_j(y)$ and infer that $f(x) = y$. If ψ_j is invertible, then using $f_j(x) = \psi_j(f(x)) = \psi_j(y)$, we have $f(x) = y$ and x is a preimage of y. If ψ_j is not invertible, then the relation might not give a preimage of y, leading to a false alarm. The condition ψ_j being invertible ensures that there are no false alarms due to the use of ψ_j. (Note that there may be false alarms due to f itself or due to the modification to f to make the domain and range the same.) A similar argument shows that ψ_j's used in the Hellman method should also be invertible.

5.3.2 Efficient Function Generation

To apply the function f_i we need to apply f and the function ψ_i. For this we need a description of the function ψ_i. One approach is to store the description of all the t functions $\psi_0, \ldots, \psi_{t-1}$. This requires an additional storage space of order t. Since $t = N^{1/3}$ in both the Hellman and the rainbow method, this storage amount can be substantial. One way to avoid this storage is to generate the functions "on the fly". Thus, we need an efficient on the fly method to generate the functions ψ_is.

5.3.3 Long Period

Consider the on-the-fly method discussed above. This means that we should actually be capable of generating a sequence of bit vectors and use these to define the functions ψ_is. Since we do not want repetition of the functions, the sequence must consist of distinct bit vectors. In other words, it must be possible to generate a sequence of bit vectors with period long enough to ensure that all the ψ_is are distinct.

5.3.4 Uniform Modification of Output

The rainbow method uses a sequence $f_0, f_1, \ldots, f_{t-1}$ of functions. These are generated using a counter. We argue that the counter method also suffers from a problem similar to the one described by Fiat and Naor for the Hellman method. Given a function f, the rainbow method constructs the modification functions f_i by defining $f_i(x) = f(x) \oplus i$. Since $i \leq t$, this modifies at most the $\log t$ least significant bits of $f(x)$. Now one can construct a function f as follows: $f : \{0,1\}^s \to \{0,1\}^s$ with the property that for any $x \in \{0,1\}^s$, if $\mathsf{First}_{n_1}(x) = (0, 0, \ldots, 0)$ (s_1 bits) then $f(x) = (0, 0, \ldots, 0)$ (s bits). Let S_1 be the set of all s-bit vectors whose most significant s_1 bits are zero. Then the size of S_1 is $N_1 = 2^{s-s_1}$. We choose $s - \log t = s_1 < \frac{s}{3}$.

Considering such a function, we may construct a cryptographic scheme so that $N - N_1$ of the keys induce a permutation and other keys map all ciphertext values to zero. For a rainbow chain

$$x_0 \xrightarrow{f_0} x_1 \xrightarrow{f_1} x_2 \rightarrow \ldots \rightarrow x_{t-1} \xrightarrow{f_{t-1}} x_t,$$

if any x_i is in S_1, then $x_{i+1}, x_{i+2}, \ldots$ up to x_{N_1} (if $N_1 \leq t$) are zeros. This will generate a huge number of zeros inside a rainbow table, resulting in the failure of the rainbow method in this case.

5.3.5 Pseudo-randomness

One way to avoid the above problem is to define $f_i(x) = f(x) \oplus X_i$, where $X_0, X_2, \ldots, X_{t-1}$ is a pseudo-random sequence of n-bit vectors. This ensures that all output bits are uniformly modified unlike the counter method where only some least significant bits are modified. Further, the pseudo-random sequence $X_0, X_2, \ldots, X_{t-1}$ should be efficient to generate "on-the-fly". The cost of generating the next element of the sequence should be negligible compared to the cost of one invocation of f.

Choices like $f_i(x) = f(x) * i$ in $GF(2^s)$ or $f_i(x) = f(x) + f(i)$ can also provide uniform modification of the output. However, these are quite expensive operations, the first one involves a polynomial multiplication and the second one involves an extra invocation of f. *We would like to define f_i such that the cost of one invocation of f_i is almost the same as that of f.*

5.4 Introducing LFSRs as Function Generators

If one chooses the output modification functions ψ_i after f is given, it is not possible to a priori construct a Fiat-Naor type example. It has perhaps not been observed earlier that the Fiat-Naor type example can be avoided by the simple trick of choosing the variants of f randomly after the function f is provided. We discuss the suitability of LFSR in this context.

In the Hellman method, we can use an LFSR to generate the random variations of f as follows. For t Hellman tables we generate a sequence $X_1, X_2, \cdots, X_t$ of s-bit vectors using an LFSR $(s, p(x))$ (say). Then we construct the f_i's as follows: $f_i(x) = f(x) \oplus X_i$ for $i = 1, 2, \ldots t$. We require the X_i's to be distinct. Choosing $l = s$ (recall that $f : \{0,1\}^s \rightarrow \{0,1\}^s$) and $p(x)$ to be a primitive polynomial will ensure this. Since the LFSR connection polynomial and the initial condition are chosen randomly *after* f is given, it is not possible to a priori construct a Fiat-Naor type example for the LFSR based Hellman method.

We now consider the application of LFSR sequences to the generation of functions for use in (multiple) rainbow tables. Suppose there are r tables each having t columns. We choose an LFSR of length $l = s$ having a primitive connection polynomial. Each bit vector in the sequence is of length s. Let the sequence be $X_0, \ldots, X_{rt-1}$.

Define $\psi_i(x) = x \oplus X_i$ and $f_i(x) = \psi_i(f(x)) = f(x) \oplus X_i$. The first table uses the functions $f_0, \ldots, f_{t-1}$; the second table uses the functions $f_t, \ldots, f_{2t-1}$; and so on. The functions ψ_i defined using the LFSR sequence satisfy the desirable properties discussed above. We mention some details.

Invertible: Each ψ_i is clearly invertible.

Efficient Generation: The function ψ_i is defined from X_i. Since the sequence $X_0, \ldots, X_{rt-1}$ can be efficiently generated in both the forward and the backward directions, the corresponding functions can also be efficiently generated.

Hardware Implementation: A hardware implementation of the rainbow method is explored in [69] using FPGA platform, where a counter based method is used for function generation. For hardware implementation, it is preferable to use an LFSR over a counter.

Long Period: For all the ψ_i's to be distinct, we need the X_i's to be distinct. The period of the sequence is $2^s - 1$. For the rainbow method, r is a small constant and $t = N^{1/3} = 2^{s/3}$. Thus, we have $2^s - 1 > rt$ and hence all the X_i's are distinct as required.

Pseudo-randomness: LFSR sequences satisfy some nice pseudo-randomness properties [62]. Using the ψ functions in the rainbow method means that at each stage the output of f is being XORed with a bit vector from the pseudo-random sequence $X_0, \ldots, X_{t-1}$. This ensures a uniform modification of all the bits of the output of f.

5.5 LFSR Based Rainbow Method

In this section, we provide the details of the LFSR based implementation of the rainbow method. This is a modification of the basic rainbow method. The details of the following algorithm are given in Section 3.3.4. Here we use the data structures from Section 3.3.3.

Suppose r tables each of size $m \times t$ are to be constructed in the precomputation phase. Let $p(x)$ be a primitive polynomial over $GF(2)$ of degree s and $0 \neq X_1, \ldots, X_{rt}$ be a sequence of s-bit vectors produced with an LFSR having connection polynomial $p(x)$ and initial condition X_1. We define $\psi_{i,j}(x) = x \oplus X_{it+j}$ and $f_{i,j}(x) = \psi_{i,j}(f(x))$, where $i = 1, \ldots, r$ and $j = 1, \ldots, t$.

We use the same data structures SPD[ ][ ] and EPD[ ][ ] for the start points and the end points respectively, which we have used in Section 3.3.3. for the DP method. In the precomputation stage, we generate the end points in the following manner. For $1 \leq i \leq r$ and $1 \leq j \leq m$,

$$\mathsf{EPD}[i][j] \leftarrow (f_{i,t} \circ f_{i,t-1} \circ \cdots \circ f_{i,1})(\mathsf{SPD}[i][j]).$$

For each table we apply the Sort algorithm which is described in Chapter 3 to sort the (start point, end point) pairs in the increasing order of the end points. For $1 \leq i \leq r$, define $Y_i^0 = X_{it}$ and $Y_i^1 = X_{it+t}$. With the i^{th} table, we associate the pair (Y_i^0, Y_i^1). These two values mark the start and the end of the LFSR sequence required to generate the $f_{i,j}$'s used in the i^{th} table. This completes the description of the table preparation, which requires rmt invocations of the function f.

Next we describe the online search technique. We will be given y and have to find x such that $f(x) = y$. Since there are r tables, we successively search in each of the tables. Hence, it is sufficient to describe the search method in the i^{th} table. The details of the following algorithm is a little different from that in Section 3.3.4. This is due to the fact that in Section 3.3.4, the search algorithm is given for a single table, whereas the algorithm below is for searching in the i^{th} table among a set of r tables. For the sake of clarity, we present the details once more.

Algorithm Search in the i^{th} table
Input: SPD[ ][ ], EPD[ ][ ], Y_i^0, Y_i^1 and y.
Output: An s-bit string x such that $f(x) = y$, else failure

```
1.  Z = Y_i^1
2.  for j = t − 1 downto 0 do
3.      set tmp = y ⊕ Z, W = Z
4.      for l = j + 1 to t − 1 do
5.          tmp = f(tmp) ⊕ W, W = L(W)
6.      end do
7.      q ← Find(EPD[i], tmp)
8.      if (q ≠ NULL) do
9.          val ← SPD[i][q]
10.         tmp ← EPD[i][q]
11.         set W = Y_i^0
12.         for l = 0 to j − 1 do
13.             val = f(val) ⊕ W, W = L(W)
14.         end do
15.         if f(val) = y, then return val
16.         else raise a false alarm
```

Table 5.1: $(r, m, t, \mathsf{PS}, M, P, T, T_t)$ with different N (r is the number of tables, m is the number of rows in each table, t is the number columns in each table, PS is the success probability, M is the memory requirement, P is the preprocessing time, T is the run time, T_t is the number of memory access and N is the size of the search space).

N	Hellman+DP	multiple rainbow
2^{56}	$(2^{20}, 2^{20}, 2^{19}, 0.93, 2^{41}, 2^{59}, 2^{36}, 2^{20})$	$(8, 2^{35}, 2^{18}, 0.94, 2^{39}, 2^{56}, 2^{38}, 2^{21})$
2^{64}	$(2^{23}, 2^{22}, 2^{22}, 0.91, 2^{46}, 2^{67}, 2^{42}, 2^{23})$	$(4, 2^{41}, 2^{21}, 0.90, 2^{44}, 2^{64}, 2^{43}, 2^{23})$
2^{72}	$(2^{25}, 2^{25}, 2^{25}, 0.92, 2^{51}, 2^{75}, 2^{48}, 2^{25})$	$(8, 2^{46}, 2^{23}, 0.91, 2^{49}, 2^{72}, 2^{49}, 2^{26})$

17. end if
18. $Z = L^{-1}(Z)$
19. end do
20. return "failure"

end Search

The Find algorithm used in Line 7 is the same as that used in Section 3.3. The algorithm implements Equation (3.2) of Section (3.3.3). If in Line 15, we have equality, then w is a preimage of y, otherwise we have a false alarm. The total number of invocations of f made per table is the same as the rainbow method and is $\approx t^2/2$. The total number of invocations of f is $\approx rt^2/2$. The memory requirement is rm pairs of n-bit strings. Additionally, it is required to store (Y_i^0, Y_i^1) for $i = 1, \ldots, r$. Since for practical implementation, r will be much smaller than m, this storage requirement is negligible compared to the storage for the tables.

5.6 Further Analysis

Consider the tuple (# tables, # rows, # columns, PS, memory, preprocessing time, runtime, number of memory access). We have computed this tuple for each method with different values of N in Table 5.1. The success probability for multiple rainbow tables is better than the Hellman + DP method for $\lambda = 1$. Taking $\lambda > 1$, we achieve the same success probability with the rainbow method, but with little higher preprocessing time.

To achieve higher success probability with the same runtime and memory requirement of rainbow method, we can choose the parameters in a way similar to [58] as follows. We choose three constants a, b and λ such that

- the memory required $rm = \frac{N}{a}$,
- the runtime $\frac{rt^2}{2} = \frac{N}{b}$,

- size of the search space $rmt = \lambda \times N$.

Solving the three equations we get

$$r = \frac{2N}{\lambda^2 a^2 b}, \quad m = \frac{\lambda^2 ab}{2} \text{ and } t = \lambda a.$$

5.7 Parallel Implementation of TMTO Precomputation

The precomputation phase of TMTO is essentially an exhaustive search which is required to be done only once. Practical implementations of TMTO attack will use parallel f-invocation units to perform the precomputation. The problem that we consider is of generating the start points on chip. We show an LFSR based method for doing this. But before that, we consider the counter based method proposed in the literature.

Counter Based Start Point Generation: Quisquater and Standaert [81] described a generic architecture for the hardware implementation of the Hellman + DP method. Mentens et al. [69] proposed a hardware architecture for key search based on the rainbow method. A global s-bit counter is used [69] as a start point generator which is connected to each of the processors. We point out the following issues regarding this approach.

- In the analysis of the success probability of the TMTO method given by Hellman [56], the start points are assumed to be chosen uniformly at random. Using a counter does not generate random start points. However, in practice counters work as well as randomly chosen strings since the success probability of the TMTO algorithms (except for Fiat-Naor) is heuristic.
- Using a global s-bit counter (adder) to generate the start points for n processors has the following disadvantage. Some (or all) of the n processors may ask for a start point at the same time. Then there will be a delay since there is only one global counter to generate the start points. Here we assume that everything is done in the hardware. However, the hardware may be connected to a low cost 2 GHz PC which can take over certain low cost tasks such as the start point generation. We did not explore this scenario. Using n different counters for the start point generation has not been suggested in the literature.

LFSR Based Start Point Generation: To generate r tables of size $m \times t$ each, we require a total of $m \times r$ start points with s-bit each. Suppose we have n processors $P_1, P_2, \ldots, P_n$ available for the precomputation phase. We may assume $n|m$ since both are usually powers of two and $n < m$.

We choose n distinct primitive polynomials $p_1(x), \ldots, p_n(x)$ and set up a local start point generator (SPG) for processor P_i as follows. The local SPG is an implementation of a maximal length LFSR L_i with connection polynomial $p_i(x)$. The initial condition S_i for L_i is chosen randomly and loaded into L_i during the set up procedure. For preparing a single table, all the n processors run in parallel. For each table, m chains need to be computed. This is done by requiring each processor to compute m/n chains. The description of P_i is as follows.

$\mathbf{P}_i$: U_i denotes the current state of L_i

1. $U_i \leftarrow S_i, j \leftarrow 1$
2. do while $(j \leq \frac{m}{n})$
3. generate the *chain* with start point U_i
4. if the *chain reaches an end point* T_i
5. store (S_i, T_i) into Tab_i
6. $j \leftarrow j + 1$
7. end if
8. $U_i = \mathsf{next}_i(U_i)$
9. end do

end.

The function $\mathsf{next}_i()$ refers to clocking LFSR L_i once. In this design, each processor P_i has its own SPG as opposed to a global SPG for all the P_is. This simplifies the design considerably while retaining the pseudo-random characteristic of the start points.

Chapter 6

New Hardware Architecture for Generic Inversion of One-way Functions

6.1 Introduction

The most feasible implementation of time/memory trade-off (TMTO) is in special purpose hardware. In this chapter, we describe a systematic architecture for implementing TMTO. We break down the offline and online phases into simpler tasks and identify opportunities for pipelining and parallelism. This results in a sufficiently detailed top-level architecture. Many of our design choices are based on intuition. Simulation studies to verify and/or propose new design choices is a possible future work. This chapter is based on our article [74].

6.2 Notational Convention and Abbreviation

We provide below the notations used in the architecture and illustrate notational convention in Figure 6.1.

– SCC: two-bit register used in the table preparation stage

– sgc1 and sgc2 are the completion signals of the chain computation and sorting unit respectively.

– start signal indicates that the assembly line movement is complete in the Table Preparation stage.

– P_i: $i = 1, 2, \ldots, n$ are the processors used to generates the (start point, end point) pairs for the table.

– PMS_i :$i = 1, 2, \ldots, n$ are the processor memory space for P_i.

– R is n-bit register used to store the completion signal of all the processors.

– SC is sequential circuit with n-bit input to check whether all the input bits are 1.

– L is s-bit LFSR corresponding to a primitive polynomial whose internal stage are used for output modifications.

– CT: one bit tag to control write blocks and movement of the assembly line.

– T: one bit tag to control the execution of the processor unit, if $T = 0$ then the unit will be idle until $T = 1$.

– SPG: the start point generator.

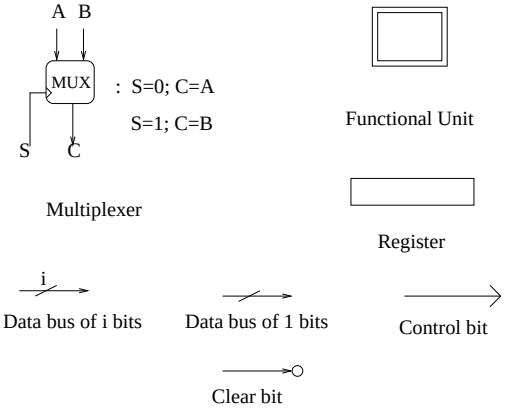

Figure 6.1: Notational convention

– C_1 and C_3 are both r_1 $(= \log t)$ bits counter. C_2 is r_2 $(= \log \frac{t}{n})$ bits counter.

– SC_1 and SC_2 are sequential circuits with r_1-bit input to check whether all the input bits are 1.

– R_i: $i = 1, 2, 3, 4, 5$ are s-bit registers.

– RF_i is the i^{th} round function.

– SPR_i: $i = 1, 2, \ldots q$ are s-bit registers used to store the start points.

– CQR is r_1-bit counter to count the number of the start points generated by SPG.

– DB: data block

– R_{2j}: $j = 1, 2, \ldots q$ are s-bit intermediate registers to store the output values for different rounds.

– SPC_i : $j = 1, 2, \ldots q - 1$ are r_1-bit counters.

– WB: write block, RB: read block and DB: data block.

– CQ: k-bit register.

– SCQ is sequential circuit with k-bit input to check whether all the input bits are 1.

– y : data point

– DP : distinguished point

– SP : start point

– $mask$:

– OMB : output memory block

– MR and DR are both s-bit registers.

– PC_1 is r_3-bit ($r_3 = \log z$) counter.

– $BUF1$ and $BUF2$ are buffer queues.

6.3 Precomputation Stage

The precomputation stage consists of two phases: *chain computation* and *sorting*. Figure 6.2 describes architecture of the precomputation stage and the tables are computed one by one. To generate a table, a fresh memory is used as an input of the chain computation phase. In the chain computation phase, chains are generated until it reaches a DP and then the start point and end point pairs are stored into the fresh memory. After storing t of pairs into the table, the chain computation unit sends a completion signal `sgc1` (1 bit value) to

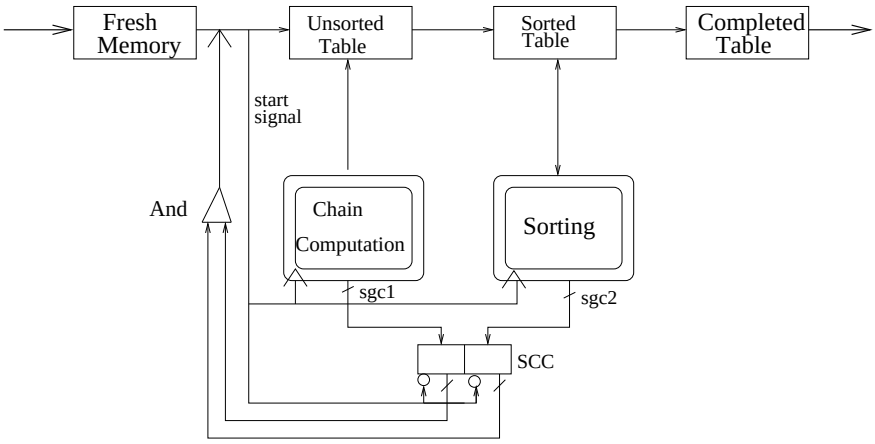

Figure 6.2: Table Preparation

the register SCC and terminates execution until the start signal is received. At the sorting phase, the previous table (unsorted) is to be sorted into increasing order of the end points. Both the chain computation unit and the sorting unit run in parallel, i.e., while the chain computation unit computes the i^{th} table, the sorting unit performs sorting on the $(i-1)^{th}$ table. After completion of the sorting phase, a completion signal sgc2 is sent to SCC and the execution is stopped until a start signal is received. The assembly line will shift (i.e., the fresh memory, unsorted table and sorted table will be copied into unsorted memory unit, sorted memory unit and completed table unit respectively) when SCC receives both the signals sgc1 and sgc2 (i.e., when both the chain computation unit and sorting unit will report completion). After completion of assembly line movement, start signal will be sent to both the chain computation unit and the sorting unit and the SCC is set to zero. There are several issues to be considered.

- The chain computation and the sorting hardware are to be designed so that they complete simultaneously. In any case, sorting should not take more time than chain computation. Since we are considering a pipelined design, if sorting takes more time than the chain computation, the pipeline will have to wait. Under the assumption that the sorting does not take more time than chain computation, the total time will be r executions of chain computation + 1 table sorting + r assembly line movements. The time between two assembly line shifts is the time for the chain computation which is sufficient time to add fresh memory and to remove the completed table. On the other hand, the actual shift of the assembly line should take very small time as this is a factor in determining the completion time of all the tables.

- *Parallel sorting:* Depending on the design and speed of the chain computation stage, it is required to determine whether parallel in-place sorting is required. The other issue

is the type of table memory being used and whether random access is supported. In case parallel sorting is to be used, one can use mesh sort which requires a 2-d table structure. Then the chain computation phase will be required to access a 2-d memory.

- Both chain computation and sorting phase will require memory writes. For the chain computation stage, batching can be used to reduce the number of memory accesses. Also chain computation and memory access can be pipelined to some extent.
- For storing tables, one can use the re-writable DVDs which is a cheap option. But the DVDs have write stage and a burn stage, so the question is where does the burn stage fit into the design? Does this stall the pipeline? Alternatively, one can use four blocks of high speed memory while keeping the actual tables into the DVDs. The completed table in a high speed memory will be written to a DVD and then the high speed memory will be cycled back into fresh memory. The time to copy from high speed memory to DVD will be overlapped with the chain computation and the sorting phases.

Comments: We consider parallel sorting in hardware. Alternatively, one could consider harware-software co-design where sorting is done in software.

6.3.1 Chain Computation Phase

Suppose there are n processor units $P_1, P_2, \ldots, P_n$ available at the chain computation phase. In Figure 6.3, we describe the architecture of the chain computation unit. The given memory block (fresh memory) is partitioned into n separate Processor Memory Space (PMS) units $PMS_1, PMS_2, \ldots, PMS_n$. Each processor P_i will store $\frac{t}{n}$ (start point, end point) pairs into PMS_i through a write block (WB) unit WB_i. Each PMS_i has $\frac{t}{n}$ memory locations to store the pairs and its starting address add_i (address of the first memory location) is stored in WB_i. Hence to access j^{th} memory location of PMS_i, the offset j is to be added with add_i to get the exact address. Processors execute the chains with different start points which are coming from *start point generator*, with each processor having its own start point generator. After encountering a DP, the processor enables the *write block* unit by the signal sg_1 and passes the address (offset:O_3) of the next free location of the corresponding PMS. Then the corresponding WB unit goes to the exact address of the free location by adding the offset with the starting address of the PMS and storing the pair (O_1, O_2) into the location.

Processors run in parallel and after generating $\frac{t}{n}$ DPs, the corresponding processor passes a completion signal (1-bit value) to the n-bit register R and stops the execution until it receives a start signal sg_2 from the CT (see Figure 6.2). L is an s-bit LFSR which is used as function generator and its internal state value passes to each of the processors to do the

output modification of the function f. SC is a sequential circuit to check whether all the values R are 1. If yes, then the table has completed and SC sends a signal sg_4 to enable L to generate the next state (for the next table) and sg_3 to set CT to 1. Then CT will send a signal $sgc1$ to SCC (see Figure 6.2) requesting to move the table, a signal sg_5 to disable write block, a signal sg_2 to the processor and clear the contents of R. After the movement of the assembly line, the start signal (see Figure 6.2) sets the value of CT to zero and the write blocks will be enabled to write the pairs for the next table.

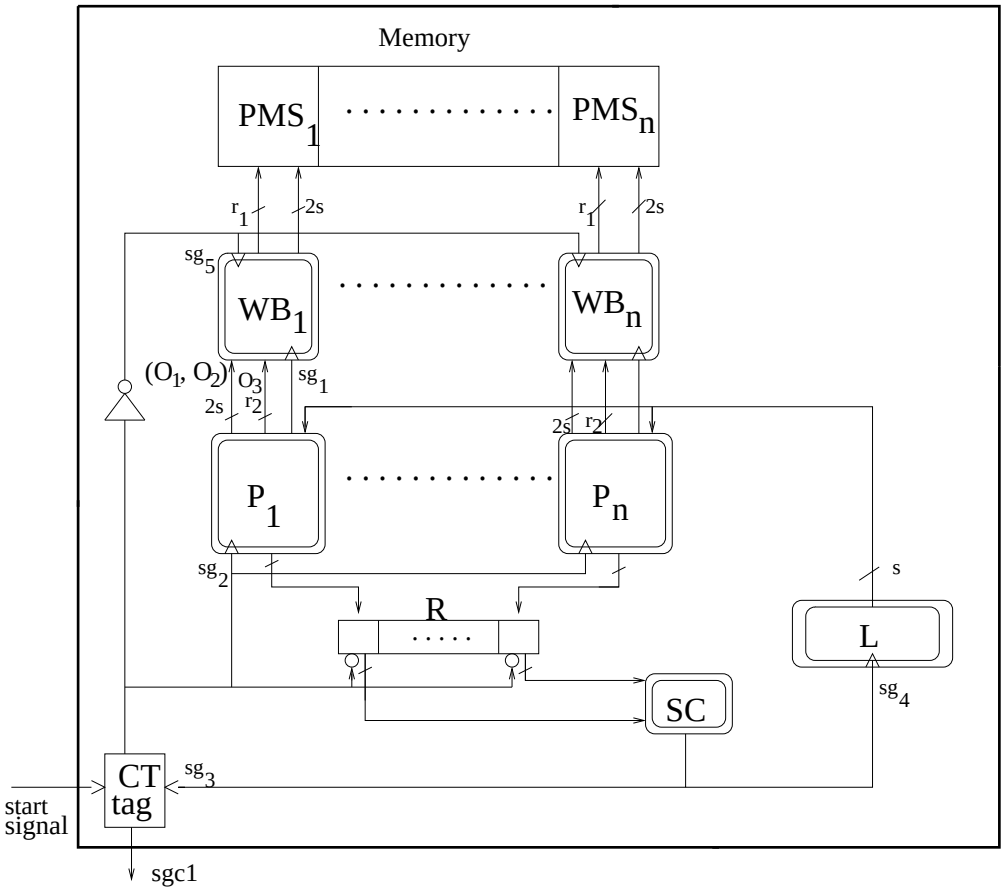

Figure 6.3: Architecture of chain computation phase. i/p : start signal; o/p : sgc1

The following are some of the rationales for our design decisions:

- *Utility of having separate memory spaces:* Each processor P_i uses separate processor memory spaces PMS_i to store the (start point, end point) pairs. This avoids multiple

access of same memory space and it is possible to use this idea since sorting is done separately.

- *Each processor generates $\frac{t}{n}$ DPs:* Since DPs are generated at different time points and a processor may have to consider different number of chains, the time taken by processor will be different (though the expected time will be the same for all processors). Consequently, it may happen that one processor may complete ahead of others and hence will be idle for some time. On the other hand allowing each processor to generate the same number of DPs considerably simplifies the design.
- *No overlap of processing between tables:* At no point of time, two processors will be handling chains of different tables. This again simplifies design.

Comments: We assume that there will be no overlap of processing between tables. While this simplifies the design, it may be more cost effective to allow such overlaps. More analysis is required to settle this point.

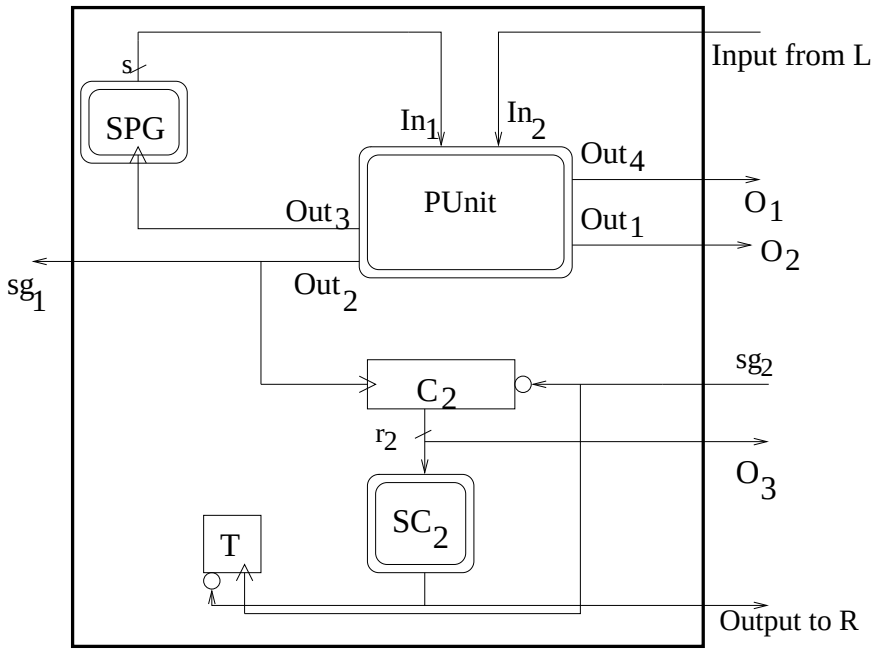

Figure 6.4: Architecture of a processor P_i. $i/p : sg_2$, function mask; $o/p : sg_1, O_1, O_2, O_3$.

Description of a Processor: Figure 6.4 describes the architecture of a processor. Each processor takes two inputs, a signal sg_2 and s-bit output modification value from L. The 1-bit register T is the control unit of the whole processor unit, the processor will stop if T is set to be zero and start running if the value of T is 1. C_2 is the counter to count the number of DPs encountered and it is incremented after encountering a DP. SC_2 checks whether the number of DPs encountered reaches $\frac{t}{n}$. If yes then the value of T will be set to zero and the whole processor unit will stop until the signal sg_2 resets C_2 to zero. A start point is generated by the *start point generator (SPG)* unit and passes to the $PUnit$ as the input In_1. Then the $PUnit$ takes other input In_2 from L, which is the internal state of L (i.e., function mask) and starts executing the chain with the start point until it reaches a DP or the chain length reaches t. If yes then it outputs a signal Out_3 to SPG to generate a new start point, loads into the register R_1 and passes to the $PUnit$ as an input (In_1) for the next chain. If a DP is encountered, then $PUnit$ outputs a signal Out_2 to increase the counter C_2 by 1 and enables (the signal sg_1) the WB unit to load the (start point, end point) pair (O_1, O_2) and the offset address O_3.
A suggestion for SPG to be implemented using an LFSR where each P_i has its own SPG as opposed to a global SPG for all the P_is. See Section 5.7 for parallel start points generation using LFSRs sequences. This simplifies the design considerably while retaining the pseudo-random characteristic of the start points.

Description of $PUnit$: Figure 6.5 describes the $PUnit$ where input In_1 is a new start point which is loaded into the register R_1 and In_2 is stored into R_4 for function masking. The counter C_3 is set to zero through the multiplexers when a new start point is loaded into R_2. The function f is applied on R_2 and the output is loaded into the register R_3 followed by function masking (xoring R_3 and R_4). The result is stored into the register R_5 to check for a DP. If a DP is encountered, then the multiplexers select the second line so that a new start point is loaded into R_2 and the counter C_2 will set to zero. Otherwise R_5 and C_1 will be copied (in a synchronized operation) into R_2 and C_3 respectively for the next iteration in the chain. The increment of C_3 and copying to C_1 will be synchronized with the application of f on R_2 and output to R_3. The result of one operation will not be used until the other one is completed.

Comments: Note that in our design we use chain length counter (i.e., C_1) which adds complexity to the circuit. Removing the chain length counter gives rise to the possibility that a DP in some chain occurs after a very long time or does not occur at all. This will stall the operation. While this will be rare event, it cannot be ignored. Counter chain length is one way of handling this. There may be other ways. Also note that we do not store chain length in the table. This reduces memory requirement but will increase online search time for false alarms.

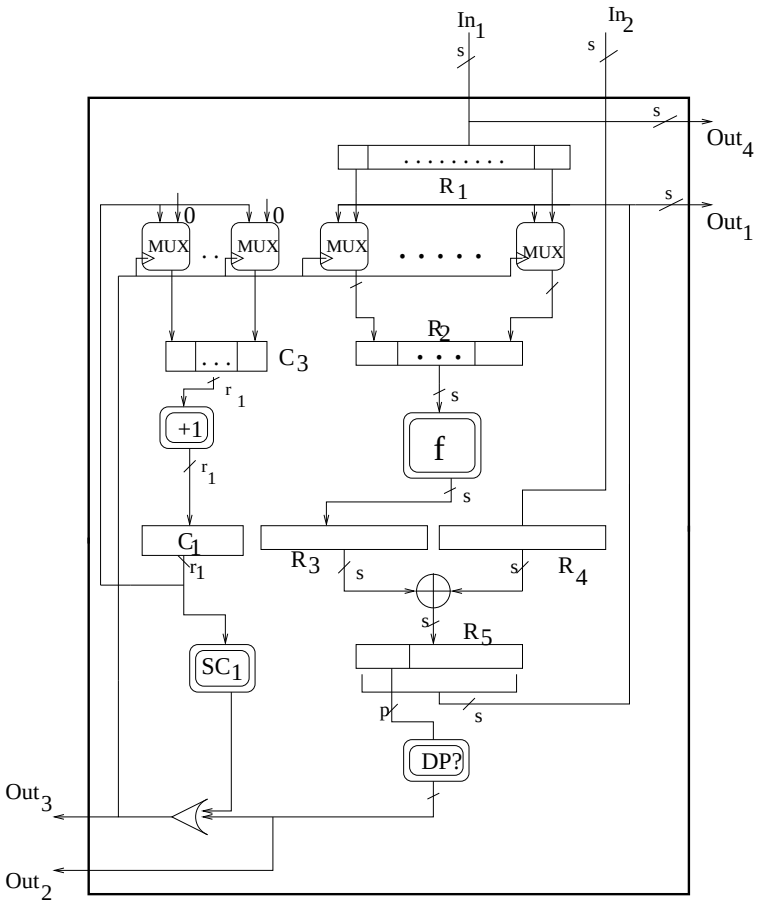

Figure 6.5: Architecture of $PUnit$. $i/p : In_1, In_2$; $o/p : Out_1, Out_2, Out_3, Out_4$.

Description of a processor when f is a multi-round function: Let us consider the case when the function f is a multi-round function, i.e.,

$$f = RF_q \circ RF_{q-1} \circ \cdots \circ RF_2 \circ RF_1$$

where q is the number of rounds. For example DES and AES are multi-round block ciphers and A5/3 [1] is an example of a stream cipher which can be seen as mode of operation of the block cipher KASUMI [2]. We apply q-stage pipeline strategies to deal with q-different chains in parallel within a processor as follows (this idea has been earlier used in [81] and [99]). In the architecture of a processor unit (Figure 6.4), the $PUnit$ is replaced by $PUnitRound$ (the description of PUnitRound is given below). For each table, the SPG unit generates q start points initially.

Description of *PUnitRound*: Figure 6.6 describes the *PUnitRound*. We use $q+1$ counters $SPC_1, SPC_2, \ldots, SPC_q, C_1$ of r_1-bit each. Initially, the SPG generates q start points. At each time, the start point in register SPR_i will be copied into the next register SPR_{i+1} to keep track of it, since after getting a DP, we need to get the corresponding start point to return. A pipelining strategy is applied in the execution of the round function and whenever a DP is encountered, the processor outputs the DP and the corresponding start point which is available at the register SPR_{q+1}. The following are synchronized operations:

- Copying SPC_i to SPC_{i+1}, SPR_i to SPR_{i+1} and R_{2i} to R_{2i+1} for $i = 1, 2, \ldots, q-1$.
- Copying SPC_q to C_1, SPR_q to SPR_{q+1} and R_{2q} to R_3.
- Copying C_1/"0" to SPC_1, SPR_{q+1} to SPR_1 and R_5 to R_{21}.

6.3.2 Sorting Phase

We do not describe details of sorting hardware but discuss the various issues that need to be considered. The sorting hardware is designed in such a way that the sorting and the chain computation should complete simultaneously. In the chain computation phase, for a table with size $m \times t$, the total f invocations required is mt whereas the sorting phase could be done in $m \log m$ comparison using a single processor and the sorting should be *in-place*. If we have t processors available at the chain computation phase, then total number of f invocations will be reduced from mt to m by running the processors in parallel. But for significantly large t, t processors may be expensive. Also one f invocation takes more time than one comparison operation for sorting. So sorting with a single processor will not take more time than chain computation. But the chain computation requires memory write which is done in parallel and sorting requires both memory read and write. Hence depending on the memory speed one may have to perform parallel sorting (including memory read and write) so that the sorting and the chain computation phase complete simultaneously.
Note that, at the sorting phase if there is a collision (i.e., common DP in different chains), then we randomly select one chain to store and remove others, but it is desirable to select the maximum length chain for getting more coverage. In our design we are not storing the individual chain length in the table, so we cannot take the maximum length chain for the collision. Also since the sorting phase starts after completion of the chain computation phase for a table, we may need to remove some of the chains at the sorting phase due to the collision. Thus to get a constant coverage, more chains need to be computed in the chain precompuation phase. On the whole, our design is simpler and requires less amount of memory since we do not take the extra overhead of storing individual chain length.

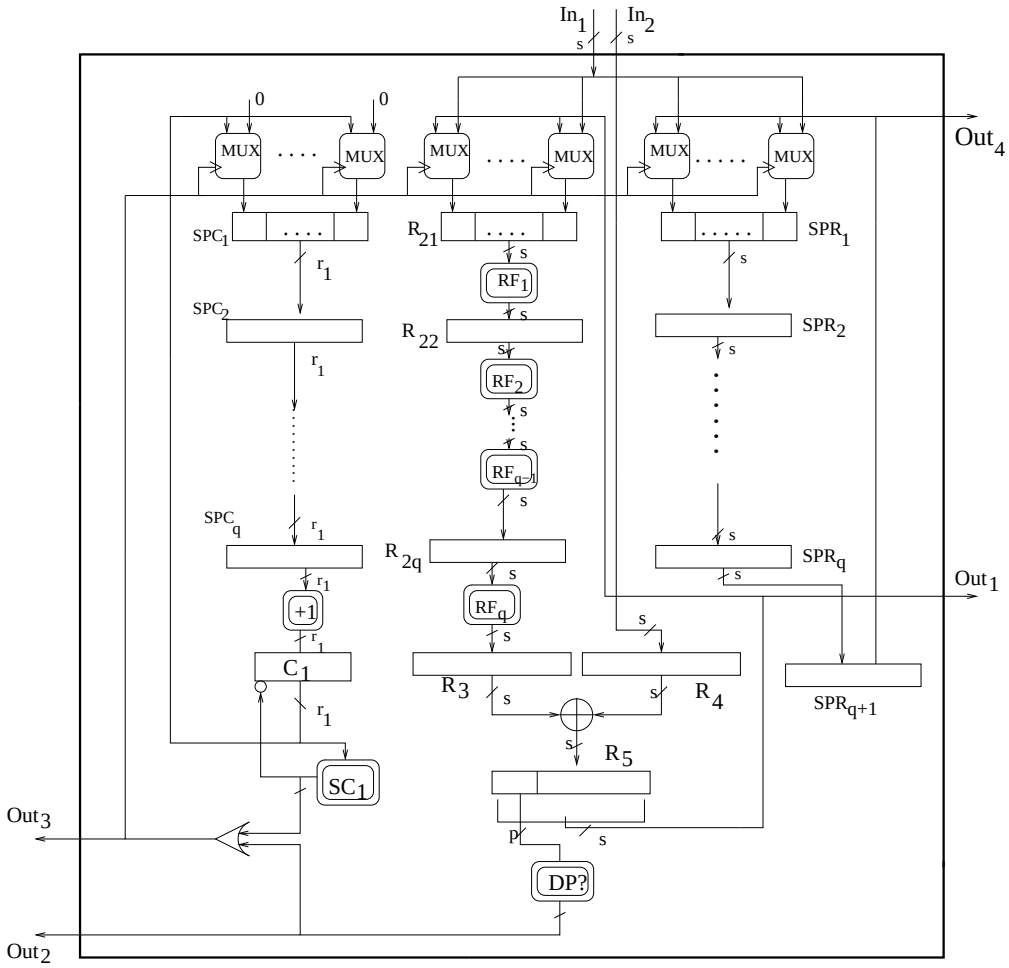

Figure 6.6: Architecture of PunitRound. $i/p : In_1$, In_2; $o/p : Out_1, Out_2, Out_3, Out_4$.

6.4 Online Search

The online stage consists of two phases – *matching* and *find key*. In the matching phase, table lookup is performed when a DP is encountered during an iteration (execution of the chain starting with the given value y). If the encountered DP is not in the table, then we will not be able to find the key by iterating further and can skip the current search in the rest of this table for that given value y. To search the key in the i^{th} table, we need to execute the following chain.

$$y \xrightarrow{\phi_i} k_{i+0} \xrightarrow{f} \xrightarrow{\phi_i} k_{i+1} \xrightarrow{f} \xrightarrow{\phi_i} k_{i+2} \rightarrow \ldots \rightarrow k_{i+t-1} \xrightarrow{f} \xrightarrow{\phi_i} k_{i+t}.$$

After each iteration (f application + masking (ϕ_i)) DP is checked and if found we stop the chain.

Suppose D points $y_1, \ldots, y_D$ are available at the online stage and we have to find the preimage of any one of these points where y_i, $i = 1, 2, \ldots, D$ are viewed as unrelated random points. This enables us to perform independent search for different data points. Suppose we have processors $P_1, P_2, \ldots, P_n$ which are dedicated to perform the f invocation and $Q_1, Q_2, \ldots Q_k$ are I/O processors to perform the table lookups. We now describe the matching phase architecture for the following cases depending on the value of D.

6.4.1 For Many Data Points

Let there be sufficient amount of data available to the attacker. We partition the data points into n separate data blocks $DB_1, DB_2, \ldots, DB_n$ with z data points in each. Then $D = z \times n$. We apply the search technique for all the data within a single table and after completion of the search for all data points we move to the next table. *Since table load is expensive, we complete the search on one table before moving onto the next table.*

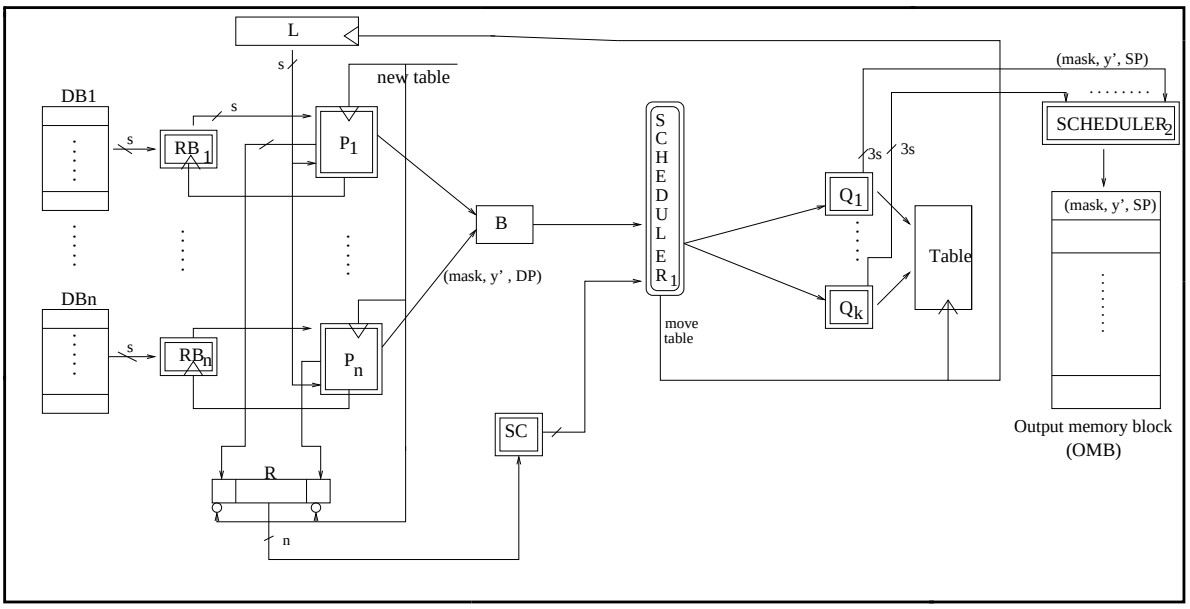

Figure 6.7: Architecture for the matching phase for a single table when D is large. i/p : Data points; o/p : OMB.

Figure 6.7 illustrates the architecture to perform the parallel searching for a single table and for all data points. The processor P_i stores a data point from DB_i into its internal register

DR and the mask value corresponding to the table (coming from L) in the register MR (see Figure 6.8 that describes a P-processor). The counters PC_1 keeps track of the number of data points already covered. The P-processors and Q-processors are connected through a common buffer queue B and a scheduler $\mathsf{SCHEDULER}_1$. To search in a table, the processors $P_1, P_2, \ldots P_n$ are assigned to perform the DP search technique for different data in parallel where the data coming from its data block through the read block unit. So these processors are essentially executing n different chains corresponding to n different data in parallel. The table lookup is needed whenever a DP is encountered during the execution of the chain while searching the key in a table.

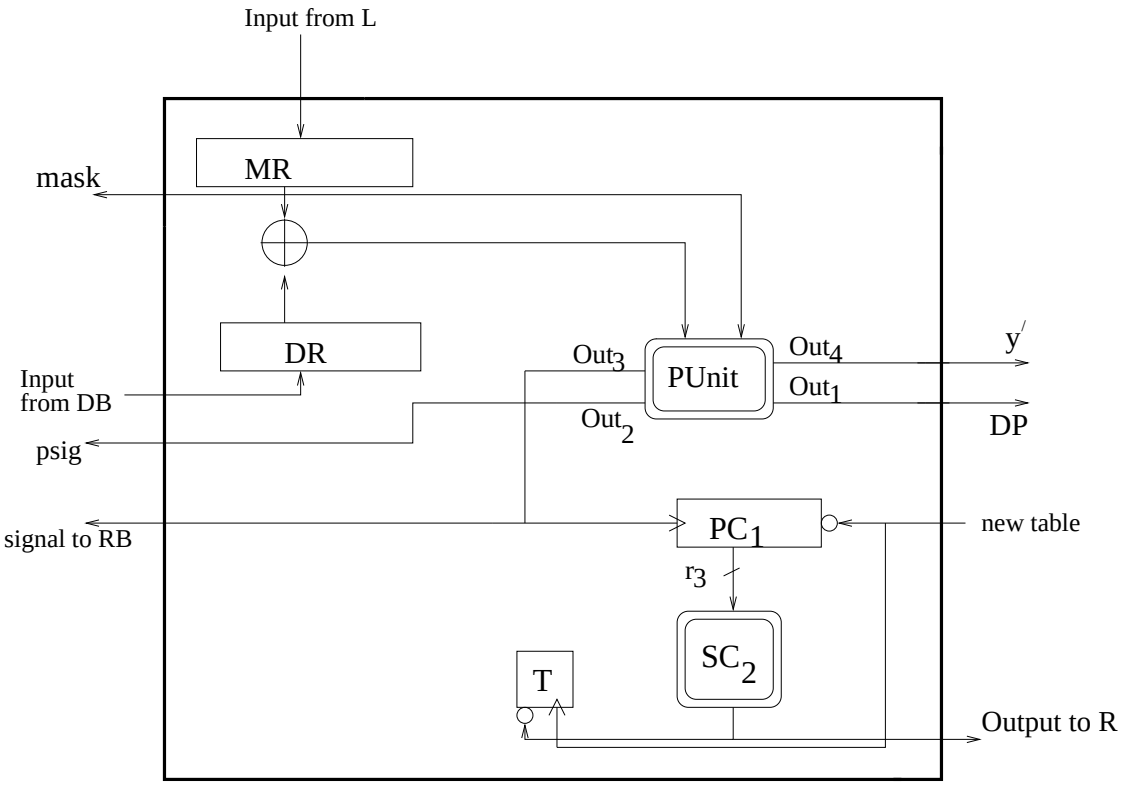

Figure 6.8: Architecture of a p-processor in the matching phase architecture when D is large. i/p : new table signal, y, function mask; o/p : mask, y', DP, psig, load signal to RB, completion signal to R.

After encountering a DP during the execution of the chain, the corresponding processor generates a signal $psig$. It then passes the tuple $(mask, y', DP)$ to B and a load signal where "mask" is generated by LFSR L and $y' = masking(y)$. The processor also passes a signal to RB to read the next data point from data block. After this, the processor starts executing

the chain for the next data. In this way, each P-processors keeps on executing until it finishes the search for all z data points. After completion it sends a completion signal to the register R. The processors wait until it receives the new table signal. The job of the $\mathsf{SCHEDULER}_1$ is to check the buffer queue. If there is any tuple in the buffer queue then it searches for a free I/O processor and if there is any free I/O processor, it assigns the DP into the (the ordering is $Q_1, Q_2, \ldots, Q_k$ circularly) free I/O processor for the table lookups. The queue size is to be chosen such that it will not be full until there is a free I/O processor.

During the table lookup, if a match occurs then the corresponding Q-processor passes the tuple $(mask, y', SP)$ (SP is the start point for the data y) to the $\mathsf{SCHEDULER}_2$ to store the tuple into OMB to get the key. After receiving a completion signal from all P-processors the $\mathsf{SCHEDULER}_1$ checks whether B is empty and all the Q-processors have finished the table lookups. After completion of all table lookups, a new table will be loaded and the algorithm continues.

Analysis: Let 2^{-p} be the probability of a point being a DP. Hence, we can expect one DP in a random collection of 2^p points. In our parallel execution, n processors are executing in parallel and generating n random points each time. Assuming $n < 2^p$, after each $\lfloor \frac{2^p}{n} \rfloor = t_1$ (say) iterations we can expect one DP. At each of time $T = it_1$ for $i = 1, 2, \ldots$, we can expect a DP. The encountered DP will be assigned to the I/O processors for table lookups. Thus at time $T = it_1$, the corresponding DP will be assigned to the processor Q_i for $i = 1, 2, \ldots, k$. The next DP will be encountered (expected) at time $T = (k+1)t_1$, but at that time the I/O processor Q_1 may not be free, since the table lookup time (γ) is quite significant. Let us consider the following cases.

Case 1: When $kt_1 = \gamma$, i.e., $k2^p = \gamma n$ then at time $T = (k+1)t_1$, the I/O processor Q_1 will be free (since the time difference between the present time and the time when the processor Q_1 was assigned the DP is $(k+1)t_1 - t_1 = kt_1 = \gamma$, the table lookup time). So the corresponding DP will be assigned to Q_1 for table lookup. In this way the next DP will be assigned to Q_2 and so on. So in this case all the processors will remain busy at all the time. For a table with size $m \times t$, the total number of f invocations will be reduced from tD to $\lfloor \frac{tD}{n} \rfloor$. Then the total runtime for a single table is $\frac{tD}{n} + \gamma$. Hence in this case the total number of f invocations is reduced by a factor of n and the *effective* number of table lookups required is only one for a single table.

Case 2: When $kt_1 < \gamma$, then we need to use the buffer queue. Note that the DPs are coming at the following expected times:

$$T = t_1, 2t_1, 3t_1, \ldots, kt_1, (k+1)t_1, \ldots$$

So upto time kt_1, we keep on assigning the DPs into the I/O processors. But after that the next (circular ordering) I/O processor, i.e., Q_1 will be free at time $t_1 + \gamma$. Thus the next generated DPs need to be stored into the waiting queue upto time $T = (k + \jmath)t_1$, where $\jmath$ is

the integer such that, $(k+\jmath-1)t_1 < \gamma < (k+\jmath)t_1$. Thus the size of the buffer queue should be $\jmath$ for no delay so that all the processors will remain busy at all the time. Hence in this case the total number of invocations of f is also reduced by a factor of n and the *effective* number of table lookups required is $\jmath$. Hence, total runtime for a single table is $\frac{tD}{n} + \jmath\gamma$.
Case 3: When $kt_1 > \gamma$, this case is similar to the case 1, except that in this case not all the k I/O processors will be busy, some of the I/O processors will always be idle which is not desirable. So this case is not suggested.
In Figure 6.7, k I/O processors are randomly accessing the table (memory block). Hence the memory block needs to have multiple data and address bus to support this multiple access. The table lookup time $\gamma = \delta \log m$ where δ is the memory access time. Note that in the above analysis, we have assumed that the Q-processors will have finished their table lookups in the same ordering which may not be true. More than one match can occur at the same time for the Q-processors and that is the reason we need to have $\mathsf{SCHEDULER}_2$.

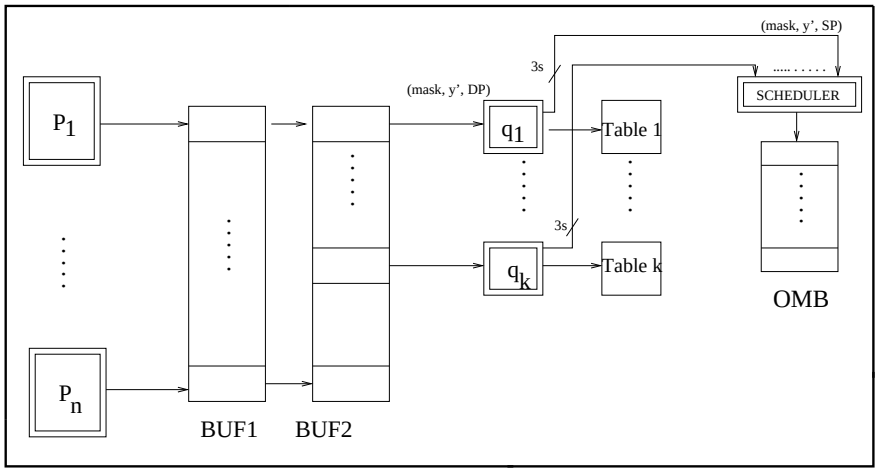

Figure 6.9: Architecture for matching phase when $D = 1$.

6.4.2 For a Single Data Point

Suppose the attacker has a single data point at the online stage, i.e., $D = 1$. We perform the parallel search strategy by grouping tables $GT_1 = \{Table_1, Table_2, \ldots, Table_n\}, GT_2 = \{Table_{n+1}, Table_{n+2}, \ldots, Table_{2n}\}, \ldots$ such that each group contains n tables. In Figure 6.9, we describe the architecture where P_i's are the processor units running in parallel to search for DP for n tables from the same group in the increasing order of the table number. After encountering a DP, the processor passes the quadruple $(mask, y', DP)$ to BUF_1 and waits

until the other processors finish their DP search for the same group. After completion of DP search for all the tables in the group, BUF_1 will be stored into BUF_2 in increasing order of table numbers and the P-processors start searching the DP for the next group of tables. The processor P_i is similar to the processor which is used in the many data points case (see Figure 6.8) expect the following. (1) Register DR will always contains the given data point. (2) For each group of tables, the LFSR (L) will clock n times to get the corresponding mask value for the tables and store it to the register MR for each table.

For table lookup, k Q-processors are connected to the first k position of BUF_2. Parallel table lookup is perform for the first k tables in the group and after completion of all these k tables, we pop the first k tuples and push the next k tuples in the first k places of BUF_2. Then we load next k tables from the same group for table lookup. This technique can also be used when D is small.

Analysis: We can expect one DP in a random collection of 2^p points. In our parallel execution, n processors are executing in parallel and generating n random points each time. Hence after 2^p time, the expected number of DPs is n which completes the f invocation for a group of tables. Let $T_1 = 2^p$. The time required to complete table lookups for a group $= \frac{\gamma n}{k}$ since k Q-processor are running in parallel. Let $T_2 = \frac{\gamma n}{k}$. Let us consider the following cases.
Case 1: When $T_1 = T_2$, i.e., the total expected time required to complete f invocations is same as the total time required to complete table lookups for a group. Then the following will be done simultaneously: (1) i^{th} group of tables, i.e., GT_i completes the f invocation stage and (2) $(i-1)^{th}$ group of tables, i.e., GT_{i-1} has completed the table lookup stage. There are total $\lfloor \frac{r}{n} \rfloor$ group of tables. Hence the expected runtime required to complete the matching phase in this case = total time required to complete f invocation stage for $\lfloor \frac{r}{n} \rfloor$ group of tables + time required to complete the table lookup step for the last group ($GT_{\lfloor \frac{r}{n} \rfloor}$) of table $= \lfloor \frac{r}{n} \rfloor \times 2^p + \frac{\gamma n}{k} = \left(\lfloor \frac{r}{n} \rfloor + 1 \right) 2^p$.
Case 2: When $T_1 < T_2$, i.e., table lookup time dominates the total time. The total table lookup time $= \frac{r}{n} \times \frac{\gamma n}{k} = \frac{\gamma t}{k}$, which is independent of n. Hence in this case we suggest a single P-processor, i.e., $n = 1$.
Case 3: $T_1 > T_2$. After completion of table lookup for GT_{i-1}, the Q-processor has to wait until the P-processor completes the f invocation stage for GT_i.

6.5 Finding the Key

After a match in table lookup step, we come to the corresponding start point and repeatedly apply the function (f + masking) until it reaches $masking(y)$. The previous value it visited

is k or this might be a false alarm. See section 3.3.2 for details about the false alarm. Hence to get the key from the given $(mask, y', SP)$, the following chain is executed.

$$SP \xrightarrow{f} \xrightarrow{masking} k_1 \rightarrow \ldots \rightarrow k_i \xrightarrow{f} y \xrightarrow{masking} k_{i+1} \rightarrow \ldots \rightarrow k_{i+t-1} \xrightarrow{f} \xrightarrow{masking} DP. \qquad (1)$$

Figure 6.10 describes the architecture for parallel key find strategy where n processors $P_1, P_2, \ldots, P_n$ are running in parallel taking input tuples from OMB.

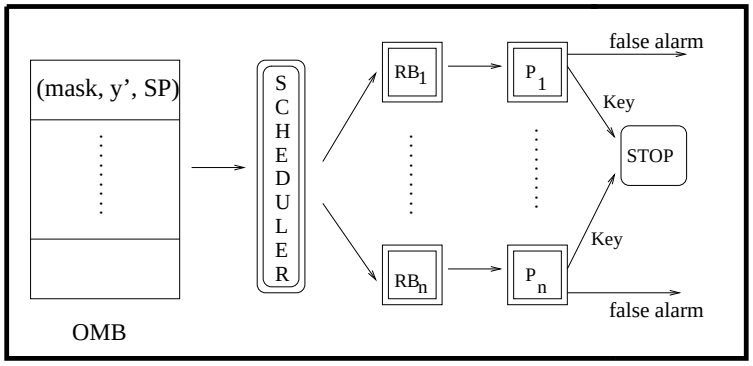

Figure 6.10: Architecture for parallel key find strategy

6.5.1 Description of a Processor

Figure 6.11 describes the P-processor in the parallel key find architecture. Each processor takes the input $(mask, y, SP)$ from the OMB and stores $mask, y', SP$ respectively into the registers R_8, R_6 and R_1. Then the processor executes chain (1). Every time it checks for equality with y' and stops if it finds the match and returns the previous point as the key. If it finds no match after executing complete chain (length t), it returns a false alarm.

6.5.2 Analysis

Finding the key can require "substantial" time compared to finding a table match due to false alarms. The number of false alarms can be as large as half the total number of f required for online phase. However, table match requires memory access, whereas finding the key does not.

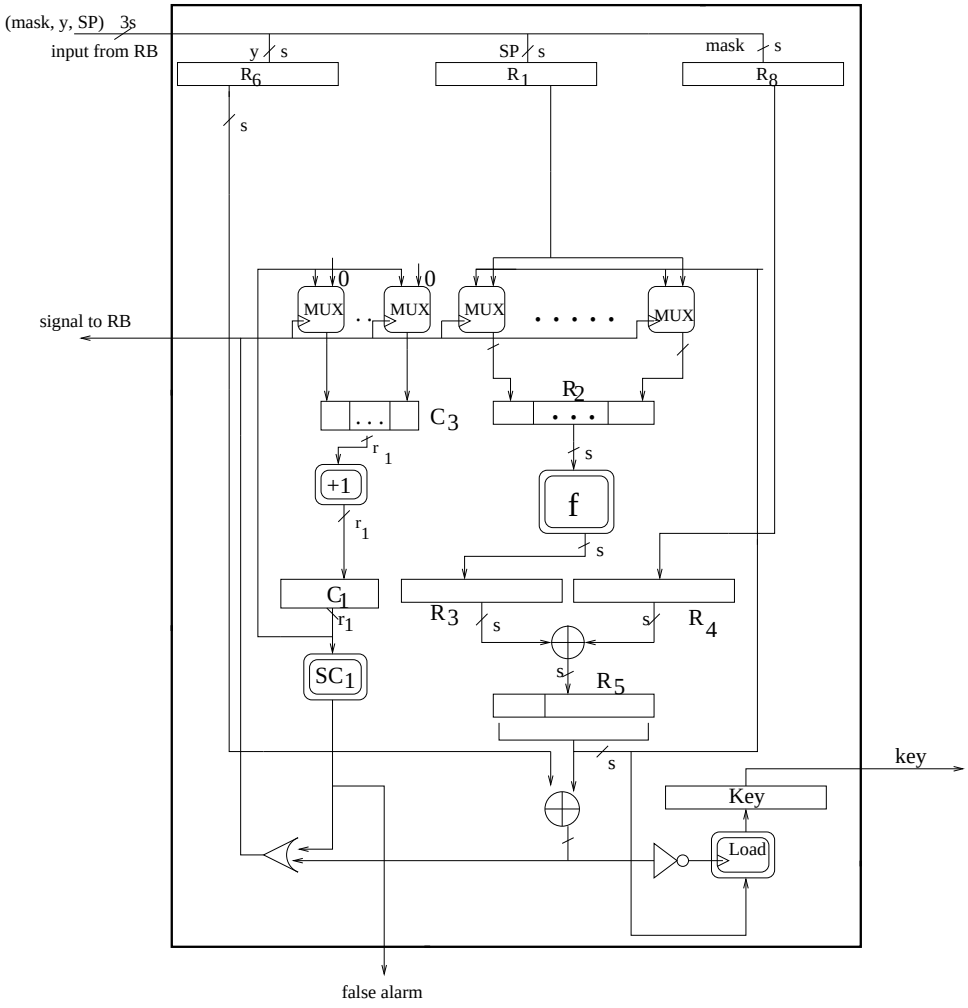

Figure 6.11: P-processor in the parallel key find architecture

Chapter 7

On the Effectiveness of TMTO and Exhaustive Search Attacks

7.1 Introduction

This chapter develops a cost model for analyzing the effectiveness of generic attacks. Our main emphases is to show how such a cost model can be developed. (A general cost model is given in Section 7.2.2.) The cost model is used to obtain estimates of the resistance of different key sizes to generic attacks. Such resistance is quantified by two parameters: the budget the adversary can afford and the duration for which security is required. To a certain extent these are sensitive to available technology. In obtaining approximate estimates, we assume the use of certain technology. Use of more advanced technology will change these estimates. For example, we assume the use of DVDs whereas HD DVD and Blu-Ray (upto 50 Gybtes recordable) are currently arriving on the market. Also, to project the estimates into the future we need to use Moore's law to translate the cost model in Section 7.2.2.

Our cost model for a TMTO attack is based on the architecture in Chapter 6. We assume that each processor is implemented using FPGA. Since the number of processors will be large it will be more cost effective to use ASIC design. We leave the task of developing an ASIC design based cost model as future work. Our cost analysis shows that 128-bit keys seem secure for the possible future. In contrast, key sizes less than 96 bits do not seem to provide comfortable security margins. The case of 80-bit key sizes can be of concern. The generic attacks (either exhaustive search or TMTO) on 80-bit keys are within the reach of powerful organizations. However, if the adversary's budget is limited to a few ten thousand dollars, then 80-bit keys may provide limited duration security against generic attacks.

7.2 Cost Analysis

Let f be the one-way function that we want to invert and N be the size of its domain. We would like to perform a cost-time analysis of TMTO and exhaustive search attacks. To do this, we need to identify the dominant components of both the attack time and the costs. This is relatively easy to do for exhaustive search. The function f has to be applied on every possible input in the domain. Hence, the dominant component of the time is the time required to apply f a total of N times. For parallel implementation, this time is scaled down by the number of processors used. The dominant cost component is the cost of implementing the parallel f-invocation units (or processors). The cost should also include the manpower cost, but this is harder to estimate.

A TMTO algorithm is more complex than exhaustive search and deriving an appropriate cost model is more difficult. The precomputation phase of the TMTO algorithm has several time components – time required to obtain the (start point, end point) pairs; memory access time required to store these pairs into the table; and the time required to sort the tables. The online time has two major components – time to obtain the end points; and the time for table lookup. Similarly, the cost has several components – the cost of the parallel f-invocation units; and the cost of storage media.

To a large extent, the appropriate choice of the cost model depends on the underlying architecture used for the implementation. Below we provide a top level description of the architecture described in Chapter 6. This top level view makes understanding of the cost analysis easier.

Precomputation Phase: Let us consider the tasks performed in the precomputation phase. At a top level this consists of the following two separate tasks for each table.

1. Compute the chains and write the (start point, end point) pairs to the table.
2. Sort the table.
3. Write the table into a DVD.

Let us call the first task, chain-computation, the second task sorting, and third task DVDwrite. We take the following issues in consideration.

- Chain computation and sorting hardware should be designed so that they complete simultaneously. In any case, sorting should not take more time than chain computation.
- Both chain computation and sorting phase will require memory writes. For the chain computation stage, batching can be used to reduce the number of memory accesses. Also chain computation and memory access can be pipelined to some extent.

- We use few blocks of high speed memory while keeping the actual tables into DVDs. The completed table in a high speed memory will be written to a DVD and then the high speed memory will be cycled back into fresh memory.
- If the DVD writing time is more than the time required for chain computation for a table, then we use more than one DVD writers (running in parallel) to synchronize the chain computation and DVD write.

In the Hellman+DP method, a total of r tables are to be prepared. Let us denote the tables by $T_1, \ldots, T_r$. Consider the following algorithm.

1. Perform chain-computation for T_1
2. do in parallel
   perform chain-computation for T_2
   perform sorting for T_1
3. for $i = 3$ to r do in parallel
   perform chain-computation for T_i
   perform sorting for T_{i-1}
   initiate DVDwrite for T_{i-2}
4. end do
5. do in parallel
   perform sorting for T_r
   perform DVDwrite for T_{r-1}
6. perform DVDwrite for T_r

This algorithm pipelines the chain computation for T_i with the sorting of T_{i-1} and DVD writing for T_{i-2}. Under the reasonable assumption that the sorting time is at most the chain computation time, the major time component is at most the time required for chain-computation of r tables plus the time required to sort a table and write to DVD. The chain-computation itself has two tasks – parallel f-invocations and writing to high speed memory. These two tasks can also be pipelined as we discuss below.

Suppose n f-invocation units are available. Each table has a total of m (start point, end point) pairs. These are divided into m/n blocks $B_1, \ldots, B_{m/n}$, where each block contains n pairs. The n f-invocation units will be operating in parallel to produce one block.

1. Generate block B_1
2. for $i = 2$ to m/n do in parallel
   Generate block B_i
   Write block B_{i-1} to the table
3. end do
4. Write block $B_{m/n}$ to the table

Producing each block B_i requires $n \times t$ f-invocations. We may assume that the time for nt f-invocations is more than the time to write a block of n pairs to the table. Hence, the dominant time is the time required to compute all the chains in a table, which is time required for $m \times t$ f invocations.

Let us consider the time required to prepare all the tables. Using the above two algorithms, the total time will essentially be mrt f-invocations done in parallel by n f-invocation units. The cost has several components–cost of the f-invocation units; cost of input/output (I/O) units to write the blocks B_i's to the table; cost of storing r tables; and cost of the sorting unit. The dominant cost components are the cost of the f-invocation units and the cost of storage (memory).

On-Line Phase: There is a set of n f-invocation units, which produce DPs and write them to a buffer. There is another set of q I/O processors, which read from this buffer and perform lookup into the tables.

At a time, the q I/O processors are connected to q tables. Once lookup on q tables has been completed, the tables are moved out and a new set of q tables are moved into place. Thus the system operates as follows: Lookup on $T_1, \ldots T_q$ are completed, then lookup on $T_{q+1}, \ldots, T_{2q}$ are completed and so on. Once a table is replaced, it is never loaded again for this data set. Thus if we have D targets, then the lookup into table T_i for all these targets are completed before T_i is replaced.

In the above scenario, the following two tasks are performed in parallel.

- Apply f-invocations to the D targets and write the final DPs to the buffer.
- Read from the buffer; perform lookup in the q tables; and then replace the tables.

With a suitable design and choice of the parameters q and n, we can make the assumption that the above two tasks require approximately the same time. Under this assumption, the total time required in the online phase can be taken to be the total time for all the f-invocations. Further, in this architecture, the wiring cost is minimal and the dominant cost is the cost of implementing the f-invocation units. The task of an I/O processor is relatively simple and also we will have q to be much less than n. Hence, the cost of implementing q I/O processors can be ignored with respect to the cost of implementing the n f-invocation units.

We summarize the above discussion with respect to the cost and time measures.

Precomputation phase:

- Time: time required for rmt f-invocations;

- Cost: cost of implementing n parallel f-invocation units and cost of storing r tables.

Online phase:

- Time: time required for rtD f-invocations;
- Cost: cost of implementing n parallel f-invocation units.

7.2.1 Approximate Cost Analysis

At CHES 2005, Good and Benaissa [50] proposed a new FPGA design for AES using Xylinx Spartan-III (XC3S2000). The cost of a Xylinx Spartan-III FPGA device is around 12 USD (see [81]). The speed of encryption of the design in [50] is 25Gbps=0.2×2^{32} AES-128 encryption/sec. Under the assumption that the cost and time scale linearly as we move from one processor to n processors, the total processor cost for n processor units is $H_p = 12n$ USD and the speed is $n \times 0.2 \times 2^{32}$ AES-128 encryptions/sec. Let T_{sec} be the precomputation time in seconds. In T_{sec} time, the number of encryptions will be, $T_{sec} \times n \times 0.2 \times 2^{32}$.

For a general s-bit ($s \leq 128$) cipher, attacking $D = 2^d$ online data points, the number of encryptions required at the precomputation stage is 2^{s-d}. *We assume that for an s-bit cipher with $s \leq 128$, the throughput and chip area will remain the same as for the best AES-128 implementation.* Hence, in T_{sec} time, the number of encryptions will be, $T_{sec} \times n \times 0.2 \times 2^{32}$ and we get,

$$T_{sec} \times n \times 0.2 \times 2^{32} = 2^{s-d}. \tag{7.1}$$

Using $H_p = 12n$, we get $T_{sec}H_p = 60 \times 2^{s-d-32}$, or

$$2^{32}T_{sec}H_pD = 60N. \tag{7.2}$$

This gives a new type of trade-off involving precomputation time T_{sec}, processor cost H_p and data D whereas usual trade-off curve involves online time (number of f invocations), data and memory.

Memory Cost: We assume that one table will fit into one memory block. This simplifies the table management and in particular the design of the sorting algorithm. The latest cheap high density storage is DVD with storage capacity between 4 and 20 Gybtes. In the near future, SONY will launch the paper disk with capacity of 100 Gybtes. At present, we consider 4 Gybtes ($= 4 \times 2^{30}$ bytes) DVD with cost around 1 USD. Since, for a table we need $\frac{2sm}{8}$ bytes storage, so $\frac{2sm}{8} \leq 4 \times 2^{30}$, or,

$$sm \leq 2^{34}. \tag{7.3}$$

DVD write time: At present, we consider the writing time for a 4 Gybtes DVD is 1min[1] ($\approx 2^6$sec). The total number of f-invocations required for a single table is mt and the time required for this is $t_1 = \frac{mt}{n \times 0.2 \times 2^{32}}$. Let $W_1, \ldots W_k$ be the DVD writers which are running in parallel. At each of time $T = it_1$ for $i = 2, \ldots r+1$, one table will be ready for DVD write. At time $T = (i+1)t_1$, the table T_i will be assigned to W_i for $i = 1, 2, \ldots, k$. The next table T_{k+1} will be ready for DVD write at time $T = (k+2)t_1$. If we choose $kt_1 \geq 2^6$, then at time $T = (k+2)t_1$, W_1 will be free (since the time difference between the present time and the time when W_1 was assigned the table is $(k+2)t_1 - 2t_1 = kt_1 \geq 2^6 =$ DVD write time). So the table T_{k+1} will be assigned to W_1 for DVD write. In this way the next table will be assigned to W_2 and so on. So in this case all the processors and DVD writers will remain busy at all the time. Hence from the above discuss we have $kt_1 \geq 2^6$, or,

$$k \times \frac{mt}{n \times 0.2 \times 2^{32}} \geq 2^6. \tag{7.4}$$

or,

$$k \geq \frac{n2^{36}}{mt}. \tag{7.5}$$

Note the there are r tables to be written into r DVDs and each DVD write takes 2^6 seconds. The total time required for DVD write is $\frac{r2^6}{k}$ while k DVD writers are running in parallel. This time must be less than or equal to the precomputation time, i.e., $\frac{r2^6}{k} \leq T_{sec}$, or,

$$k \geq \frac{r2^6}{T_{sec}}. \tag{7.6}$$

We take $k = max\left(\frac{n2^{36}}{mt}, \frac{r2^6}{T_{sec}}, 1\right)$. Then k satisfies both the inequalities (7.5) and (7.6). At present, we consider the DVD writer cost is 100 USD each. The total DVD writer cost is $H_w = 100k$ USD. For r tables, memory cost is $H_m = r$ USD and total hardware cost $C = H_p + H_m + H_w = (12n + r + 100k)$ USD. Let us consider the following cases.

Case 1: $D = 1$ $(d = 0)$**.** We choose the Hellman table parameters as: $r = m = t = N^{1/3} = 2^{s/3}$. The total number of f invocations required at the online stage $= r \times t$ and the time required for this is $\tau_{sec} = \frac{r \times t}{n \times 0.2 \times 2^{32}}$, running n processors in parallel with the speed of 0.2×2^{32} encryptions/sec. Suppose we want to finish the precomputation within a day, then $T_{sec} = 2^{16.5}$ (the number of seconds in one day). From Equation (7.1), we get, $n = 5 \times 2^{s-48.5}$. For 1 year precomputation time, i.e., $T_{sec} = 2^{25}$ (the number of seconds in one year) we need the number of processors, $n = 5 \times 2^{s-57}$. In Table 7.1, we summarize some of the trade-offs with different values of s. If Table 7.1 is used to asses the strength of a block cipher then the considered block cipher should be used in CBC mode.

[1]For example writing speed of Samsung SH-W162 is 21.6MB/sec (16X).

Table 7.1: Trade-off for different values of s with $D = 1$ (r is the number of table, m is the number of rows in each table, t is the number of columns in each table, T_{sec} is the preprocessing time in seconds, n is the number of processors, H_p is the total cost for the processor in USD, H_m is the total memory cost in USD, k is the number DVD writers, H_w is the total cost for DVD writer in USD and τ_{sec} is the runtime) .

s	r	m	t	T_{sec}	n	H_p	H_m	k	H_w	τ_{sec}
56	2^{19}	2^{19}	2^{19}	$2^{16.5}$	2^{10}	$2^{13.6}$	2^{19}	$2^{8.5}$	2^{15}	< 1
64	2^{21}	2^{21}	2^{21}	$2^{16.5}$	2^{18}	$2^{21.6}$	2^{21}	2^{12}	$2^{18.5}$	< 1
80	2^{27}	2^{27}	2^{27}	2^{25}	2^{25}	$2^{28.6}$	2^{27}	2^{8}	$2^{14.5}$	< 1
86	2^{29}	2^{29}	2^{29}	2^{25}	2^{31}	$2^{34.6}$	2^{29}	2^{10}	$2^{16.5}$	< 1
96	2^{32}	2^{32}	2^{32}	$2^{38.3}$	2^{28}	2^{32}	2^{32}	1	$2^{6.5}$	80
128	2^{32}	2^{64}	2^{32}	$2^{70.3}$	2^{28}	2^{32}	2^{32}	1	$2^{6.5}$	80

Case 2: $D > 1$**.** The memory cost increases with the number of tables. We consider the following table parameters as in [25]: $r = \frac{N^{1/3}}{D} = 2^{\frac{s}{3}-d}$ and $m = t = N^{1/3} = 2^{s/3}$. The total number of f invocations required for online search $= rtD$ and the time required for this is $\tau_{sec} = \frac{r \times t \times D}{n \times 0.2 \times 2^{32}}$, running n processors in parallel with speed of 0.2×2^{32} encryption/sec. From Equation (7.1) we get, $n = \frac{5 \times 2^{s-d-32}}{T_{sec}}$. Table 7.2 summarizes some of the trade-offs with different values of s and $d = \frac{s}{4}$. The rows of the tables were calculated by fixing some of the parameters as mentioned below.

- Table 7.1 ($d = 0$)
  - –*rows 1 and 2*: Fix T_{sec} to be one day.
  - –*rows 3 and 4*: Fix T_{sec} to be one year.
  - –*rows 5 and 6*: Fix $H_p = H_m = 2^{32}$.
- Table 7.2 ($d = s/4$)
  - –*rows 1, 2 and 3*: Fix T_{sec} to be one day.
  - –*rows 4 and 5*: Fix T_{sec} to be one year.
  - –*row 6*: Fix T_{sec} to be one year and $H_m = 2^{32}$.
  - –*row 7*: Fix $H_p = H_m = 2^{32}$.

Discussion: From Tables 7.1 and 7.2, we conclude the following.

- 56-bit and 64-bit fs are completely insecure.

- For $d = 0$, with one year precomputation time and around 500M USD investment it is possible to crack 80-bit f in online time less than one second. For multiple targets (data) with $d = s/4$, attacking 80-bit becomes easier.
- For $s = 96$, and with a single data point, precomputation time is more than 4000 years. This is at a cost of around 1 billion USD. It is possible to bring down the precomputation time to a few years by increasing the cost to around 1 trillion dollar. Another problem is that the size of single table becomes large and barely fits in a single storage unit (see the bound of Equation (7.3)). In the presence of multiple data of the order of 2^{24} ($d = s/4$), the attack becomes reasonable. Hence 96-bit f also does not provide comfortable security.
- For $s = 128$, and with a single data point ($d = 0$), at least one of the parameters among (T_{sec}, H_p, H_m) become infeasible. Also even with $d = s/4 = 32$, one of the above parameters continues to remain infeasible. Increasing d beyond 32 is not practical. Hence 128-bit can be considered to provide adequate security margin in the forseeable future.

7.2.2 General Cost Model

For the general case, let us assume that C_1 and C_2 are the costs of one search unit and one storage unit respectively and ρ, δ are the rate of encryption and size of one storage unit in Gybtes respectively. Then Equation (7.1) becomes

$$T_{sec} \times n \times \rho = 2^{s-d} \tag{7.7}$$

and, $H_p = C_1 n$ and $H_m = C_2 r$. Using $H_p = C_1 n$ in Equation (7.7), we get $T_{sec} \times H_p \times \rho = 2^{s-d} C_1$, or $\rho T_{sec} H_p D = C_1 N$. Since for a table we need $\frac{2sm}{8}$ bytes storage, so $\frac{2sm}{8} \leq \delta \times 2^{30}$, or,

$$sm \leq \delta 2^{32}. \tag{7.8}$$

This constraint is required because we are fitting one table into one storage unit. Let ϵ be the DVD (storage) writing time. Then Equation (7.4) becomes $k \times \frac{mt}{n \times \rho} \geq \epsilon$, or, $k \geq \frac{n \times \rho \times \epsilon}{m \times t}$ and Equation (7.6) becomes $k \geq \frac{r\epsilon}{T_{sec}}$. Thus we take $k = max\left(\frac{n \times \rho \times \epsilon}{m \times t}, \frac{r\epsilon}{T_{sec}}, 1\right)$. Let C_3 be the cost of one DVD writer then $H_w = kC_3$USD.

7.2.3 Cost of Exhaustive Search

Cost analysis of exhaustive search is the same as the cost analysis for TMTO precomputation except the memory cost and DVD writer cost. Note that the processor cost H_p is required for

Table 7.2: Trade-offs for inverting a one-way function $f : \{0,1\}^s \to \{0,1\}^s$ for different values of s and $d = \frac{s}{4}$. Here r is the number of tables, m is the number of rows in each table, t is the number of columns in each table, T_{sec} is the preprocessing time in seconds, n is the number of processors, H_p is the total cost for the processors in USD, H_m is the total memory cost in USD, k is the number DVD writers, H_w is the total cost for DVD writer in USD and τ_{sec} is the runtime.

s	r	$m = t$	T_{sec}	n	H_p	H_m	k	H_w	τ_{sec}
80	$2^{6.7}$	$2^{26.7}$	$2^{16.5}$	2^{14}	$2^{17.6}$	$2^{6.7}$	1	$2^{6.5}$	845
86	$2^{6.7}$	$2^{28.6}$	$2^{16.5}$	2^{18}	$2^{21.6}$	$2^{6.7}$	1	$2^{6.5}$	776
96	2^{8}	2^{32}	$2^{16.5}$	2^{26}	$2^{29.6}$	2^{8}	1	$2^{6.5}$	320
96	2^{8}	2^{32}	2^{25}	2^{17}	$2^{20.6}$	2^{8}	1	$2^{6.5}$	$2^{17.3}$
128	2^{11}	2^{43}	2^{25}	2^{41}	$2^{44.6}$	2^{11}	1	$2^{6.5}$	$2^{15.3}$
128	2^{32}	2^{32}	2^{25}	2^{41}	$2^{44.6}$	2^{32}	2^{13}	$2^{19.5}$	$2^{25.3}$
128	2^{32}	2^{32}	2^{38}	2^{28}	2^{32}	2^{32}	1	$2^{6.5}$	$2^{38.3}$

both the exhaustive search and the TMTO precomputation. The factor H_m is additionally required for TMTO. Hence the trade-off for exhaustive search is same as Equation (7.2), i.e.,

$$2^{32}THD = 60N \tag{7.9}$$

where T denotes the time in seconds required for exhaustive search, H is the total processor cost and D is the number of data points. The general equation is the following.

$$\rho THD = C_1 N. \tag{7.10}$$

7.2.4 Rainbow Method

The rainbow method replaces t Hellman tables of size $m \times t$ each into a single rainbow table with size $m' \times t$, where $m' = mt$. Let us consider the case when $s = 56$ (DES). Then $N = 2^{56}$. Taking $m = t = N^{1/3}$, we get $m' = 2^{36}$, i.e. $sm' = 56 \times 2^{36} > 2^{36}$. This violates the constraint (7.3) ($sm' \leq 2^{34}$). Hence a single large rainbow table has to be stored into more than one memory block (the number of memory block will increase with the value of s). Then the sorting algorithm becomes much more complicated since it has now to sort the table which is split into different memory blocks. On the other hand, if we break the large single rainbow table into several small mutually disjoint rainbow tables, the online time increases by a factor of r, where r is the number of rainbow tables. In view of this, the rainbow method is not a good choice for hardware implementation. On the other hand, a hardware-software co-design approach can be adopted to handle the issue of large tables. However, it is not clear that such an approach can improve over the Hellman + DP method.

7.3 Application to Stream Ciphers with IV

Application of TMTO to stream ciphers with IV was analyzed in [57]. For an k-bit stream cipher using an l-bit IV, consider the following $(k+l)$-bit one-way function f:

$$(k\text{-bit key}, l\text{-bit IV}) \mapsto (k+l)\text{-bit keystream prefix.} \tag{7.11}$$

As pointed out in [57], inverting this one-way function f will provide the secret key. Since many IVs are used with the same key, and since IVs are public, one can apply multiple data TMTO to f, using D publicly available IVs. It has been shown in [57], that if the IV length is less than the key length, then the online time of TMTO is less than exhaustive key search. However, the precomputation time becomes 2^{k+l} which is more than exhaustive key search. On the other hand, the importance of IV in a TMTO attack matters more than its length. The effective length of IV is also crucial and has been pointed out in [57]. Let us consider this point in more detail.

The usual requirement on IV is that it should be a nonce, i.e., no value should be repeated. Thus, for example, one can fix a key and use the numbers 1, 2, ... as IVs for different messages. Suppose at most 2^λ messages are encrypted before a key change. The above appears to be a valid protocol for using stream cipher. The problem is that in this approach, only the last λ bits of the IV ever change. If we put the (arbitrary) restriction that at most 1000 messages are encrypted before a key change then $\lambda \approx 10$.

Suppose for a particular key we have access to the keystream segment for about $32 = 2^5$ messages. This gives $D = 2^5$. Since we know all the IVs, we can apply TMTO to a search space of size $N = 2^{k+10}$ with $D = 2^5$. The precomputation time is $N/D = 2^{k+10}/2^5 = 2^{k+5}$ and the online time then comes to around $2^{2(k+5)/3}$. If $k = 80$, then the precomputation can be completed in one year at a cost of 2^{32} USD and the online time is around a minute.

We interpret this situation as indicating that to resist TMTO, it is *not* sufficient to have IV length to be equal to key length. The protocol must ensure that the entire IV length is actually used. One simple way of doing this can be to choose a random nonce as IV for the first msg encrypted using a particular key and then use nonce + 1, nonce + 2, ... as IVs for subsequent msg.

7.3.1 GSM

Here we consider the security of GSM with respect to generic attack only. (A reviewer of the thesis has pointed out that GSM has other serious weaknesses.) For the GSM mobile phones [7], A5/3 stream cipher is used which is based on the iterated block cipher KASUMI. The cipher A5/3 uses 64-bit key and 22-bit effective IV size (others bits of IV are fixed).

Table 7.3: Trade-off of GSM for different values of D (r is the number of table, m is the number of rows in each table, t is the number of columns in each table, T_{sec} is the preprocessing time in seconds, n is the number of processors, H_p is the total cost for the processors in USD, H_m is the total memory cost in USD, k is the number DVD writers, H_w is the total cost for DVD writer in USD and τ_{sec} is the runtime).

D	r	$m = t$	T_{sec}	n	H_p	H_m	k	H_w	τ_{sec}
1	2^{29}	2^{29}	2^{25}	2^{31}	$2^{34.6}$	2^{29}	2^{10}	$2^{16.5}$	0.61
2^8	2^{21}	2^{29}	2^{25}	2^{23}	$2^{26.6}$	2^{21}	2^2	$2^{8.5}$	32
2^{16}	2^{13}	2^{29}	$2^{16.5}$	2^{24}	$2^{27.6}$	2^{13}	2^3	$2^{9.5}$	16
2^{22}	2^7	2^{29}	$2^{16.5}$	2^{18}	$2^{21.6}$	2^7	1	$2^{6.5}$	2^{10}

The following one-way function f from 86-bit to 86-bit has been considered in [57]:

$$(\text{64-bit key}, \text{22-bit effective IV}) \mapsto \text{86-bit keystream prefix}. \tag{7.12}$$

The size of the search space for exhaustive search attack is 2^{64}. From Table 7.1 (see row 2), we have the time for exhaustive search attack which is same as the precomputation time for TMTO to be $2^{16.5}$ sec = 1 day with a 2^{21} USD investment.

This is certainly doable and hence GSM mobile phone communications cannot be considered secure for more than a day. However, we can consider such communications to be secure for a shorter duration such as an hour. For example, a stock order is placed over a phone and the order is executed within an hour. Once the order is executed there is no need for secrecy. Thus, it is enough to ensure secrecy from the point of the order being placed and it being executed, which is at most an hour. If we consider only exhaustive search attacks, then such communication over GSM phones appears to be secure. However, if we apply TMTO to the search space of the function f defined in (7.12), then this might not be true.

The size of the search space f is $N = 2^{86}$. From Equation (7.1) we get, $n = \frac{5 \times 2^{86-d-32}}{T_{sec}}$ where 2^d is the data points availible to the attacker. Table 7.3 summarizes some of the trade-offs with different values of D where the table parameters are taken as: $r = \frac{N^{1/3}}{D} = 2^{\frac{s}{3}-d}$ and $m = t = N^{1/3} = 2^{s/3}$. From Table 7.3, we conclude that the A5/3 algorithm of GSM provides inadequate security assurance.

7.4 TMTO versus Exhaustive Search

In this section, we provide a comparison between TMTO and exhaustive search. Note that the size of the search space is same irrespective of whether we use TMTO or exhaustive search. The availability of multiple data (targets) bring down both the precomputation and

online time of TMTO. The same is true for exhaustive search which of course does not have separate online and offline phases.

1. TMTO is a chosen plaintext attack which can be converted to weak known plaintext or ciphertext only attack (see [56]). On the other hand, exhaustive search can be a ciphertext only attack [3].

2. The TMTO precomputation phase is also an exhaustive search. However it additionally requires the following,
   – Memory is required to store the table(s).
   – Memory access is needed to write the (start point, end point) pairs into the table.
   – Sorting is performed on the table(s) to sort the (start point, end point) pairs in the increasing order of the end points.

Possible advantages of TMTO over exhaustive search. Precomputation of TMTO is a one-time activity. Once completed, the online stage is much faster than exhaustive search for target available at different times. In case of exhaustive search, the entire attack has to be repeated every time.

Rechannelising the memory cost of TMTO into processor cost for exhaustive search does not significantly reduce the exhaustive search time. To justify this, we consider a TMTO which can find the key in time τ_{sec} with precomputation time T_{sec}, processor cost H_p and memory cost H_m. We also consider an exhaustive search attack which can find the key in time T with the processor cost $H = H_p + H_m$. Then we will have the following three cases:

Case 1: If $H_p > H_m$, then $H \approx H_p$. Equations (7.2) and (7.9) yield $T = T_{sec} > \tau_{sec}$.

Case 2: If $H_p \approx H_m$, then $H \approx 2H_p$. Equations (7.2) and (7.9) yield $T = \frac{1}{2}T_{sec} > \tau_{sec}$.

Case 3: If $H_p < H_m$, then $H \approx H_m$. This case occurs only when the key size is small. For instant consider $s = 56$. Then from Table 7.1, we see that $H_p = 2^{13.6}$ and $H_m = 2^{19}$. So $H \approx 2^{19}$ and from Equation (7.9), we get $T = 480$ sec $> 0.31 = \tau_{sec}$

The above three cases show that the exhaustive search time will be more than the online search time for TMTO. Hence, transferring the cost of memory to the processor and performing only exhaustive search does not bring down search time to make it comparable to online phase of TMTO.

Chapter 8

Concluding Remarks

Our contribution is in analyzing TMTO attacks, application of LFSRs in TMTO attacks and hardware design for TMTO attack. This work can be broadly classified into three parts. The first part (Chapter 4) consists of a unified approach to the analysis of TMTO method in the presence of multiple data. The article [24] falls in this category. The second part of the thesis (Chapters 5 and Appendix) is devoted to the application of LFSR sequences in cryptologic algorithms. This covers our articles [72, 73]. Finally, the third part of the thesis (Chapters 6 and 7) provides a detailed architecture for implementing TMTO attack and the effectiveness of TMTO and exhaustive search attacks. We include articles [74, 75] in these chapters.

Possible Future Work: Chapter 4 identifies new single table attacks. It might be interesting to look for practical situations where such attacks are applicable.

An LFSR based approach to the function generation and the start point generation is outlined in Chapter 5. The discussion is with respect to the rainbow attack, though it is clear that the same discussion also holds for the Hellman + DP attack. In fact, the architecture in Chapter 6 adopts the LFSR based approach.

The main future work is to validate the different hardware architectures through a hardware synthesis tool. Such a simulation will provide estimates of the number of gates, the clock frequency, routing costs, power consumption, mean time between failures and other relevant parameters. This may lead to possible alterations of the design as well as provide a better understanding of different implementation issues. Not having the relevant resources we have not been able to take up this work. Finally, we would like to say that we will be very satisfied if, in the future, it is possible to build (either by us or by others) a cryptanalytic machine based on our ideas.

Bibliography

[1] ETSI/SAGE. Specification of the A5/3 Encryption Algorithms for GSM and EDGE, and the GEA3 Encryption Algorithm for GPRS, Document 1: A5/3 and GEA 3 Specifications, ETSI/SAGE, May 2002.

[2] 3rd Generation Partnership Program. 3GPP home page. `http://www.3gpp.org/`

[3] Electronics Frontier Foundation. *Cracking DES*, O'Reilly and Associates, 1998.

[4] ECRYPT. Call for stream cipher primitives. Version 1.2, Feb. 2004. `http://www.ecrypt.eu.org/stream/`

[5] National Bureau of Standards. Data Encryption Standard, U.S. Department of Commerce, FIPS pub. 46, 1977.

[6] National Institute of Standards and Technology (2001). Advanced Encryption Standard. Federal Information Processing Standard, FIPS-197.

[7] 3GPP TS 55.215 V6.2.0 (2003-09), A5/3 and GEA3 Specifications. Available at `http://www.gsmworld.com`

[8] Consortium for Efficient Embedded Security. Efficient Embedded Security Standards (EESS) #1. Version 2.0, June 2003. Available at `http://www.ceesstandards.org/`

[9] RainbowCrack: General Propose Implementation of Rainbow Method. `http://www.antsight.com/zsl/rainbowcrack/`.

[10] H. R. Amirazizi and M. E. Hellman. "Time-Memory-Processor Trade-offs". *IEEE Transactions on Information Theory*, vol. 34, no. 3, pp. 505-512, 1988.

[11] G. Avoine, P. Junod and P. Oechslin. "Time-Memory Trade-Offs: False Alarm Detection Using Checkpoints". Proceedings of Indocrypt 2005, LNCS 3797, pp. 183-196, 2005.

[12] S. H. Babbage. "Improved Exhaustive Search Attacks on Stream Ciphers". European Convention on Security and Detection, IEE Conference publication, no. 408, pp. 161-166, IEE, 1995.

[13] E. Barkan, E. Biham and N. Keller. "Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication". Proceedings of Crypto 2003, LNCS 2729, pp. 600-616, 2003.

[14] E. Barkan, E. Biham and A. Shamir. "Rigorous Bounds on Cryptanalytic Time/Memory Tradeoffs". Proceedings of Crypto 2006, LNCS 4117, pp. 1-21 , 2006.

[15] K. Batcher. "Sorting Networks and Their Applications". Proceedings of SJCC'68, pp. 307-314, 1968.

[16] D. J. Bernstein. "Understanding Brute Force". `http://cr.yp.to/papers.html#bruteforce`, 2005.

[17] E. Biham. "How to Decrypt or Even Substitute DES-Encrypted Messages in 2^{28} Steps". *Information Processing Letters*, vol. 84, pp. 117-124, 2002.

[18] E. Biham. "New Types of Cryptanalytic Attacks Using Related Keys". *Journal of Cryptology*, vol. 7, no. 4, pp. 229-246, 1994.

[19] E. Biham, A. Biryukov, and A. Shamir. "Cryptanalysis of Skipjack Reduced to 31 Rounds using Impossible Differentials". Proceedings of Eurocrypt 1999, LNCS 1592, pp. 12-23, 1999.

[20] E. Biham, A. Biryukov and A. Shamir. "Miss in the Middle Attacks on IDEA and Khufu". Proceedings FSE 1999, LNCS 1636, pp. 124-138, 1999.

[21] E. Biham and A. Shamir. *Differential Cryptanalysis of the Data Encryption Standard*, Springer Verlag, 1993.

[22] A. Biryukov. "Block Cipher and Stream Cipher: The State of the Art". Cryptology ePrint Archive, Report 2004/207, `http://eprint.iacr.org/2005/207`, 30 Jun 2004.

[23] A. Biryukov. "Some Thoughts on Time-Memory-Data Tradeoffs". Cryptology ePrint Archive, Report 2005/207, `http://eprint.iacr.org/2005/207`, 30 June, 2005.

[24] A. Biryukov, S. Mukhopadhyay and P. Sarkar. "Improved Time-Memory Trade-offs with Multiple Data". Proceedings of SAC 2005, LNCS 3897, pp. 110-127, 2006.

[25] A. Biryukov and A. Shamir. "Cyptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers". Proceedings of Asiacrypt 2000, LNCS 1976, pp. 1-13, 2000.

[26] A. Biryukov and A. Shamir and D. Wagner. "Real Time Cryptanalsis of A5/1 on a PC". Proceedings of FSE 2000, LNCS 1978, pp. 1-18, 2000.

[27] A. Biryukov and D. Wagner. "Slide Attacks". Proceedings FSE 1999, LNCS 1636, pp. 245-259, 1999.

[28] M. Boesgaard, M. Vesterager, T. Pedersen, J. Christiansen and O. Scavenius. "Rabbit: A New High-Performance Stream Cipher". Proceedings of FSE 2003, LNCS 2887, pp. 307-329, 2000.

[29] J. Borst. "Block Ciphers: Design, Analysis and Side-Channel Analysis". *Doctoral dissertation*, K.U.Leuven, September 2001.

[30] J. Borst, B. Preneel and J. Vandewalle. "On the Time-Memory Tradeoffs Between Exhaustive Key Search and Table Precomputing". Proceedings of the 19th Symposium on Information Theory in the Benelux, WIC, pp. 111-118, 1998.

[31] J. Borst, B. Preneel and J. Vandewalle. "A Time-Memory Tradeoff using Distinguished Points". Technical report ESAT-COSIC Report 98-1, Department of Electrical Engineering, Katholieke Universiteit Leuven, 2001.

[32] J. Borst, B. Preneel and J. Vandewalle. "Linear Cryptanalysis of RC5 and RC6". Proceedings of FSE 1999, LNCS 1636, pp. 16-30, 1999.

[33] S. Burman and P. Sarkar. "An Efficient Algorithm for Software Generation of Linear Binary Recurrences". *Applicable Algebra in Engineering, Communication and Computing*, vol. 15, Issue 3/4, December 2004.

[34] S. Contini, R. Rivest, M. Robshaw and Y. Yin. "Improved Analysis of Some Simplified Variants of RC6". Proceedings FSE 1999, LNCS 1636, pp. 1-15, 1999.

[35] D. Coppersmith, H. Krawczyk and Y. Mansour. "The Shrinking Generator". Proceedings of Crypto 1993, LNCS 773, pp. 22-39, 1993.

[36] N. Courtois. "Fast Algebraic Attacks on Stream Ciphers with Linear Feedback". Proceedings of Crypto 2003, LNCS 2729, pp. 176-194, 2003.

[37] N. Courtois and W. Meier. "Algebraic Attacks on Stream Ciphers with Linear Feedback". Proceedings of Eurocrypt 2003, LNCS 2656, pp. 345-359, 2003.

[38] C. De Cannière, J. Lano, and B. Preneel. "Comment on the Rediscovery of Time Memory Data Tradeoffs". Available as a link on the *ECRYPT Call for Stream Cipher Primitives* [4] page version 1.3, April 2005.

[39] D. E. Denning. *Cryptography and data security*. Addison-Wesley, 1982.

[40] Y. Desmedt. "An Exhaustive Key Search Machine Breaking One Million DES Keys". Presented at Eurocrypt 1987. See Chapter 9 of [3].

[41] W. Diffie and M. Hellman. "Exhaustive Cryptanalysis of the NBS Data Encryption Standard". *Computer*, vol. 10, no. 6, pp. 74-84, June 1977.

[42] W. Diffie and M. Hellman. "Privacy and Authentication: An Introduction to Cryptography". Proceedings of the IEEE, vol. 67, pp. 397-427, 1979.

[43] H. Eberle. "A High-Speed DES Implementation for Network Applications". Proceedings of Crypto 1992, LNCS 740 , pp 527-545, 1993.

[44] P. Ekdahl and T. Johansson. "A New Version of the Stream Cipher SNOW". Proceedings of SAC 2002, LNCS 2595, pp 47-61, 2002.

[45] A. Fiat and M. Naor. "Rigorous Time/Space Tradeoffs for Inverting Functions". STOC 1991, pp. 534-541, 1991.

[46] J. Dj. Golić. "Cryptanalysis of Alleged A5 Stream Cipher". Proceedings of Eurocrypt 1997, LNCS 1233, pp. 239–255, 1997.

[47] H. Gilbert, H. Handschuh, A. Joux and S. Vaudenay. "A Statistical Attack on RC6". Proceedings FSE 2000, LNCS 1978, pp. 64-74, 2000.

[48] I. Goldberg and D. Wagner. *Architectural Considerations for Cryptanalytic Hardware*. Chapter 10 of [3], also available at `http://citeseer.ist.psu.edu/goldberg96architectural.html`.

[49] O. Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, Cambridge, 2001.

[50] T. Good and M. Benaissa. "AES on FPGA from the Fastest to the Smallest". Proceedings of CHES 2005, LNCS 3659, pp 427-440, 2005.

[51] Z. Gutterman and D. Malkhi. "Hold your Sessions: An Attack on Java Session-id Generation". Proceedings of CT-RSA 2005, LNCS 3376, pp. 44-57, 2005.

[52] S. Halevi, D. Coppersmith and C. S. Jutla. "Scream: A Software-Efficient Stream Cipher". Proceedings of FSE 2002, LNCS 2365, pp 195-209, 2002.

[53] I. Hamer and P. Chow. "DES Cracking on the Transmogrifier 2a". Proceedings of CHES 1999, LNCS 1717, pp 13-24, 1999.

[54] H. Handschuh and H. Gilbert. "χ^2 Cryptanalysis of the SEAL Encryption Algorithm". Proceedings FSE 1997, LNCS 1267, pp. 1-12, 1997.

[55] P. Hawkes and G. Rose. "Primitive Specification and Supporting Documentation for SOBER-t32". *First Open* NESSIE Workshop, submission to NESSIE, 2000.

[56] M. Hellman. "A Cryptanalytic Time-Memory Trade-off". *IEEE Transactions on Information Theory*, vol. 26, pp. 401-406, 1980.

[57] J. Hong and P. Sarkar. "New Applications of Time Memory Data Tradeoffs". Proceedings of Asiacrypt 2005, LNCS 3788, pp. 353-372, 2005.

[58] I.J. Kim and T. Matsumoto. "Achieving Higher Success Probability in Time-Memory Trade-Off Cryptanalysis without Increasing Memory Size". *TIEICE: IEICE Transactions on Communications/Electronics/Information and System*, pp. 123-129, 1999.

[59] S. Kumar, C. Paar, J. Pelzl, Gerd Pfeiffer and Manfred Schimmler. "Breaking Ciphers with COPACOBANA - A Cost-Optimized Parallel Code Breaker". Proceedings of CHES 2006, LNCS 4249, pp. 101-118, 2006.

[60] K. Kusuda and T. Matsumoto. "Optimization of Time-Memory Trade-Off Cryptanalysis and its Application to DES, FEAL-32 and Skipjack". *IEICE Transactions on Fundamentals*, vol. E79-A, no. 1, pp. 35-48, 1996.

[61] X. Lai. "Higher Order Derivatives and Differential Cryptanalysis". *Communication and Cryptography*, Kluwer Academic Publishers, pp. 227-233, 1994.

[62] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and their Applications*. Cambridge University Press, Cambridge, pp. 189-249, 1994 (revised edition).

[63] H. Lipmaa, P. Rogaway and D. Wagner. "Comments to NIST Concerning AES Modes of Operations: CTR-Mode Encryption".

[64] M. Matsui. "Linear Cryptanalysis Method for DES Cipher". Proceedings of Eurocrypt 1993, LNCS 765, pp. 386- 397, 1993.

[65] M. Matsui. "The First Experimental Cryptanalysis of the Data Encryption Standard". Proceedings of Crypto 1994, LNCS 839, pp. 1-11, 1994.

[66] T. Matsumoto, I. Kim and T. Hara. "Methods to Reduce Time and Memory in Time-Memory Trade-Off, FEAL-32 and Skipjack". IEICE Technical Report, ISEC97-10, May 26, 1997.

[67] D. McGrew. "Segmented Integer Counter Mode: Specification and Rationale". Submitted to NIST Modes of Operation Workshop, October, 2000.

[68] A.J. Menezes, P.C. van Oorschot and S.A. Vanstone. *Handbook of Applied Cryptography*, CRC, Boca Raton, 2001.

[69] N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede. "Cracking Unix passwords using FPGA platforms". Proceedings of SHARCS'05, 2005.

[70] R. C. Merkle and M. E. Hellman. "Hiding Information and Signatures in Trapdoor Function". *IEEE Transaction on Information Theory*, no. 24, pp. 525-530, 1978.

[71] S. Mukhopadhyay. "Time/Memory Trade-Off: A Survey". *Journal of the Indian Statistical Association (JISA)*, Special Issue on Statistics in Cryptology, Volume 42, No. 2, ISSN 0537-2585, December, 2004.

[72] S. Mukhopadhyay and P. Sarkar. "Application of LFSRs in Time/Memory Trade-Off Cryptanalysis". Proceedings of WISA 2005, LNCS 3786, pp. 25-37, 2006.

[73] S. Mukhopadhyay and P. Sarkar. "Application of LFSRs for Parallel Sequence Generation in Cryptologic Algorithms". Proceedings of Applied Cryptography and Information Security 2006 (ACIS'06) in conjunction with ICCSA 2006, LNCS 3982, pp. 426-435, 2006.

[74] S. Mukhopadhyay and P. Sarkar. "Hardware Architecture and Trade-offs for Generic Inversion of One-way Functions". 2006 IEEE International Symposium on Circuits and Systems (ISCAS 2006), 2006. Full version available as (Indian Statistical Institute Technical Report No. ASD/2006/2).

[75] S. Mukhopadhyay and P. Sarkar. "On the Effectiveness of TMTO and Exhaustive Search Attacks". Proceedings of IWSEC 2006, LNCS 4266 , pp. 337-352, 2006.

[76] P. Oechslin. "Making a Faster Cryptanalytic Time-Memory Trade-Off". Proceedings of Crypto 2003, LNCS 2729, pp. 617-630, 2003.

[77] P.C. van Oorschot and M.J. Wiener. "Parallel Collision Search with Cryptanalytic Applications". *Journal of Cryptology*, vol. 12, no. 1, pp. 1-28, Winter 1999.

[78] S. C. Pohlig and M. E. Hellman. "An Improved Algorithm for Computing Logarithms Over GF(p) and its Cryptographic significance". *IEEE Transactions on Information Theory*, vol. IT-24, pp. 106-110, 1978.

[79] J.J. Quisquater and J.P. Delescaille. "How Easy is Collision Search? Application to DES". Proceedings of Eurocrypt 1989, LNCS 434, pp 429-434, 1990.

[80] J.J. Quisquater and Y. Desmedt, "Chinese Lotto as an Exhaustive Code-Breaking Machine". in *Computer*, vol. 24, issue 11 (November 1991), pp. 14-22, 1991.

[81] J.J. Quisquater and F.X. Standaert. "Exhaustive Key Search of the DES: Updates and Refinements". Presented at SHARCS'05, 2005.

[82] J.J. Quisquater, F.X. Standaert, G. Rouvroy, J.P. David and J.D. Legat. "A Cryptanalytic Time-Memory Tradeoff: First FPGA Implementation". Proceedings of FPL 2002, LNCS 2438, pp. 780-789, 2002.

[83] R. L. Rivest, M. J. B. Robshaw and Y. L. Yin. "RC6 as the AES". AES Candidate Conference 2000, pp. 337-342, 2000.

[84] G. G. Rose and P. Hawkes. "Turing: A Fast Stream Cipher". Proceeding of FSE 2003, LNCS 2887, pp 290-306, 2003.

[85] B. Roy and S. Mukhopadhyay. "Statistical Cryptanalysis On Block Cipher". *Journal of the Indian Society for Probability and Statistics*, Vol. 7, pp. 33-50, 2003.

[86] R. Rueppel. *Stream Cipher. Contemporary cryptology: The science of information integrity*, G. Simmons, ed., IEEE Press, pp. 65-134, 1991.

[87] M.-J. O. Saarinen. "A Time-Memory Trade-off Attack Against LILI-128". Proceedings of FSE 2002, LNCS 2365, pp. 231-236, 2002.

[88] R. Schroeppel and A. Shamir. "A $T = O(2^{(n/2)})$, $S = O(2^{(n/4)})$ Algorithm for Certain NP-Complete Problems". *SIAM Journal of Computing*, vol. 10, no. 3, pp. 456-464, 1981.

[89] T. Shimoyama, M. Takenaka and T. Koshiba. "Multiple Linear Cryptanalysis of a Reduced Round RC6". Proceedings of FSE 2002, LNCS 2365, pp. 76-88, 2002.

[90] T. Shimoyama, M. Takeuchi and J. Hayakawa (2000). "Correlation Attack to the Block Cipher RC5 and Simplified Variants of RC6". 3rd AES Candidate Conference, 2000.

[91] S. Singh. *The Code Book: The Secret History of Codes and Code-Breaking*. Fourth Estate, London, 1999.

[92] F.X. Standaert, G. Rouvroy, J.J. Quisquater and J.D. Legat. "A Time-Memory Tradeoff Using Distinguished Points: New Analysis & FPGA Results". Proceedings of CHES 2002, LNCS 2523, pp. 593-609, 2002.

[93] D. Stinson. *Cryptography: Theory and Practice*, Second Edition, CRC press company, 2002.

[94] E. Tromer. *Special-purpose cryptanalytic devices – an annotated taxonomy*, http://www.wisdom.weizmann.ac.il/∼tromer/

[95] S. Vaudenay. "Decorrelation: A Theory for Block Cipher Security". *Journal of Cryptology*, vol. 16, no. 4, pp. 249-286, 2003.

[96] D. Wagner. "The Boomerang Attack". Proceedings of FSE 1999, LNCS 1636, pp. 156-170, 1999.

[97] D. Watanabe, A. Biryukov and C. De Cannière "A Distinguishing Attack of SNOW 2.0 with Linear Masking Method". Proceedings of SAC 2003, LNCS 3006, pp 222-233, 2003.

[98] M. J. Wiener. "The Full Cost of Cryptanalytic Attacks". *Journal of Cryptology*, vol. 17, no. 2, pp. 105-124, 2004.

[99] M. J. Wiener. "Efficient DES Key Search". Crypto 1993 (rump session presentation), Santa Barbara, California, USA, August 1993. Reprint in Practical Cryptography for Data Internetworks, William Stallings editor, IEEE Computer Society Press, pp. 31-79, 1996.

Appendix A

Other Applications of LFSRs for Parallel Sequence Generation in Cryptologic Algorithms

A.1 Introduction

In continuation of Chapter 5, we consider the problem of efficiently generating sequences in hardware for use in certain other cryptographic algorithms. The conventional method of doing this is to use a counter. We show that sequences generated by linear feedback shift registers (LFSRs) can be tailored to suit the appropriate algorithms. As a result, we are able to suggest improvements to the design of DES Cracker built by the Electronic Frontier Foundation in 1998 and present an improved parallel hardware implementation of a variant of the counter mode of operation of a block cipher. The material in this Chapter is based on Sections 3, 4 and 6 of [73].

The first work to suggest the use of LFSRs in exhaustive key search was by Wiener [99]. In [99], Wiener described a detailed design for a special purpose hardware for cracking DES. The suggestion for generating candidate keys was to use an LFSR having $g(x) = x^{56} + x^7 + x^4 + x^2 + 1$ as the primitive characteristic polynomial. The generation of keys starting from a key k was to use the sequence $k, kx \ mod \ g, kx^2 \ mod \ g$ and so on. This idea is also used in [53]. The idea of using parallel LFSR sequences was suggested by Goldberg and Wagner [48] and was used by them in 1996 for an FPGA-based DES keysearch engine.

In Section A.4, we consider the possibility of using LFSRs for use in the counter mode of operation of a block cipher. As pointed out by a reviewer of the thesis, this was earlier suggested by McGrew [67]. Compared to [67], we provide more details on the possible use

of parallelism in this context.

We consider the following cryptologic algorithms which require the generation of a sequence of s-bit vectors for parallel implementation.

- Exhaustive search algorithms like the DES Cracker (described in Section 2.3.1) employ a high degree of parallelism. Hence, the requirement is to generate in parallel a set of pairwise disjoint sequences of s-bit vectors whose union is the set $\{0,1\}^s$.
- *Counter Mode of Operation* [63] is a mode of operation of a block cipher, which converts the block cipher into an additive stream cipher. In this mode of operation, one requires to generate a long non-repeating sequence of s-bit values.

The first one is cryptanalytic algorithm, while the second one is a cryptographic algorithm. Implementations of the above two algorithms use a counter to generate the required sequences.

In this Chapter, we explore the possibility of using sequences obtained from linear feedback shift registers (LFSRs) for the hardware implementation of the above algorithms. In each case, we show how LFSR sequences can be tailored for use in the respective algorithms. This leads us to suggest changes to the DES Cracker which simplify the design as well as reduce the time and to describe a variant of the classical counter mode of operation of a block cipher.

A.2 Parallel Sequence Generation

In this section, we consider the problem for parallel implementation of TMTO attacks. Suppose there are n processors available at the pre-computation phase. Then it is required to generate parallel independent pseudo random sequences of s-bit start points in the pre-computation phase, i.e., the problem is as follows.

- Generate n parallel and pairwise disjoint sequences of s-bit strings such that the union of these n sequences is the set of all (non-zero) s-bit strings.

We provide a simple LFSR based strategy for solving the above problem. Let $L = (s, p(x))$ be an s-bit LFSR where $p(x)$ is a primitive polynomial of degree s over GF(2). Let $2^s - 1 = \tau \times n + r = (\tau + 1)r + \tau(n - r)$ where $0 \leq r < n$. Let $n_1 = n - r$, $n_2 = r$ and note that $\tau = \lfloor \frac{2^s-1}{n} \rfloor$. Let S_0 be any nonzero s-bit string and for $t \geq 1$, we define $S_t = S_0 M^t$, where M is the state transition matrix of L. Further, let $T_0 = S_{n_1 \tau}$ and for $t \geq 1$, $T_t = T_0 M^t = T_{t-1} M$.

Also let $\tau' = \lceil \frac{2^s-1}{n} \rceil$. Define n sequences as follows.

$$\begin{array}{llll} \mathcal{S}_0: & S_0, S_1, \ldots, S_{\tau-1}; & \mathcal{T}_0: & T_0, T_1, \ldots, T_{\tau'-1}; \\ \mathcal{S}_1: & S_\tau, S_{\tau+1}, \ldots, S_{2\tau-1}; & \mathcal{T}_1: & T_{\tau'}, T_{\tau'+1}, \ldots, T_{2\tau'-1}; \\ \vdots & & \vdots & \\ \mathcal{S}_{n_1-1}: & S_{(n_1-1)\tau}, \ldots, S_{n_1\times\tau-1}; & \mathcal{T}_{n_2-1}: & T_{(n_2-1)\tau'}, \ldots, T_{n_2\times\tau'-1}. \end{array} \tag{A.1}$$

The $\mathcal{S}$ sequences are of length τ, while the $\mathcal{T}$ sequences are of length $\tau' \geq \tau$. Note that, $T_{n_2\times\tau-1} = T_0 M^{n_2\times\tau'-1} = S_0 M^{n_1\tau} M^{n_2\times\tau'-1} = S_0 M^{n_1\tau+n_2\times\tau'-1} = S_0 M^{2^s-2} = S_{2^s-2}$. Since $p(x)$ is primitive, the sequence $S_0, S_1, \ldots, S_{n_1\times\tau-1}, T_0, T_1, \ldots, T_{n_2\times\tau'-1}$ consists of all non-zero s-bit vectors. This ensures that the sequences $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_{n_1-1}, \mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_{n_2-1}$ are pairwise disjoint. Thus, we obtain a solution to the problem mentioned above. We now consider the problem of actually generating the sequences in hardware.

Implementation: Let $L_0, \ldots, L_{n-1}$ be n implementations of the LFSR L. Hence, each L_i has $p(x)$ as its connection polynomial. The initial conditions for $L_0, \ldots, L_{n_1-1}$ are $S_0, S_\tau, \ldots, S_{(n_1-1)\tau}$ respectively and the initial conditions for $L_{n_1}, \ldots, L_{n-1}$ are $T_0, T_{\tau'}, \ldots, T_{(n_2-1)\tau'}$ respectively. At any point of time, the current states of the L_i's provide the current values of the $\mathcal{S}$ and the $\mathcal{T}$ sequences. All the L_i's operate in parallel, i.e., they are all clocked together and hence the next states of the $\mathcal{S}$ and the $\mathcal{T}$ sequences are generated in parallel. The total hardware cost for implementing the n LFSRs consists of $n \times s$ flip-flops and $n \times (\mathsf{wt}(p(x) - 1)$ 2-input XOR gates. With this minimal hardware cost, the parallel generation of the $\mathcal{S}$ and the $\mathcal{T}$ sequences become possible.

Obtaining the initial conditions: We explain how to obtain the initial condition for the n LFSRs. Let $M_1 = M^\tau$ and $M_2 = M^{\tau+1} = M \times M_1$. Then $S_{i\tau} = S_0 M^{i\tau} = S_{(i-1)\tau} \times M^\tau = S_{(i-1)\tau} \times M_1$. Now $T_0 = S_{(n_1-1)\tau} \times M_1$ and $T_{j\tau'} = T_{(j-1)\tau'} \times M_2$. Once we know M_1 and M_2 it is easy to find all the $S_{i\tau}$'s and $T_{j\tau'}$'s. Computing M_1 requires a matrix exponentiation which as mentioned before requires $2 \log \tau \leq 2s$ matrix multiplications. Obtaining M_2 from M_1 requires one matrix multiplication. After M_1 and M_2 have been obtained, computing the initial conditions require a total of n vector-matrix multiplications. These initial conditions are obtained once for all in an offline phase. These are then pre-loaded into the LFSRs and do not need to re-computed during the actual generation of the parallel sequences.

A.3 Application 1: The DES Cracker

The DES Cracker is described in Section 2.3.1. Here we briefly mention some of the salient points. In the design of DES cracker (described in section 2.3.1), a computer drives 2^{16}

search units. The search units are parallel hardware units while the computer provides a central control software. The key space is divided into segments and each search unit searches through one segment. For each candidate key, a search unit does the following. Let x be the current candidate key. A search unit decrypts the first ciphertext using x and checks whether the resulting plaintext is "interesting". If yes, then it decrypts the second ciphertext using x and checks if it is also interesting. (The search unit considers a plaintext to be interesting if all its 8 bytes are ASCII.) If both the plaintexts are found to be interesting then the (key, plaintext) pair is passed to a computer to take the final decision. The search unit then adds one to x to obtain the next candidate key.

Recall that in DES, the message and cipher block size is 64 bits while the key size is 56 bits. In each search unit, a counter (and an adder) generates the candidate keys. A 32-bit counter is used to count through the bottom 32 bits of the key. The reason for using a 32-bit adder is that it is cheaper to implement than a 56-bit adder. The top 24 bits of the key are loaded onto the search unit by the computer. After completing 2^{32} keys with a fixed value of the 24 bits, a search unit sends a signal to the computer. The computer stops the chip; resets the key counter; puts a new value in the top 24 bits; and the search starts once more with this new 24-bit value.

A.3.1 LFSR Based Solution

We describe an alternative LFSR based solution for candidate key generation in the DES cracker. This solution is based on the parallel sequence generation described in Section A.2. The number of parallel search units $n = 2^{16}$, while $s = 56$. Thus, $\tau = 2^{40} - 1$, $\tau' = 2^{40}$, $n_1 = 1$ and $n_2 = 2^{16} - 1$.

Choose the LFSR L such that $p(x)$ is the primitive pentanomial $(x^{56} + x^{22} + x^{21} + x + 1)$. Choose S_0 to be an arbitrary non-zero 56-bit value and compute the values $T_0, \ldots, T_{n_2-1}$ using the method of Section A.2. The total number of 56×56 binary matrix multiplications required is at most $2 \times s + 1 = 113$. Additionally, one has to compute a total of 2^{16} multiplications of a 56-bit vector with a 56×56 binary matrix. Even with a straightforward software implementation, the entire computation can be completed within a few hours. The initial condition of the LFSR in the first search unit is set to S_0, while the initial conditions for the LFSRs in the other search units are set to $T_0, T_1, \ldots, T_{n_2-1}$. Computing the initial conditions can be considered to be part of the design stage activity.

In our design, each search unit of the DES cracker has its own implementation of L. This implementation requires n flip-flops and only four 2-input XOR gates. Each search unit now generates the candidate keys independently of the computer and also independently of each other. To obtain the next candidate key, it simply clocks its local LFSR once and uses the state of the LFSR as the candidate key. The first search unit does this for $\tau = 2^{40} - 1$ steps

while the other search units do this for $\tau' = 2^{40}$ steps. This ensures that all non-zero keys are considered, with the all-zero key being considered separately.

A.3.2 Comparison to the Counter Based Solution

There are two ways in which the LFSR based solution improves over the counter based solution.

- There are 2^{16} search units. In the counter based solution, each search unit sends an interrupt signal to the computer after completing an assigned key segment. Thus, the computer needs to handle a total of 2^{24} interrupts from all the search units. This may cause some delay. In the LFSR based solution, candidate key generation is done solely by the search unit without any involvement from the computer.
- In the counter method, each search unit requires a 32-bit adder for a total of 2^{16} such adders. In contrast, in the LFSR based solution, the circuitry for generating the next candidate key consists of only 4 XOR gates per search unit.

A.4 Application 2: Counter Mode of Operation

In 1979, Diffie and Hellman [42] introduced the counter mode (CTR mode) of operation for a block cipher. This mode actually turns a block cipher into an additive stream cipher. See [63] for more details about CTR mode. Let $E_x()$ be $2s$-bit block cipher. The pseudo-random sequence is produced as follows:

$$E_x(\mathsf{nonce}||S_0)||E_x(\mathsf{nonce}||S_1)||E_x(\mathsf{nonce}||S_2)||\ldots,$$

where nonce is an s-bit value and $S_0, S_1, \ldots$ is a sequence of s-bit values. The security requirements are the following.

1. The nonce is changed with each message such that the same (key,nonce) pair is never repeated.
2. The sequence $S_0, S_1, S_2, \ldots$ is a non-repeating sequence.

Usual implementations define $S_i = \mathsf{bin}_s(i)$, where $\mathsf{bin}_s(i)$ is the s-bit representation of the integer i. With this definition, the sequence S_i can be implemented using a counter.

Hardware implementation of CTR mode can incorporate a high degree of parallel processing. The inherent parallelism is that each $2s$-bit block of pseudo-random bits can be

produced in parallel. Suppose we have n processors $P_0, P_1, \ldots, P_{n-1}$ where each processor is capable of one block cipher encryption. Processor P_i encrypts the values $\mathsf{nonce}||S_i$, $\mathsf{nonce}||S_{n+i}$, $\mathsf{nonce}||S_{2n+i}, \ldots$. If S_i is defined to be $\mathsf{bin}_s(i)$, then there are two ways of generating the sequence.

Single adder: With a single adder, the algorithm proceeds as follows. At the start of the jth round ($j \geq 1$), the adder generates the values $S_{n(j-1)}, \ldots, S_{nj-1}$. Then all the processors operate in parallel and processor P_i encrypts $\mathsf{nonce}||S_{n(j-1)+i}$.
Problem: The single adder introduces delay which affects the overall performance of the parallel implementation.

n **adders:** In this case, each P_i has its own adder. Its local counter is initialized to S_i and after each block cipher invocation, the adder adds n to the local counter.
Problem: In this implementation, the cost of implementing n adders can take up chip area which is better utilized otherwise.

A.4.1 LFSR Based Solution

Note that the only restriction on the sequence $S_0, S_1, \ldots$ is that it is non-repeating. Thus, one can use a maximal length LFSR with a primitive connection polynomial to generate the sequence. Again there are two approaches to the design both of which are better than the corresponding approach based on using adders.

Single LFSR: In this case, a single LFSR is used which is initialized with a non-zero s-bit value. For $j \geq 1$, before the start of the jth round, the LFSR is clocked n times to produce the values $S_{n(j-1)}, \ldots, S_{nj-1}$. P_i then encrypts $\mathsf{nonce}||S_{n(j-1)+i}$ as before. Clocking the LFSR n times introduces a delay of only n clocks into the system.

n **LFSRs:** We can avoid the delay of n clocks by using n different implementations of the same LFSR initialized by suitable s-bit values to ensure that the sequences generated by the implementations are pairwise disjoint. The description of how this can be done is given in Section A.2. As discussed earlier, the cost of n separate implementations of the same LFSR scales linearly with the value of n.

Let us consider n AES blocks which are running in parallel. We could use either a single sequence generator to feed the n blocks or use n sequence generators to feed the n blocks. In the second approach, n registers will be required, irrespective of whether a sequence generator is implemented using a counter or an LFSR.

The design must specify the actual LFSR being used, and the required initial condition(s). Since there are many maximal length LFSRs to choose from, this provides additional flexibility to the designer.

A.4.2 Salsa20 Stream Cipher

Salsa20 is an additive stream cipher which has been proposed by Dan Bernstien as a candidate for the recent Ecrypt call for stream cipher primitives. The core design of Salsa20 consists of a hash function which is used in the counter mode to obtain a stream cipher. Denote by $\mathsf{Salsa20}_x()$ the Salsa20 hash function. Then the pseudo-random stream is defined as follows.

$$\mathsf{Salsa20}_x(v, S_0), \mathsf{Salsa20}_x(v, S_1), \mathsf{Salsa20}_x(v, S_2), \ldots$$

where v is a 64-bit nonce and $S_i = \mathsf{bin}_{64}(i)$. For hardware implementation, we can possibly generate the sequence $S_0, S_1, \ldots$ using an LFSR as described above. This defines a variant of the Salsa20 stream cipher algorithm. We believe that this modification does not diminish the security of Salsa20.

A.4.3 Discussion

For certain algorithms replacing counters by LFSRs will not provide substantial improvements. For example, hardware implementation of $\mathsf{Salsa20}_x()$ will require an adder since addition operation is required by the Salsa20 algorithm itself. Hence, avoiding the adder for generating the sequence $S_0, S_1, S_2, \ldots$ might not provide substantial improvements. On the other hand, let us consider AES. No adder is required for hardware implementation of AES. Hence, using LFSR(s) to produce the sequence $S_0, S_1, S_2, \ldots$ will ensure that no adder is required for hardware implementation of the counter mode of operation. In this case, the benefits of using LFSRs will be more pronounced.