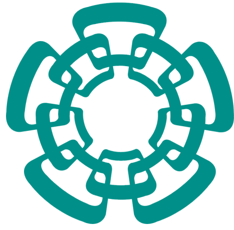


Centro de Investigación y de Estudios
Avanzados del Instituto Politécnico Nacional

Unidad Zacatenco
Departamento de Computación

Studies on Disk Encryption

A dissertation submitted by

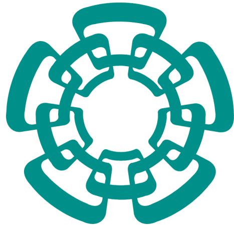
Cuauhtemoc Mancillas López

For the degree of

Doctor of Computer Science

Advisor:
Dr. Debrup Chakraborty

México, D.F. March, 2013



Centro de Investigación y de Estudios
Avanzados del Instituto Politécnico Nacional

Unidad Zacatenco
Departamento de Computación

Estudios sobre Cifradores de Disco

Tesis que presenta

Cuauhtemoc Mancillas López

para obtener el grado de

Doctor en Ciencias de la Computación

Director de tesis:
Dr. Debrup Chakraborty

México, D.F. March, 2013

Abstract

Security of data stored in bulk storage devices like hard disks, flash memories, CDs and DVDs has gained a lot of importance in the current days. The importance of this topic is reflected in recent standardizing activities and a variety of cryptographic schemes proposed in the last decade as a solution to this problem. In this thesis we address several issues related to the problem of encryption of stored data. Our main focus is on block oriented storage medias like hard disks and flash memories. In the following paragraphs we summarize the different problems that we address in this thesis along with our contributions.

There has been a consensus among researchers that a class of cryptographic algorithms called tweakable enciphering schemes (TES) can be used in the application of encrypting hard disks. In the last decade there have been many different proposals of TES each using different philosophies of construction. As a first contribution of this thesis we provide the first experimental performance data for (almost) all existing TES. The reported performance data is based on optimized implementations of the schemes on several families of reconfigurable hardware. While working towards efficient implementations of existing schemes we encountered some very interesting algorithmic and combinatorial problems. We present solutions to these problems also in this thesis, and they can be of a more broad interest.

We also propose some new schemes suitable for the problem. Among others, we propose a new TES called STES (Small TES) which is designed using a different philosophy compared to the other existing TES. The design goal of STES is to make it suitable for encrypting storage provided in devices which are constrained in terms of power consumption and area. STES uses cryptographic primitives which when implemented would have a very low hardware and power footprint in a novel way. We formally prove that STES provides adequate security for the application and also provide performance data in two classes of FPGAs which are suitable for low-power implementations. The performance of STES both in terms of throughput per area and power consumption is very encouraging.

In real life, all computations run in some physical device. When a physical device performs some computation it always emit or leak certain information. This leakage can be in the form of timing information, electromagnetic radiation, power consumption information or

even sound. In the case of cryptographic computations, these leakages if measured properly can be used to gain important information regarding secret quantities handled by the computational process. Analyzing cryptographic implementations in the light of these leakages is collectively called side channel analysis. We provide some preliminary side channel analysis on some TES. To our knowledge no such analysis has been done before on TES.

TES are length preserving schemes, in the sense that the length of the cipher text produced by a TES is same as that of the plain text. This property of length preservation has been considered very important for an encryption scheme to be suitable for encrypting hard disks. In this thesis we contest this well established notion, and argue why it may be possible to use encryption schemes which are not length preserving. We argue about this taking in consideration the structure of modern day hard disk. Finally we propose a new scheme called BRW-Counter mode (BCTR) which is not length preserving but provides the same security of that of a TES. We also present an optimal hardware architecture for BCTR and show that BCTR would outperform all other TES in terms of throughput.

Finally, we address the problem of securing backups by use of a new cryptographic scheme. We propose a cryptographic primitive which we call as the double cipher text mode (DCM) and discuss the general syntax and security definition of a DCM. We provide two efficient constructions of DCM which we name as DCMG and DCM-BRW. We argue why DCM would be suitable for the application of secure backup.

Resumen

En estos días, la seguridad de la información contenida en dispositivos de almacenamiento masivo como discos duros, memorias flash, CDs y DVDs ha ganado mucha importancia. Dicha importancia se refleja en las recientes actividades de estandarización y la gran variedad de esquemas criptográficos propuestos para dar solución a este problema. En esta tesis tratamos varios temas relacionados con este problema. Nuestro principal interés está en los medios de almacenamiento tales como discos duros y memorias flash que están organizados en sectores. En los próximos párrafos se resumen los diferentes problemas que fueron abordados a lo largo de esta tesis.

Hay un consenso entre los investigadores de que una clase de algoritmos criptográficos conocidos en inglés como *Teakable Enciphering Schemes (TES)*, pueden ser usados para cifrar discos duros. Como primera contribución de esta tesis presentamos el primer reporte con evidencia experimental acerca de la eficiencia para casi todos los TES existentes. Dicho reporte está basado en implementaciones optimizadas en diversas familias de dispositivos de hardware reconfigurable. Mientras desarrollábamos las implementaciones encontramos algunos problemas algorítmicos y combinatorios muy interesantes. Presentamos soluciones a dichos problemas, que pueden ser de un interés más amplio en otros contextros.

Así mismo proponemos algunos esquemas novedosos que resultan ser adecuados para la resolución de este problema. Entre otros, proponemos un nuevo TES llamado STES (por sus siglas en inglés provenientes de *Small TES*) el cual comparado con los TES existentes fue diseñado con una filosofía diferente. El objetivo de diseño de STES es de hacerlo apto para cifrar medios de almacenamiento disponibles en dispositivos restringidos en términos de área y consumo de potencia. STES está construido con primitivas criptográficas que al ser implementadas ocupan pocos recursos de hardware y consumen poca potencia. Además demostramos formalmente que STES provee la seguridad necesaria para la aplicación de cifrado de disco y también presentamos datos acerca del rendimiento usando dos familias diferentes de FPGAs que son apropiados para implementaciones orientadas al bajo consumo de potencia. El rendimiento de STES en términos de tasa de procesamiento de datos y consumo de potencia es muy alentador.

En la vida real todos los algoritmos se ejecutan en algún dispositivo físico. Cuando un dispositivo físico realiza algún cálculo, siempre se produce la fuga de cierta información. Dicha fuga puede ser en forma de información acerca del tiempo, radiación electromagnética, información del consumo de potencia e incluso ondas acústicas. En el caso de cálculos criptográficos cuando estas fugas de información son medidas apropiadamente pueden ser utilizadas para obtener información muy sensible acerca de los parámetros secretos manejados por el proceso de cómputo. El análisis de las implementaciones criptográficas analizando la informaciÃşn fugada es llamado *side channel analysis.* En esta tesis presentamos ataques de este tipo contra algunos TES existentes, de nuestro conocimiento, no se ha realizado antes este tipo de análisis contra los TES.

Los TES son esquemas que preservan la longitud, ya que el tamaño del texto cifrado producido por ellos es exactamente el mismo que el tamaño del texto plano. Esta propiedad ha sido considerada como muy importante para que un esquema criptográfico sea adecuado para cifrar discos duros. En esta tesis, impugnamos esta noción bien establecida, y argumentamos el porqué puede ser posible utilizar algoritmos de cifrado de disco que no preservan la longitud. Argumentamos esto tomando en consideración la estructura física de los discos duros modernos. Finalmente proponemos un nuevo esquema llamado *BRW-Counter mode* (BCTR) que no preserva la longitud pero provee la misma seguridad que un TES. También presentamos una arquitectura de hardware para BCTR y mostramos que mejora a los TES en términos de tasa de procesamiento de datos.

Finalmente, tratamos el problema de respaldo seguro de información utilizando un nuevo esquema criptográfico. Proponemos una primitiva criptográfica la cual llamamos doble texto cifrado (DCM por sus siglas en inglés) y discutimos su sintaxis y definición de seguridad. Damos dos construcciones eficientes de DCM las cuales llamamos DCMG y DCM-BRW. Argumentamos porqué DCM es conveniente para la aplicación de respaldo seguro de información.

Acknowledgments

This work was done under the supervision of Dr. Debrup Chakraborty. I want to acknowledge his dedication and patience towards this project and the valuable friendship that he extended to me during the course of this work. He showed me the way to do research.

I would like to thank Dr. Francisco Rodríguez Henríquez for his valuable collaboration in the works related to hardware implementations. We had had many enlightening discussions during the course of this work which helped me a lot to understand intricate issues of hardware design.

I want to give a special thanks to Prof. Palash Sarkar of Indian Statistical Institute, Kolkata, India, for receiving me as a visitor in the Cryptology Research Group (CRG). The discussions with him have been very interesting and productive. Also an important part of the work reported in this thesis have been done in collaboration with him. Also I would like to thank the students and professors for their hospitality during my research visit.

I want to thank Dr. Gerardo Vega Hernández, Dr. César Torres Huitzil and Dr. Guillermo Morales Luna who were part of the committee of my pre-doctoral examination. Critical comments and some subtle observations made by them during my pre-doctoral examination helped a lot.

I would like to thank Jérémie Detrey for the very valuable discussions about digital design and also for the good conversations about politics.

Dr. Luis Gerardo de la Fraga, Dr. Guillermo Morales Luna, Dr. Francisco Rodríguez Henríquez, Dr. Gerardo Vega Hernández and Dr. Raúl Monroy Borja were part of my thesis defence committee. I would like to thank them all for the time they spent in reading my thesis and their suggestions.

I thank my family Yaucalli, Yollolxochitl, Baudelio, Eugenia and Ahuitz for their unconditional love and support.

I will always remember the time that I lived among great friends and companions at the CINVESTAV: Saúl Zapotecas, William de la Cruz, Ivonne Avila, Alejandro García, Edgar Ventura, Luis Julian Domínguez, Sandra Díaz, Líl María, Arturo Yee, Eduardo Vázquez, Alfredo Arias, Anallely Olivares. And to all the professors, administrative staff and students at Computer Science Department of CINVESTAV who in one way or another helped me in my research.

A very special thanks to Elizabhet Cruz, that showed me that life is better and that one can be truly happy.

I acknowledge the support from CONACyT project 166763 and the CONACyT scholarship along these four years.

Contents

1 Introduction **17**

1.1 The Disk Encryption Problem 17

1.2 Scope of the thesis 19

2 Preliminaries **23**

2.1 General Notation 23

2.2 Fields 23

2.3 Block Ciphers 27

2.4 Pseudorandom Functions and Permutations 27

2.4.1 Pseudorandom Functions 29

2.4.2 Pseudorandom Permutation 30

2.5 Block Cipher Mode of Operation 32

2.5.1 Privacy Only Modes 32

2.5.2 Authenticated Encryption 33

2.6 Game Based Security Proofs 35

2.6.1 An Example 37

2.7 Summary 43

3 Reconfigurable Hardware **45**

3.1 Reconfigurable Computing: A Brief History 45

3.2 Field Programmable Gates Arrays 48

3.2.1 Logic Elements 48
3.2.2 Interconnection Resources 49
3.3 Xilinx Spartan 3 FPGAs 53
3.3.1 Configurable Logic Blocks 54
3.3.2 Interconnection Resources 55
3.4 Xilinx Virtex 5 FPGAs 58
3.4.1 Configurable Logic Blocks 59
3.5 Lattice ICE40 FPGAs 62
3.5.1 Programmable Logic Block 63
3.6 Basics of FPGA Programming 64
3.6.1 Hardware Description Languages 65
3.6.2 Design Flow 67
3.7 Summary 69

4 Tweakable Enciphering Schemes 71
4.1 Tweakable Block Ciphers 71
4.2 Tweakable Enciphering Schemes: Definitions and Security Notions 73
4.3 A Brief History of the Known Constructions of Tweakable Enciphering Schemes 75
4.4 Tweakable Enciphering Schemes and Disk Encryption 76
4.5 Description of some TES 77
4.5.1 Hash-Counter-Hash 77
4.5.2 Encrypt-Mask-Encrypt 79
4.5.3 Hash-Encrypt-Hash 81
4.6 Activities of IEEE SISWG 83
4.7 Final Remarks 84

5 Baseline Hardware Implementations of TES 85
5.1 Design Decisions 86
5.2 Implementation of Basic Blocks 87
5.2.1 The AES Design 88

5.2.2 The Design of the Multiplier 89
5.3 The Design Overviews 91
5.4 Implementation Aspects 97
5.5 Implementation of HCH 100
5.6 Results 104
5.6.1 Main Building Blocks 104
5.6.2 Performance Comparison of the Six TES Modes 105
5.7 Discussions 107

6 Efficient Implementations of BRW Polynomials 111

6.1 BRW Polynomials 113
6.2 A Tree Based Analysis 114
6.3 Scheduling of Multiplications 119
6.3.1 Some examples on algorithm Schedule 122
6.4 Optimal Scheduling 123
6.5 The Issue of Extra Storage 127
6.5.1 Determining the number of intermediate storage locations required by Schedule 129
6.6 A Hardware Architecture for the Efficient Evaluation of BRW Polynomials . 129
6.6.1 The Multiplier 130
6.6.2 Hardware Architecture to Evaluate BRW Polynomials 131
6.6.3 Scalability 135
6.7 Summary and Discussions 136

7 TES constructions based on BRW Polynomials 139

7.1 The Schemes 139
7.2 Analysis of the Schemes and Design Decisions 140
7.3 Analysis of the schemes 143
7.4 Architecture of HMCH[BRW] 144
7.4.1 The AES 144

7.4.2 Design of HMCH 146
7.4.3 HEH[Poly] and HMCH[Poly] Using Pipelined Multipliers 148
7.5 Experimental Results 149
7.5.1 Comparison with implementations in Chapter 5 151
7.6 Final Remarks 152

8 STES: A New TES Amenable to Low Area/Power Implementation 153

8.1 Some Technical Preliminaries 155
8.1.1 Stream Ciphers with IV 155
8.1.2 Multilinear Universal Hash 155
8.2 Construction of STES 157
8.2.1 Some Characteristics of the Construction 160
8.3 Security of STES 161
8.3.1 Proof of Theorem 8.1 161
8.3.2 Collision Analysis 164
8.4 Hardware Implementation of STES 168
8.4.1 Basic Design Decisions 168
8.4.2 Implementation of Universal Hash 170
8.4.3 Implementation of stream ciphers 173
8.4.4 Implementation of STES 174
8.4.5 Data Flow and Timing Analysis 176
8.5 Experimental Results 178
8.5.1 Primitives 179
8.5.2 Experimental results on STES 181
8.5.3 Comparison with Block Cipher Based Constructions 184
8.5.4 Discussions 187

9 Side Channel Attacks on Some TES 189

9.1 Adversaries with Access to Side Channel Information 190

9.2 Side Channel Weakness in the *xtimes* operation 191
9.3 The Attack on EME 193
9.3.1 The Distinguishing Attack 194
9.3.2 The Stronger Attack 194
9.4 EME2 Mode of Operation 198
9.5 A Distinguishing Attack on EME2 200
9.6 Final Remarks 201

10 A New Model for Disk Encryption 203
10.1 Deterministic Authenticated Encryption Schemes 204
10.1.1 Security of DAEs 205
10.2 In Support of Tagged Mode for disk Encryption 207
10.2.1 Which Encryption Scheme? 207
10.2.2 Gains and Loses in using DAE for disk encryption. 209
10.3 BCTR: A new DAE suitable for disk encryption 210
10.4 Security of BCTR 212
10.5 Hardware Implementation 214
10.5.1 Proposed Architecture 214
10.5.2 Timing Analysis 215
10.5.3 Results 217
10.6 Deferred Proofs 218
10.7 Final Remarks 224

11 A Proposal for Secure Backup 225
11.1 The Double Ciphertext Mode 227
11.1.1 Secure Backup Through DCM 227
11.1.2 Security of DCM 229
11.1.3 Discussions on the Adversarial Restrictions 230
11.2 DCMG: A generic construction of DCM 231
11.2.1 Characteristics of the construction 232

11.2.2 Security of DCMG . . . 234
11.3 Constructing a DCM Using BRW Polynomials . . . 236
11.3.1 The Construction . . . 236
11.3.2 Comparisons . . . 237
11.3.3 Security of DCM-BRW . . . 240
11.4 Proofs . . . 240
11.5 Remarks . . . 250

12 Conclusions and Future Work 251
12.1 Conclusions and Summary of Contributions . . . 251
12.2 Future Work . . . 254

Notation

$\perp$	Undefined value.
$\{0,1\}^*$	The set of all binary strings.
$\{0,1\}^n$	The set of n-bit binary strings.
$\|L\|$	If L is a string $\|L\|$ denotes its length, if A is a set then $\|A\|$ denotes its cardinality.
$A\|\|B$	Concatenation of the string A and B.
$a \xleftarrow{\$} A$	a is an element drawn uniformly at random from the set A.
$L << k$	Left-shift of L by k bits.
$L >> k$	Right-shift of L by k bits.
$L \lll k$	Left-circular rotation of L by k bits.
$\mathcal{A}^{\mathcal{O}_1} \Rightarrow 1$	An adversary $\mathcal{A}$, interacts with the oracle $\mathcal{O}_1$, and finally outputs the bit 1.
$\mathbf{Adv}_F^U(\mathcal{A})$	Advantage of the adversary $\mathcal{A}$ in breaking F in the sense U.
$\mathsf{bin}_n(\ell)$	n bit binary representation of an integer ℓ, where $0 \leq \ell \leq 2^n - 1$.
$\mathsf{bits}(\tau, i, j)$	A substring of τ between bits i and j.
$E_K()$	Block cipher with key K.
$\widetilde{E}_K^T()$	Tweakable block cipher, with key K and tweak T.
$E_K^T(M)$	Tweakable enciphering scheme.
$\mathbb{F}_q$	A finite field with q elements.
$\mathsf{Func}(m, n)$	The set of all functions mapping from m bits to n bits.
$GF(q)$	A finite field with q elements.
$\mathsf{lsb}(L)$	The least significant bit of L.
$\mathsf{msb}(L)$	The most significant bit of L.
$\mathsf{Perm}(n)$	The set of all permutations from $\{0,1\}^n$ to $\{0,1\}^n$.
$\mathsf{Perm}^{\mathcal{T}}(n)$	The set of all tweak indexed permutations from $\{0,1\}^n$ to $\{0,1\}^n$.
$\Pr[\zeta]$	The probability of the event ζ.
SC_K	Stream Cipher with key K.
xtimes(A)	Polynomial $A(x)$ multiplied by the monomial x modulo an irreducible polynomial.

Abbreviations

AE	Authenticated Encryption.
AEAD	Authenticated Encryption with Associated Data.
AES	Advanced Encryption Standard.
ASIC	Application Specific Integrated Circuit.
AXU	Almost Xor Universal.
AU	Almost Universal.
BRW	Bernstein-Rabin-Winograd polynomials.
CLB	Configurable Logic Block.
DAE	Deterministic Authenticated Encryption.
FPGA	Field Programmable Gate Array.
HDL	Hardware Description Language.
IV	Initialization Vector.
LUT	Lookup Table.
MAC	Message Authentication Code.
MLUH	Multilinear Universal Hash Function.
PD	Pseudo dot product.
PRF	Pseudorandom Funtion.
PRP	Pseudorandom Permutation.
SCA	Side Channel Analysis.
SPRP	Strong Pseudorandom Permutation.
TBC	Tweakable Block Cipher.
TES	Tweakable Enciphering Scheme.
VHDL	Very High Speed Integrated Circuits Hardware Description Language.
VLSI	Very Large Scale Integration.

Chapter 1

Introduction

The basic difference between an ordinary man and a warrior is that a warrior takes everything as a challenge, while an ordinary man takes everything as a blessing or as a curse.

Don Juan Matus

These days we manage a lot of information and some of it is kept in storage devices such as hard disks, flash memories, DVDs, CDs, etc. Independently of whether the data is for personal use or corporate use, always there is sensitive information which we need to protect against possible unauthorized accesses and modifications. In several ways an unwanted person can gain access to our sensitive information. For example, it has been estimated that the laptop loss rates are around 2% per year [54], which signifies that an organization with 100,000 laptops, may lose on average several of them per day. A stolen laptop amounts to the loss of the hardware and the data stored in it. But, what is of more severe consequence is that sensitive information gets into the hands of an unwanted person who can potentially cause much greater damage than the one incurred by the mere physical loss of the hardware and the data. A possible countermeasure of this important problem is to encrypt the data being written in the hard disk or other kinds of storage media. Securing stored data has received a lot of attention in the current days. In this thesis we study various aspects of this problem, we would be particularly interested in securing information stored in hard disks and flash memories. In this Chapter we provide an informal introduction to the problem of disk encryption and later in Section 1.2 we provide a brief summary of the rest of the thesis.

1.1 The Disk Encryption Problem

To protect unauthorized access to stored information in hard disks one can apply encryption to the stored data. Although there exist numerous encryption schemes meant for varied

scenarios, this special application brings with it specific design problems which cannot be readily solved by traditional encryption schemes.

It has been argued that the best solution to this issue would be a hardware based scheme, where the encryption algorithm resides in the disk controller, which has access to the disk sectors but has no knowledge about the high-level logical partitions of the disk, such as files and directories, which are maintained by the operating system. Under this scenario, the disk controller encrypts the data before it writes a sector, and similarly after reading a sector, the disk controller decrypts it before sending it to the operating system. This type of encryption has been termed in the literature as *low level disk encryption* or *in-place disk encryption*, see Figure 1.1.

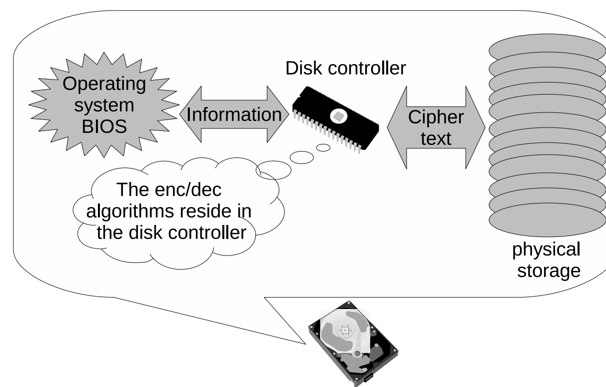


Figure 1.1: Disk Encryption Problem

A symmetric key cryptosystem with certain specific properties can serve as a solution to the low level disk encryption problem. One particularly important property to achieve is length preserving encryption, *i.e.*, the length of the ciphertext should not be more than that of the plaintext. This implies that the ciphertext itself must be enough to decrypt the enclosed data, since there is no scope to store associated data like states, nonces, salts or initialization vectors, which are common parameters in numerous symmetric key cryptosystems. Furthermore, the schemes to be selected must be secure against adaptive chosen plaintext and adaptive chosen ciphertext adversaries. Such schemes are generally called CCA secure schemes (secure against chosen cipher text attacks). Achieving CCA security means that no adversary can be able to distinguish the ciphertexts from random strings, and additionally the attacker must not be able to modify the ciphertext so that it gets decrypted to something meaningful.

The properties which we described above are all provided by a class of encryption schemes called *Tweakable Enciphering Schemes* (TES). TES are length preserving, their security model is a strong pseudorandom permutation indexed by a tweak. The tweak is an extra public parameter that increases the variability of the cipertext, i.e., if two different plaintext are encrypted using the same key but the tweaks are different the resulting ciphertexts will

be different. Other important property of TES is that if any part of the ciphertext is changed the plaintext obtained after decryption looks like random strings.

There have been a vigourous activity in designing tweakable enciphering schemes in the last decade. To date there are more than ten different proposals for such schemes, additionally there is an active standardization effort by the IEEE working group on security in storage (SISWG) [2] which has been working for the past few years to formulating standards for encryption algorithms suitable for various storage medias.

This thesis is largely devoted to the study of tweakable enciphering schemes. We study various aspects of TES including their implementations, security in various situations and their usability in various environments. We summarize the main contributions and the contents of the rest of the thesis in the next section.

1.2 Scope of the thesis

This thesis aims to study various aspects of the problem of disk encryption. The thesis is divided into twelve chapters and in each chapter we address some problem related to disk encryption. Next we provide a brief summary of each chapter and discuss the important contributions that are reported in each chapter.

Chapters 2 to 4 do not contain any new material they are supposed to provide the background information necessary to appreciate the contributions that we present in the later chapters.

In **Chapter 2** we describe some mathematical preliminaries. We begin with a brief exposition of fields with a focus on binary extension fields. Field operations are extensively used in rest of the thesis. We then briefly discuss the syntax and use of a block-cipher which is probably the most used symmetric key cryptographic primitive. We define the mathematical concepts of a pseudorandom function and permutation and describe how these objects are used to define security of block ciphers. Finally, we give quite a detailed account of the structure of reductionist security proofs which are used throughout the thesis to prove security of proposed schemes. We also give a detailed example of a security proof to explain the various techniques involved in it.

A big part of the thesis deals with efficient implementations of disk encryption schemes in reconfigurable hardware of various families. We give a brief introduction to reconfigurable computing in **Chapter 3**. In Chapter 3 we start with a brief history of reconfigurable computing and then in the subsequent sections we describe the basic resources of a field programable gate array (FPGA). We also give a detailed description of three FPGA families namely Xilinx Spartan 3, Xilinx Virtex 5 and Lattice ICE40. These families of FPGAs have been extensively used for prototypical implementations in this study. Finally, we describe some issues involved in FPGA programming.

In **Chapter 4** we discuss tweakable enciphering schemes. As stated earlier till now TESs are considered to be the most suitable option for the application of disk encryption. In Chapter 4 we discuss the syntax and security definitions of TES and also give algorithmic descriptions of some existing TES. This Chapter also contains discussions regarding the suitability of TES for disk encryption and a detailed account of the activities IEEE Security in Storage Working Group. For the past few years this working group has been working for formulating standards for various aspects of storage encryption. The original contributions of this thesis begins from Chapter 5.

We mentioned earlier that currently there are around ten different proposals for TES, but before we started working on this problem there were no implementations reported in the open literature. Thus, we were the first to report optimized hardware implementations of (almost) all existing TES in different hardware platforms. In **Chapter 5** we report the baseline hardware performance of six TES. The designs presented in Chapter 5 are optimized for Virtex 4 FPGAs but we also present place and route performance data for other FPGA families. The designs are for 512-byte sector size , but they are scalable and the throughput rates would be sustained if the designs are upgraded to support sector sizes of 4096 bytes. The performance data of the TES in Chapter 5 indicates that they can be used in disk controllers which have a data rate close to 3 Gbits/sec. The contents of this Chapter is based on the paper [97].

Some of the schemes presented in Chapter 5 are constructed using polynomial hashes. In a paper published in 2009 [125] it was noted that if a normal polynomial hash is replaced by a special polynomial called Bernstein Rabin Winograd (BRW) polynomial then one can design more efficient TES. In **Chapter 6** we study the problem of designing an efficient circuit for computing BRW polynomials. This goal led us to do an in-depth study of the structure of BRW polynomials to find opportunities for exploiting parallelism. This study led us to some very interesting combinatorial results involving the structure of BRW polynomials. In Chapter 5 we provide a complete characterization of parallelism which is achievable for computing BRW polynomials. In addition we present an algorithm which decides the order in which different multiplications are to be performed given a certain level of allowed parallelism. We show that the order of multiplication given by our algorithm is the optimal one (for cases where the optimal is achievable). Furthermore, using these combinatorial properties of BRW polynomials we design an efficient circuit to compute them.

Chapter 7 is devoted to the construction of hardware for the TESs reported in [125] which uses BRW polynomials. The results obtained in the implementations in this Chapter are far better than the ones reported in Chapter 5. For some designs we obtain throughput greater than 10 Gbits per second, and this throughput is far above the data rate of any commercially available disk controller till date. The contents of Chapters 6 and 7 are based on the paper [27].

In **Chapter 8** we focus on a different application of storage encryption. Now-a-days there

are variety of small and mobile devices which provide a non-negligible amount of storage, for example, mobile phones, tablets, cameras etc. The security of stored information in these devices is more important than those in desktops/laptops, as the possibility that an user loses this small device is far more than the loss of a laptop. These devices are all battery powered and are thus constrained in terms of power utilization and also size. Moreover the storage in these devices is provided through non-volatile flash memories. A class of flash memories called NAND type memories have a similar block wise organization as in hard disks. Thus the schemes which can be used to encrypt hard disks can also be applied here, but two important considerations are the constrained environment in which these devices work and the data rates in flash memories are far less than what is provided by modern disk controllers. These considerations dictates that the constructions and architectures developed in Chapters 5 and 7 are not suitable for these devices. Hence, in Chapter 8 we propose a new TES which is very different from the existing ones, we call this new construction STES. STES uses stream ciphers, the motivation of developing a TES using stream cipher is the fact that there are many recent proposals of stream ciphers which can be implemented with a very low hardware and power footprint and yet provide reasonable performance in terms of speed. We discuss the details of STES and its philosophy of construction and also formally prove that STES is a secure TES. Finally, we provide implementations of STES with various instantiations of stream ciphers. Also we implement STES using various data-paths to achieve a wide range of throughput. All these implementations provide a very good time/area tradeoff and the range of speed, area and power consumption characteristics of these implementations would be able to provide encryption functionality in a wide range of commercially available non-volatile memory devices.

In **Chapter 9** we give a brief discussion of side channel attacks against cryptographic implementations and explore side channel vulnerabilities in two TES schemes EME and EME-2. Our analysis suggests that both EME and EME-2 are insecure if an adversary has access to some side channel information. The study in this chapter is purely theoretical in nature, we are yet to obtain experimental results which would show the extent of security weakness that these modes have in a real implementation. The contents of this chapter have been previously published in [96].

In **Chapter 10** we investigate an established belief that only length preserving encryption schemes can be applied for encrypting block oriented storage devices like hard disks. The main argument which nurtures this belief is that hard disk sectors are of fixed length and thus cannot accommodate expansions in the ciphertext. Though it is true that hard disk sectors provide a fixed amount of space for user data, sectors are physically larger than the amount of user data it can store, and it stores many more information other than the user data for its proper functioning. This extra space in a sector is known as the format overhead of a disk. Though the amount of format overhead in various commercially available hard disks is a trade secret, it is estimated that in the modern disk the format overhead is around 15% of the user data. Our analysis reveals that the requirement of the length preserving

property may not be that important for disk encryption as disk sectors may be suitably formatted to accommodate ciphertext expansion. In this Chapter we also propose that an existing model of encryption called deterministic authenticated encryption (DAE) can be suitably used for disk encryption. We argue that DAE would be a much efficient alternative than TES, and DAEs can be suitably used to provide all security and usability characteristics required for disk encryption. Finally we propose a new DAE scheme called BCTR which is tailored to the application of disk encryption. We prove that BCTR is a secure DAE and build a hardware circuit for BCTR which is far more efficient in terms of speed compared to other implementations reported in this thesis.

In **Chapter 11** we study a different problem related to storage encryption. We explore the possibility of how one can maintain secure backups of the data, such that loss of a physical device will mean neither loss of the data nor the fact that the data gets revealed to the adversary. We propose an efficient solution to this problem through a new cryptographic scheme which we call as the double ciphertext mode (DCM). In this Chapter we describe the syntax of DCM, define security for it and give some efficient constructions. Moreover we argue regarding the suitability of DCM for the secure backup application. The contents of this Chapter is based on the paper [26].

In **Chapter 12** we conclude the thesis and discuss some feasible future directions of research related to the topic presented in this thesis.

Chapter

Preliminaries

The revolution is not an apple that falls when it is ripe. You have to make it fall.

Ernesto Guevara el Che

In this Chapter we introduce some important concepts necessary to follow this thesis. First of all we give some general notation and then describe the concepts of finite fields, block ciphers, pseudorandom functions and permutations. These mathematical objects are central to the thesis. We also give brief descriptions about block cipher modes of operation and an overview of the technique of game playing used to construct reductionist security proofs.

2.1 General Notation

We denote the set of all binary strings by $\{0,1\}^*$ and the set of all n bit strings by $\{0,1\}^n$. If $A, B \in \{0,1\}^*$, then by $A||B$ we mean concatenation of the strings A and B. By $L << k$ we shall mean left-shift of L by k bits; and $L >> k$ will mean the right-shift of L by k bits. $\mathsf{take}_k(L)$ will mean the k most significant bits of L. $\mathsf{msb}(L)$ and $\mathsf{lsb}(L)$ will mean the most significant and the least significant bits of L respectively. $\mathsf{bits}(L, i, j)$ will mean a substring of L from bit i to bit j. If b is a bit, then $\bar{b}$ will mean the complement of b. For a positive integer $\ell < 2^n$, by $\mathsf{bin}_n(\ell)$ we shall mean the n bit binary representation of ℓ. If $L \in \{0,1\}^*$, $|L|$ will denote the length of L and when A is a set, $|A|$ will denote the cardinality of A. $a \stackrel{\$}{\leftarrow} A$ will mean that a is an element drawn uniformly at random from the set A. For easy reference a list of notation is provided in the beginning of the thesis.

2.2 Fields

We sometimes see bit strings as elements of a field and apply field operations on bit strings. Here we discuss some relevant and elementary properties of fields which we shall require.

The discussion here is far from comprehensive, we refer the reader to relevant texts for a comprehensive treatment [68, 91].

Field: A field $\mathbb{F} = (S, +, \cdot)$ is a set S, together with two binary operations $+$ and $\cdot$ on S satisfying the following axioms,

For any elements $a, b, c \in S$:

(i)	$(a+b)+c = a+(b+c)$.	(Associativity of addition)
(ii)	$a+b = b+a$.	(Commutativity of addition)
(iii)	there exists $0 \in S$ such that $a+0 = a$.	(Existence of additive identity)
(iv)	for every $a \in S$ there exists $(-a) \in S$ such that $a+(-a) = 0$.	(Existence of additive inverse)
(v)	$(a \cdot b) \cdot c = a \cdot (b \cdot c)$.	(Associativity of multiplication)
(vi)	$a \cdot (b+c) = a \cdot b + a \cdot c$ and $(b+c) \cdot a = b \cdot a + c \cdot a$.	(Distributivity)
(vii)	there exists $1 \in S$ such that $1 \cdot a = a \cdot 1 = a$.	(Existence of multiplicative identity)
$(viii)$	$a \cdot b = b \cdot a$.	(Commutativity of multiplication)
(ix)	for every nonzero $a \in S$, there exists $a^{-1} \in S$ such that $a \cdot (a^{-1}) = 1$.	(Existence of multiplicative inverse)

Some examples of fields are the sets of rational, real and complex numbers with the usual operations of addition and multiplications defined for those sets. If $\mathbb{F}$ is a field we will also use $\mathbb{F}$ to denote the set of elements in the field, i.e., for us $\mathbb{F} = (\mathbb{F}, +, \cdot)$.

A vector space can be defined over any field by the same properties that are used to define a vector space over reals. Any vector space has a *basis* and the number of elements in its basis is called its *dimension*. Let $\mathbb{F}$, $\mathbb{L}$ be fields defined on the same binary operators and let $\mathbb{F} \subseteq \mathbb{L}$, then $\mathbb{F}$ is called the sub-field of $\mathbb{L}$ and $\mathbb{L}$ is said to be an extension of $\mathbb{F}$. Thus an *extension field* $\mathbb{L}$ of $\mathbb{F}$ is a bigger field containing $\mathbb{F}$, and is automatically a vector space over $\mathbb{F}$. We call $\mathbb{L}$ to be a *finite extension* of $\mathbb{F}$ if $\mathbb{L}$ is a finite dimensional vector space. By degree of a finite extension we mean the dimension of $\mathbb{L}$ over $\mathbb{F}$. A common way to obtain extension fields is to *adjoin* an element to $\mathbb{F}$, i.e., we say we obtain $\mathbb{F}(\alpha)$ by adjoining α to $\mathbb{F}$ where $\mathbb{F}(\alpha)$ consists of all rational expressions that can be formed using α and elements of $\mathbb{F}$. It is easy to see that $\mathbb{F}(\alpha)$ forms an extension of $\mathbb{F}$.

We define a polynomial over a field $\mathbb{F}$ as a formal expression

$$q(x) = a_n x^n + ... + a_2 x^2 + a_1 x + a_0, \tag{2.1}$$

where $a_n, ..., a_2, a_1, a_0 \in \mathbb{F}$. One can add and multiply polynomials over a field in the same

way as polynomials over reals, the operations over the coefficients are the operations in $\mathbb{F}$. The set of all polynomials over a field $\mathbb{F}$ is denoted by $\mathbb{F}[X]$. $\mathbb{F}[X]$ is a commutative ring with identity (i.e., it follows all the field axioms except the existence of multiplicative inverse), and is called the *polynomial ring* of $\mathbb{F}$. The *degree* of a polynomial is the largest power of x which occurs in the polynomial with a non zero coefficient. For example, if $a_n \neq 0$ then the degree of $q(x)$ (in Eq. 2.1) is n. A degree d polynomial is called *monic* if the coefficient of x^d is 1. So, if $a_n = 1$ then $q(x)$ is a monic polynomial of degree n.

For $f, g \in \mathbb{F}[X]$, we say that f divides g if there exist $h \in \mathbb{F}[X]$ such that $g = fh$. $f \in \mathbb{F}[X]$ is said to be *irreducible* if f is not divisible by any other polynomial in $\mathbb{F}[X]$ of lower degree except constants. The irreducible polynomials plays the same role in polynomials as primes play in the set of integers. The ring $\mathbb{F}[X]$ has the *unique factorization* property, i.e., a monic polynomial in $\mathbb{F}[X]$ can be written in one and only one way (except for the order of the factors) as a product of monic irreducible polynomials.

An element α in some extension $\mathbb{L}$ of $\mathbb{F}$ is called *algebraic* over $\mathbb{F}$ if it satisfies some polynomial in $\mathbb{F}[X]$. If α is algebraic over $\mathbb{F}$ then there exists an unique monic irreducible polynomial in $\mathbb{F}[X]$ with α as the root (and any other polynomial which is satisfied by α must be divisible by this monic irreducible polynomial). If this monic irreducible polynomial has degree d then any element in $\mathbb{F}(\alpha)$ can be expressed as linear combinations of the powers of α, i.e, $1, \alpha, \alpha^2, \ldots, \alpha^{d-1}$. Thus $\{1, \alpha, \alpha^2, \ldots, \alpha^{d-1}\}$ forms a basis of the vector space $\mathbb{F}(\alpha)$ over $\mathbb{F}$, thus the degree of extension of $\mathbb{F}(\alpha)$ is same as the degree of the monic irreducible polynomial.

If by adding the multiplicative identity 1 to itself in $\mathbb{F}$ never gives zero, then we say that $\mathbb{F}$ has *characteristric* zero. If otherwise, i.e., if $\mathbb{F}$ has a non-zero characteristic, then there always exist a prime number p such that $1 + 1 + \ldots + 1$ (p times) equals zero, and p is called the characteristic of $\mathbb{F}$. If $\mathbb{F}$ is a field of characteristic p then $\mathbb{F}$ always contains a copy of the field $\mathbb{Z}_p$ (the integers modulo p) in it.

Finite fields: A finite field is a field with finite number of elements. Let $\mathbb{F}_q$ denote a field with q elements. Clearly a finite field cannot have a zero characteristic, so let p (a prime) be the characteristic of $\mathbb{F}_q$. Thus, $\mathbb{F}_q$ contains the *prime field* $\mathbb{F}_p = \mathbb{Z}_p$ and thus, $\mathbb{F}_q$ is a finite dimensional vector space over $\mathbb{Z}_p$. Let f be the dimension of $\mathbb{F}_q$, thus every element of $\mathbb{F}_q$ can be represented uniquely by a tuple of f elements in $\mathbb{Z}_p$ and also each f tuple of elements in $\mathbb{Z}_p$ represents an unique element in $\mathbb{F}_q$. Thus the number of elements in $\mathbb{F}_q$ is $q = p^f$. Which shows that the number of elements in a finite field is always a power of a prime. Additionally one can show that for every prime p and every positive integer f there exists a unique field (up to isomorphisms) with p^f elements. A finite field is often called a Galois field and a field with p^f elements is denoted as $GF(p^f)$.

It is clear from the field axioms that the set of non zero elements in $\mathbb{F}_q$ forms a group under multiplication. This group is denoted $\mathbb{F}_q^*$. This group $\mathbb{F}_q^*$ is cyclic and thus have a generator, such a generator is sometimes called as a *primitive element* of the field.

Given a prime field $\mathbb{F}_p = \mathbb{Z}_p$, one can easily construct an extension field $\mathbb{F}_{p^n}$, as follows. Let $g(x)$ be a primitive irreducible polynomial of degree n in $\mathbb{F}_p[X]$.[1] Let α be a root of g (surely $\alpha \notin \mathbb{F}_p$ as g is irreducible). As discussed, $\mathbb{F}_p(\alpha)$ is an n degree extension of $\mathbb{F}_p$, i.e., it is a vector space of dimension n over $\mathbb{F}_p$, and has p^n elements, thus $\mathbb{F}_p(\alpha) = \mathbb{F}_{p^n}$.

The elements of $\mathbb{F}_{p^n} = \mathbb{F}_p(\alpha)$ can be represented as linear combination of $\{1, \alpha, \alpha^2, \alpha^3, ..., \alpha^{n-1}\}$ thus $\mathbb{F}_{p^n}$ can be considered to be the set of all polynomials of degree less than n over $\mathbb{F}_p$. Note that we can represent the elements in this field even without knowing the specific value of α, i.e., any polynomial $f(x)$ with coefficients in $\mathbb{F}_p$ and degree less than n represents an element in $\mathbb{F}_{p^n}$. The addition in the field is defined as ordinary polynomial addition. Multiplication is also defined as ordinary polynomial multiplication, but it is to be noted that $g(x) = 0$. Thus, if $a(x), b(x) \in \mathbb{F}_{p^n}$, then we define $a(x) \cdot b(x) = a(x) \cdot b(x) \bmod g(x)$.

Binary Extension Fields and String Representations: The field $\mathbb{F}_2 = GF(2)$ has $\{0, 1\}$ as its elements hence it receives the name of *binary field*, the addition operation in this field is simply an bit-xor operation and multiplication is a bit-and operation. $\mathbb{F}_{2^n} = GF(2^n)$ is an extension field of the field $GF(2)$ formed using an irreducible polynomial $q(x)$ of degree n in $\mathbb{F}_2[X]$, this field is called a *binary extension field.* The elements in $\mathbb{F}_{2^n}$ can be viewed as polynomials (with binary coefficients) of degree at most $n-1$. In turn such polynomials can be viewed as n bit strings. For example the set of all 8 bit strings can be viewed as the field $\mathbb{F}_8$. Like, the polynomial $x^7 + x^5 + x^3 + x^2 + 1$ corresponds to the binary string 10101101 or in more general way as an hexadecimal number 155.

Quite often we shall consider the set $\{0, 1\}^n$ as the binary extension field $\mathbb{F}_{2^n}$. Thus, if $A, B \in \{0, 1\}^n$, we can consider A, B as bit strings of length n or as polynomials $A(x), B(x)$ of degree less than n and with binary coefficients. The addition of elements $A, B \in \{0, 1\}^n$ is defined as $A \oplus B$, where the operation $\oplus$ denotes the bit wise xor of the strings. For defining multiplication of strings A and B we consider them as polynomials $A(x), B(x)$ and define $AB = A(x) \cdot B(x) \bmod q(x)$ where $q(x)$ is an irreducible polynomial of degree n.

If g is a n degree polynomial in $\mathbb{F}_2[X]$ such that its root $\alpha \in \mathbb{F}_{2^n}$ is a primitive element of $\mathbb{F}_{2^n}$, then $\alpha, \alpha^2, \ldots, \alpha^{n-1}$ are all distinct and α generates $\mathbb{F}^*_{p^n}$. So, given a non zero element $A \in \mathbb{F}_{2^n}$, all $A\alpha, A\alpha^2, \ldots, A\alpha^{n-1}$ would also be distinct. This property would find many applications in the schemes that we discuss. Thus if we consider $A \in \{0, 1\}^n$, we should be interested in the quantity αA which is computed as $xA(x)$, i.e., the product of the polynomial $A(x)$ with the monomial x modulo the irreducible polynomial $q(x)$ representing the field $\mathbb{F}_{2^n}$. This operation would be called *xtimes* and can be very efficiently realized by a bit shift and a conditional xor.

[1]such a polynomial always exist for every n and p, and there exist efficient algorithms to find them.

2.3 Block Ciphers

Consider a message space $\mathcal{M}$, a cipher space $\mathcal{C}$ and a key space $\mathcal{K}$, then a block cipher can be viewed as a function $E : \mathcal{M} \times \mathcal{K} \to \mathcal{M}$, where $\mathcal{M} = \mathcal{C} = \{0,1\}^n$ and $\mathcal{K} = \{0,1\}^k$. So a block cipher takes an n-bit input and produce an n-bit output under the action of a k-bit key, the values of n and k varies for different block ciphers. For any $K \in \mathcal{K}$ and $P \in \mathcal{M}$ we will denote a block cipher by $E_K(P)$ instead of $E(K,P)$. It is a requirement that for any $K \in \mathcal{K}$, $E_K(\cdot)$ must be a permutation, i.e., the function $E_K : \{0,1\}^n \to \{0,1\}^n$ must be a bijection. If E_K is a bijection then for every cipher text $C \in \{0,1\}^n$, there exists only one message $P \in \{0,1\}^n$ such that $C = E_K(P)$. $E_K(\cdot)$ has an inverse function denoted as $E_K^{-1}(\cdot)$ or $D_K(\cdot)$ such that $P = D_K(E_K(P))$.

In practice $E_K(\cdot)$ and $D_K(\cdot)$ are publicly known functions which are easy to compute, the key is secret which is in general drawn uniformly at random from the key space $\mathcal{K}$. Modern block ciphers in general are constructed using several identical transforms called rounds and iterating them. The structure of existing block ciphers can be classified into two types: Substitution Permutation Networks (SPNetwork) and Feistel Networks [81]. The first block cipher standardized was Data Encryption Standard (DES) [108]. DES is based on a Feistel Network. Currently the standard is Rijndael which is widely known as the Advanced Encryption Standard (AES) which is an SPNetwork [38].

2.4 Pseudorandom Functions and Permutations

In Section 2.3 we gave a syntactic definition of block-ciphers by specifying the domain and range of such functions. But any function with these syntactic properties cannot provide security that is expected of a block cipher. In this section we would try to give a security definition of a block-cipher. Such a definition is given by the help of pseudorandom objects called pseudorandom functions/ permutations. These objects form the fundamental building blocks of symmetric key cryptography, here we give an overview of these important cryptographic objects.

Consider the map $F : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ where $\mathcal{K}, \mathcal{D}, \mathcal{R}$ (commonly called keys, domain and range respectively) are all non-empty and $\mathcal{K}$ and $\mathcal{R}$ are finite. We view this map as representing a *family of functions* $F = \{F_K\}_{K \in \mathcal{K}}$, i.e., for each $K \in \mathcal{K}$, F_K is a function from $\mathcal{D}$ to $\mathcal{R}$ defined as $F_K(X) = F(K,X)$. For every $K \in \mathcal{K}$, we call F_K to be a instance of the family F.

Given a function family F where the sets keys, domain and range are not specified we shall often write $\mathsf{Keys}(F)$, $\mathsf{Dom}(F)$, $\mathsf{Range}(F)$ to specify them.

If $F : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ is a function family where $\mathcal{D} = \mathcal{R}$, and for every $K \in \mathcal{K}$, $F_K : \mathcal{D} \to \mathcal{D}$ is a bijection, then we say that F is a permutation family. So, if F is a permutation

family, then for every $F_K(\cdot)$, we have a $F_K^{-1}(\cdot)$, such that for all $K \in \mathcal{K}$ and all $X \in \mathcal{D}$, $F_K^{-1}(F_K(X) = X$. Note that, as per the definition of a block cipher in Section 2.3, a block-cipher is a permutation family.

We would be interested in probability distributions over a function family F, in particular we would often talk of sampling an instance at random from the family. By sampling an instance f uniformly at random from F we would mean $K \xleftarrow{\$} \mathcal{K}$ and $f = F_K()$, we will denote this by $f \xleftarrow{\$} F$.

Random Function. Let $\mathsf{Func}(\mathcal{D}, \mathcal{R})$ be the set of all functions mapping $\mathcal{D}$ to $\mathcal{R}$, if $\mathcal{D} = \{0,1\}^m$ and $\mathcal{R} = \{0,1\}^n$ then $\mathsf{Func}(m,n)$ is the set of all functions that map from m bits to n bits. Note, there are exactly 2^{n2^m} of these functions. i.e., $|\mathsf{Func}(m,n)| = 2^{n2^m}$. If $\mathcal{D}$ and $\mathcal{R}$ are specified, then by a random function with domain $\mathcal{D}$ and range $\mathcal{R}$ we mean a function sampled uniformly at random from $\mathsf{Func}(\mathcal{D}, \mathcal{R})$. Hence by a random function, we are not talking of the "randomness" of a specific function but we are talking of a function sampled from a probability distribution (specifically, the uniform distribution) over the set of all possible functions with a specified domain and range.

To work with random functions a more intuitive way is necessary, we will consider the procedure **RndF** described in Figure 2.1 which acts as a random function F. The procedure **RndF** maintains a table T indexed on the domain elements across invocations. It is assumed that initially $T[x]$ is undefined for every $x \in \mathsf{Dom}(F)$. Whenever **RndF**(x) is invoked, it checks if $T[x]$ contains a value, if $T[x]$ is not undefined then it returns $T[x]$, otherwise it returns an element sampled uniformly at random from $\mathsf{Range}(F)$ and stores the value returned in $T[x]$.

RndF(x)
1. **if** $T[x]$ is not defined **then**
2. $\quad Y \xleftarrow{\$} \mathsf{Range}(F)$;
3. $\quad T[x] \leftarrow Y$;
4. **end if**;
5. **return** T[x];

Figure 2.1: Simulation of a random function.

The behavior of the program **RndF** is as a random function because each assignation $Y \xleftarrow{\$} \mathsf{Range}(F)$ is independent to the all others, i.e., it constructs one member of the family F in a random way, and if it is invoked multiple times on the same domain element then it returns the same value. Note that the procedure **RndF** is not meant to be a practical realization of a random function, as even for functions from 128 bits to 128 bits it is infeasible to maintain the table T also the procedure does not mention the source of its randomness. The procedure

just helps us to see a random function in a procedural way.

Random Permutations. Let $\mathsf{Perm}(\mathcal{D})$ be the set of all bijective maps from $\mathcal{D}$ to $\mathcal{D}$. If $\mathcal{D} = \{0,1\}^n$, then we denote by $\mathsf{Perm}(n)$ the set of all permutations from $\{0,1\}^n$ to $\{0,1\}^n$. Similar to a random function, we define a random permutation with domain $\mathcal{D}$ to be a function chosen uniformly at random from $\mathsf{Perm}(\mathcal{D})$. To get a more intuitive feeling of a random permutation we describe a piece of code which implements a random permutation with domain $\mathcal{D}$ in Figure 2.2. The program $\mathbf{RndP}(x)$ shown in Figure 2.2 maintains a table T as in the procedure **RndF**, additionally it maintains a set S, through the information stored in the set S the procedure keep tracks of the values that it has already returned and thus ensures that the map it implements is a one-to-one map. The table T is indexed with the elements of $\mathcal{D}$ and initially $T[x]$ is undefined for every $x \in \mathcal{D}$. The set S is initially empty. When **RndP** is invoked with an element $x \in \mathcal{D}$, it first checks if $T[x]$ is undefined, if it is so then **RndP** has never been invoked on x, so it selects uniformly at random an element from $\mathcal{D} - S$ (the set of elements which have not yet been returned by **RndP**) returns it and stores it in $T[x]$ and also adds the returned value in the set S. If $T[x]$ is not undefined then the value $T[x]$ is returned. This way **RndP** keep the correctness of the permutation, i.e., all the values stored in T are different.

RndP(x)
1. **if** $T[x]$ is not defined **then**
2. $\quad Y \xleftarrow{\$} D - S$;
3. $\quad T[x] \leftarrow Y$;
4. $\quad S \leftarrow S \cup \{T[X]\}$;
5. **end if**;
6. **return** T[x];

Figure 2.2: Simulation of a random permutation.

2.4.1 Pseudorandom Functions

Informally a pseudorandom function (PRF) is a family of functions whose behavior is computationally indistinguishable from a random function. Consider the function family $F : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$, and let $f \xleftarrow{\$} F$ and let $\rho \xleftarrow{\$} \mathsf{Func}(\mathcal{D}, \mathcal{R})$. If F is a PRF family then there should be no efficient procedure to distinguish between f and ρ. To formalize this goal of distinguishing between a random instance of F and a random instance of $\mathsf{Func}(\mathcal{D}, \mathcal{R})$, we introduce an entity which we call as a PRF adversary. A PRF adversary is considered to be a probabilistic algorithm whose goal is to distinguish between f and ρ, and if it can successfully do so then we say that the adversary has broken the PRF property of F. The

adversary is not provided with the description of the functions but it has an *oracle access* to a function g which is either f or ρ and it needs to decide whether $g = f$. By an oracle access we mean that for any $x \in \mathcal{D}$ of its choice, the adversary can obtain the value $g(x)$ by querying the oracle of g. The adversary has the ability to query its oracle g adaptively, i.e., it may be that first it wishes to query its oracle on x_1 and thus obtain $g(x_1)$, seeing $g(x_1)$ it decides its next query x_2 and so on. The adversary can query its oracle as long as it wants and finally it outputs a bit, say it outputs a 1 if it thinks that its oracle is f (a real instance from the family F) and a zero if it thinks its oracle is ρ (a random function). An adversary $\mathcal{A}$ interacting with an oracle $\mathcal{O}$ and outputting a 1 will be denoted by $A^{\mathcal{O}} \Rightarrow 1$.

The PRF advantage of an adversary $\mathcal{A}$ in distinguishing F from a random function is defined as

$$\mathbf{Adv}_F^{\mathrm{prf}}(\mathcal{A}) = \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{F(K,.)} \Rightarrow 1\right] - \Pr\left[\rho \xleftarrow{\$} \mathsf{Func}(\mathcal{D},\mathcal{R}) : \mathcal{A}^{\rho(.)} \Rightarrow 1\right]. \tag{2.2}$$

Hence the PRF advantage of the adversary $\mathcal{A}$ is computed as a difference between two probabilities, the adversary $\mathcal{A}$ is required to distinguish between two situations, the first situation is where $\mathcal{A}$ is given a uniformly chosen member of the family F (i.e., $\mathcal{A}$ has oracle access to the procedure F_K, where $K \xleftarrow{\$} \mathcal{K}$) and in the other $\mathcal{A}$ is given oracle access to a uniformly chosen element of $\mathsf{Func}(\mathcal{D},\mathcal{R})$ (i.e., $\mathcal{A}$ is given oracle access to the procedure **RndF**). If the adversary cannot tell apart these two situations then we consider F to be a pseudorandom family. In other words F is considered to be pseudorandom if for all *efficient* adversaries $\mathcal{A}$, $\mathbf{Adv}_F^{\mathrm{prf}}(\mathcal{A})$ is *small*.

In this definition we use *efficient* adversary with *small* advantage. We will never make this more precise, and this is standard with the paradigm of "concrete security" where a precise notion of efficiency and small advantage is never specified. What makes an adversary efficient and its advantage small is left to be interpreted with respect to a specific application where such an object would be used. [2]

2.4.2 Pseudorandom Permutation

Let $E : \mathcal{K} \times \mathcal{D} \to \mathcal{D}$ be a family of functions such that for every $K \in \mathcal{K}$, $E_K : \mathcal{D} \to \mathcal{D}$ is a bijection. Analogous to the definition of PRF advantage, we define the PRP advantage of an

[2]There is also a notion of asymptotic security where efficiency is equated with a polynomial time adversary and small advantage as a negligible function, but to use such an asymptotic notion we need an asymptotic definition of a function family such that there is a infinite sequence of domains and ranges for a given function and they are indexed with some countable set called the security parameter. Thus based on each security parameter we get a different family of functions and the running time of the adversary is bounded by a polynomial in the security parameter and its advantage is bounded above by a function negligible in the security parameter. Such a notion works well for complexity theoretic studies but seems to be insufficient in practical purposes where we talk of families with fixed domains and ranges.

adversary in distinguishing a random instance of the family E from a random permutation π as

$$\mathbf{Adv}_E^{\mathrm{prp}}(\mathcal{A}) = \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{E_K()} \Rightarrow 1\right] - \Pr\left[\pi \xleftarrow{\$} \mathsf{Perm}(\mathcal{D}) : A^{\pi()} \Rightarrow 1\right].$$

And, E is considered to be a pseudorandom permutation family if for all efficient adversaries $\mathcal{A}$, $\mathbf{Adv}_E^{\mathrm{prp}}(\mathcal{A})$ is small.

As every member of a permutation family has an inverse, hence in case of permutations we can define a stronger notion of pseudorandomness. Here we assume that the adversary is given two oracles one of the permutation and other of its inverse and the adversary can adaptively query both oracles. As before there are two possible scenarios, in the first scenario the adversary is provided with the oracles $E_K()$ and $E_K^{-1}()$ where $K \xleftarrow{\$} \mathcal{K}$ and in the other scenario the oracles $\pi(), \pi^{-1}()$ are provided where $\pi \xleftarrow{\$} \mathsf{Perm}(\mathcal{D})$. And the goal of the adversary is to distinguish between these two scenarios. We define the advantage of an adversary $\mathcal{A}$ in distinguishing a permutation family E from a random permutation in the $\pm$prp sense as

$$\mathbf{Adv}_E^{\pm\mathrm{prp}}(\mathcal{A}) = \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{E_K(), E_K^{-1}()} \Rightarrow 1\right] - \Pr\left[\pi \xleftarrow{\$} \mathsf{Perm}(\mathcal{D}) : \mathcal{A}^{\pi(), \pi^{-1}()} \Rightarrow 1\right],$$

and if for all efficient adversaries $\mathcal{A}$, $\mathbf{Adv}_E^{\pm\mathrm{prp}}(\mathcal{A})$ is small then we say E is a strong pseudorandom permutation (SPRP) family.

Security of Block Ciphers. As defined in Section 2.3, a block cipher is a permutation family $E : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$. Of course any such permutation family cannot be considered as a block cipher, as a block cipher should have some security properties associated with it which any permutation family will not have. Defining security of a block cipher is tricky (as is true for all cryptographic primitives), if we consider that a block cipher $E_K()$ is used to encrypt n bit strings then ideally given $E_K(X)$ one should not be able to obtain any information regarding K or X, this property can be achieved if $E_K(X)$ "looks random" to any computationally bounded adversary. In practice we consider a block cipher to be secure if it behaves like a strong pseudo-random permutation.

Unfortunately for the block ciphers that are in use we are not able to prove that they are really SPRPs. So we assume that a secure block-cipher is a SPRP, the assumption is based on our long term inability to find an efficient algorithm which can distinguish a block cipher from a random permutation. If such an algorithm is discovered then the block cipher would be broken. It is worth mentioning here that one can construct PRFs, PRPs and SPRPs based on other mathematical assumptions, in particular if we assume that one way functions exist then we can construct PRFs, PRPs and SPRPs using one way functions [80], such constructions though theoretically are more appealing but are much inefficient compared to the block ciphers in use.

2.5 Block Cipher Mode of Operation

With block ciphers we can encrypt messages of fixed length same as its block length, but in practice we need to encrypt messages which are arbitrarily long, also we need to obtain other kinds of security services which cannot be provided by a stand alone block cipher. Informally, a mode of operation is a specific way to use a block cipher to enable it to encrypt arbitrary long messages and to provide specific security services, such as data confidentiality/privacy, authentication or a combination of both. A mode of operation can be defined as a procedure that takes as input a key $K \in \{0,1\}^k$, a message $P \in \{0,1\}^*$ of arbitrary length and sometimes an initialization vector or *nonce* $IV \in \{0,1\}^v$, and produces a ciphertext $C \in \{0,1\}^*$ as its output. During the encryption process, some modes also produce a tag $\tau \in \{0,1\}^\tau$ which can be considered as a small footprint or hash value of the plaintext message (explained in Section 2.5.2). The notion of a tag value is useful for offering the security service of data integrity/authentication.

There can be various modes of operations which provides different kinds of security. Roughly the modes of operations can be classified as follows:

- Privacy only modes.
- Authenticated Encryption.
- Authenticated encryption with associated data.
- Deterministic authenticated encryption.
- Tweakable enciphering schemes.

In the subsequent sub-sections we give an overview of privacy only and authenticated encryption modes of operations. Tweakable enciphering schemes form an important component of this thesis, hence they are treated in details in Chapter 4. Deterministic authenticated encryption schemes are described in Chapter 10.

2.5.1 Privacy Only Modes

As the name suggests these modes are supposed to give privacy. A privacy only mode consists of three algorithms $P = (\mathcal{K}, \mathbf{E}, \mathbf{D})$ where $\mathcal{K}$ is the key generation algorithm (we will abuse this notation to denote the key space by $\mathcal{K}$ also). The encryption algorithm is a function $\mathbf{E} : \mathcal{K} \times \mathcal{IV} \times \{0,1\}^* \to \{0,1\}^*$, where $\mathcal{K}$ and $\mathcal{IV}$ are the key space and initialization vector (IV) space respectively and $\{0,1\}^*$ is the message and cipher space. And the decryption algorithm is $\mathbf{D} : \mathcal{K} \times \mathcal{IV} \times \{0,1\}^* \to \{0,1\}^*$, and for any $K \in \mathcal{K}$, $IV \in \mathcal{IV}$ and $M \in \{0,1\}^*$,

$\mathbf{D}(K, IV, \mathbf{E}(K, IV, M)) = M.$

These modes are considered secure if they produce ciphertexts which are indistinguishable from random strings by an adaptive chosen plain-text adversary. [3] Let $\mathcal{A}$ be an adversary attacking $\mathbf{E}$, then we define the privacy advantage of $\mathcal{A}$ as

$$\mathbf{Adv}_{\mathbf{E}}^{\text{priv}}(\mathcal{A}) = \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\mathbf{E}(K,.,.)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\$(.,.,.)} \Rightarrow 1\right] \tag{2.3}$$

The oracle $\$(.,.,.)$ returns random strings of the size same as that of the ciphertext for every query of $\mathcal{A}$. $\mathbf{E}$ is considered secure if for all efficient adversaries $\mathcal{A}$, $\mathbf{Adv}_{\mathbf{E}}^{\text{priv}}(\mathcal{A})$ is small. Depending on the mode there may be certain restrictions imposed on the adversary, as in case of most privacy only modes, it is required that the IVs are never repeated. Hence in such modes, the adversary $\mathcal{A}$ would have the restriction that it cannot make two or more queries with the same IV. In case where repetition of IVs are not allowed, the IV is called a nonce.

Some important modes which are secure in the above sense are Cipher Block Chaining (CBC), Counter, Cipher Feed Back (CFB) and Output Feedback (OFB).

2.5.2 Authenticated Encryption

The security provided by privacy only modes may not be enough in certain scenarios. Recall, for defining privacy we assumed the adversary to be an adaptive chosen plaintext adversary whose task was to distinguish the output of the mode from random strings. Thus, if an adversary sees only ciphertexts from a secure privacy only mode, he cannot determine anything meaningful from the ciphertexts. But, if we assume that the adversary wants to tamper the ciphertexts which goes through the public channel he can always do so. In a privacy only mode the receiver has no way to determine whether (s)he received the ciphertext that was originally sent by the sender. This forms a major limitation of privacy only modes.

---

[3] Adversaries, as described before are considered as probabilistic algorithms with access to oracles. They are generally classified as (a) Cipher text only (b)Chosen plain text (c) Chosen plain text and chosen cipher text (d) Adaptive chosen plain text (e) Adaptive chosen plain text and adaptive chosen cipher text. This classification is based on the resources that the adversary is provided. As the names suggests, a cipher text only adversary has access to the cipher text only (these adversaries are also sometimes called as eavesdropping adversaries), whereas a chosen plain text adversary has the capability of obtaining cipher texts of the plain texts of its choice, and analogously a chosen cipher text adversary has the capability of obtaining decryptions of the cipher texts of its choice. The adaptive versions of the chosen plain text and chosen cipher text adversaries has the added capability of interacting with the encryption/decryption algorithms in an adaptive manner, for example for an adaptive chosen plain text adversary the adversary may ask for an encryption of a plain text of its choice and then after seeing the result may decide the next message whose encryption it wants to see etc. The adaptive adversaries are considered more stronger than the non-adaptive ones.

To overcome this limitation we need to add some other functionality to a mode so that the receiver of a message can verify whether (s)he had obtained the ciphertext sent by the sender. This is obtained by a tag. A tag can be considered as a checksum of the message that was used to generate the ciphertext. A sender after decrypting the ciphertext can always compute the tag and match the tag which (s)he computed using the decrypted message with the tag that she received. If the tags do not match the receiver can know that a tampering of the ciphertext has taken place during the transit. This functionality in the symmetric setting is called authentication and the modes which provides both privacy and authentication are called authenticated encryption modes.

Thus an authenticated encryption mode can be seen as a collection of three algorithms $\Pi = (\mathcal{K}, \boldsymbol{E}, \boldsymbol{D})$ where $\boldsymbol{E} : \mathcal{K} \times \mathcal{N} \times \{0,1\}^* \to \{0,1\}^*$, and $\boldsymbol{D} : \mathcal{K} \times \mathcal{N} \times \{0,1\}^* \to \{0,1\}^* \cup \{\bot\}$. Where $\mathcal{K}$ and $\mathcal{N}$ are the key space and nonce space respectively. Nonce is an IV which is never repeated. The ciphertext C produced by $\boldsymbol{E}$ can be parsed as $\mathsf{C} = (C, \mathsf{tag})$. Where $\mathsf{tag} \in \{0,1\}^\tau$ is a fixed length string which is called the authentication tag. The decryption algorithm $\boldsymbol{D}$ on an input C produces the corresponding plaintext P or outputs $\bot$ if the computed tag does not match tag.

The security of an authenticated encryption protocol consists of two parts – privacy and authenticity. The adversary is given access to the encryption oracle and is assumed to be nonce respecting, i.e., it does not repeat a nonce in its queries to the oracle. Following Rogaway [118], the privacy of a encryption scheme $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ against a nonce respecting adversary $\mathcal{A}$ is defined in the sense of "indistinguishability from random strings" in the following manner:

$$\mathbf{Adv}_{\Pi}^{\mathrm{priv}}(\mathcal{A}) = \Pr[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\boldsymbol{E}_K(\cdot,\cdot)} \Rightarrow 1] - \Pr[\mathcal{A}^{\$(\cdot,\cdot)} \Rightarrow 1],$$

where $\$(\cdot,\cdot)$ is an oracle that takes (N, M) as input and returns $|M| + |\mathsf{tag}|$ many random bits as output.

For defining authenticity, we stress that no adversary should be able to create a valid ciphertext which gets decrypted. To put this formally, we allow the adversary an oracle access to $\boldsymbol{E}_K()$ and finally it is required to output a pair $(N, (C, \mathsf{tag}))$ such that (C, tag) was not a response of its oracle for some query (N, M). If the adversary outputs such a pair $(N, (C, \mathsf{tag}))$ and $\boldsymbol{D}_K(N, (C, \mathsf{tag})) \neq \bot$, then we say that the adversary has committed a *forgery*. Formally, we define the authenticity advantage of an adversary $\mathcal{A}$ as

$$\mathbf{Adv}_{\Pi}^{\mathrm{auth}}(\mathcal{A}) = \Pr[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\boldsymbol{E}_K(\cdot,\cdot)} \text{ forges}].$$

For a authenticated encryption scheme Π to be secure, it is required that for all efficient adversaries $\mathcal{A}$, both $\mathbf{Adv}_{\Pi}^{\mathrm{priv}}(\mathcal{A})$ and $\mathbf{Adv}_{\Pi}^{\mathrm{auth}}(\mathcal{A})$ are small.

Some secure authenticated encryption schemes are OCB [118], IAPM [79], CCM [41], EAX

[9], GCM [99] etc.

Another class of AE schemes are called Authenticated Encryption with Associated Data (AEAD). These schemes can be useful in certain realistic scenarios. Like if we consider network packets, we do not want to encrypt the headers but we want to authenticate the headers so that they cannot be tampered. Such schemes takes as input the message and an associated data (the packet header in this case), the message is only encrypted but the tag is produced both with the message and the header. Most AE schemes can be easily converted into AEADs. Another related type of modes are deterministic authenticated encryption (DAE) schemes. DAEs are treated in Chapter 10 and Chapter 11.

2.6 Game Based Security Proofs

In the setting of "provable security" we provide a security proof for a cryptographic scheme. Proving security of a cryptographic scheme generally involves the following steps:

1. Given a scheme Ψ we first precisely define what we mean by Ψ to be secure. Such a definition is generally given in terms of an interaction between an adversary $\mathcal{A}$ and the scheme Ψ. Example of a typical security definition is as expressed in Eq. (2.3), where the advantage of an adversary $\mathcal{A}$ in breaking a privacy only mode of operation $\mathbf{E}$ is provided. Further it is stated that $\mathbf{E}$ is secure if the advantage of any efficient adversary as defined in Eq. (2.3) is small. Thus, a security statement is tied with a specific event involving the interaction of the scheme and the adversary $\mathcal{A}$, and the probability that such an event occurs for all efficient $\mathcal{A}$ should be small.

2. The security proof consists of arguments that shows that a scheme really have the properties specified in the security definition. In computational security it is almost never possible to show that a scheme satisfies a security definition unconditionally. The security definition is satisfied by a cryptographic construction based on some assumptions. In general a scheme is constructed using some basic primitives, for example a mode of operation is constructed using block ciphers. The primitives used to build the scheme are assumed to be secure in some sense, and the security of Ψ is proved based on this assumption. The proof technique involves a reduction which shows that if the scheme is insecure then the primitive is also insecure. Thus a security theorem is not an absolute statement and needs to be interpreted carefully. [4]

The reductionist argument used to prove security of a scheme involves computing probabilities in certain probability spaces which involves randomness of the scheme and also the

[4] In the recent years there have been some criticisms to the paradigm of provable security and there have been proposals that given the relative nature of a security theorem and its proof they should not be called as security proofs but as reductionist arguments [83, 84].

adversary. Such computations becomes difficult if the arguments are not well structured. A popular way of constructing such reductions is the *game playing technique* or *sequence of games*. We have extensively used the game playing technique to prove security of our proposals. In this section we will give a brief overview of the technique including an example. The material in this section is mostly taken from [133], but we recast it according to our needs using our notations. The reader is referred to [133] and [8] for more detailed examples and discussions about the technique.

In the technique of sequence of games, the interaction of the adversary with a scheme is seen as an attack game. The attack game can be written as an algorithmic procedure involving the adversary and its oracles. The initial game (call it G_0) is the original attack game, and this game specifies an event S_0 whose probability is to be computed, and if the probability of S_0 is near to a target value then one can conclude that the scheme is secure. But computing $\Pr[S_0]$ may sometimes be very difficult in the original attack game G_0. So, one does small modifications to the original game to obtain a new game G_1, further one changes G_1 to obtain G_2 etc. and thus obtain a sequence of games $G_0, G_1, G_2, \ldots, G_n$. Similar to the event S_0 in G_0, we can specify an event S_i associated with each game G_i. The sequence of games is designed in such a manner such that for each i, one can easily bound the quantity $|\Pr[S_i] - \Pr[S_{i+1}]|$ for $0 \leq i \leq n-1$ and also easily find a bound on $\Pr[S_n]$. Further using these bounds one can find a bound on $\Pr[S_0]$, as S_0 is the event tied with the security of the scheme hence proving a bound on $\Pr[S_0]$ proves the security of the scheme.

To construct proofs using games it is desirable that the changes between two consecutive games are very small so that the analysis of that change becomes very simple. There can be various types of transitions between two consecutive games as described in [133], among them a transition called as "transition based on failure" is very interesting and useful, we describe such transitions in more details next.

Transitions based on failure events. In this kind of transition, the two games proceed identically unless some failure event F occurs. The two games should be defined on the same probability space and the unique difference between them are the rules to compute certain random variables. To make things concrete let us consider two consecutive games G_i and G_{i+1} in the game sequence, and let S_i and S_{i+1} be the events of interest tied to the respective games. Moreover, we assume that the games G_i and G_{i+1} proceed identically unless a failure event F occurs, which is equivalent to saying,

$$S_i \wedge \neg F \iff S_{i+1} \wedge \neg F. \tag{2.4}$$

As both events $S_i \wedge \neg F$ and $S_{i+1} \wedge \neg F$ are the same, so we can use the following lemma to bound $|\Pr[S_i] - \Pr[S_{i+1}]|$ if the condition in Eq. (2.4) is satisfied.

Lemma 2.1. *[The Difference Lemma or the Fundamental Lemma of Game Playing] Let A, B and F be events defined in some probability distribution and suppose that $A \wedge \neg F \iff$*

$B \wedge \neg F$. *Then* $|\Pr[A] - \Pr[B]| \leq \Pr[F]$.

Proof.

$$\begin{aligned} |\Pr[A] - \Pr[B]| &= |\Pr[A \wedge \neg F] + \Pr[A \wedge F] - \Pr[B \wedge \neg F] - \Pr[B \wedge F]| && (2.5)\\ &= |\Pr[A \wedge F] - \Pr[B \wedge F]| && (2.6)\\ &= |\Pr[F](\Pr[A|F] - \Pr[B|F])| && (2.7)\\ &\leq \Pr[F] && (2.8) \end{aligned}$$

It is easy to see that Eq. (2.6) follows from Eq. (2.5) by the hypothesis. Equation (2.7) follows from Eq. (2.6) by the definition of conditional probabilities, and the final inequality uses the fact that the absolute value of the difference of two probabilities can be at most 1. □

Thus, using the difference lemma, it sometimes becomes very simple to bound the difference $|\Pr[S_i] - \Pr[S_{i+1}]|$ whenever $\Pr[S_i \wedge \neg F] = \Pr[S_{i+1} \wedge \neg F]$, we will see a specific use of this in the example that follows.

2.6.1 An Example

We now give a concrete example of a proof using the game playing technique. Suppose we are given a PRF family $F : \mathcal{K}_1 \times \{0,1\}^\ell \to \{0,1\}^\ell$ and another family of functions $H : \mathcal{K}_2 \times \{0,1\}^{\ell_1} \to \{0,1\}^\ell$, where $\ell_1 > \ell$. Also we are told that family H is $\epsilon-$almost universal ($\epsilon-$AU), which we define next

Definition 2.1. *[$\epsilon-$almost universal] Let $H : \mathcal{K}_2 \times \{0,1\}^{\ell_1} \to \{0,1\}^\ell$ be a family of functions. H is called a $\epsilon-$almost universal ($\epsilon-$AU) family if for all $w, w' \in \{0,1\}^{\ell_1}$ where $w \neq w'$,*

$$\Pr[K \xleftarrow{\$} \mathcal{K}_2 : H_K(w) = H_K(w')] \leq \epsilon. \tag{2.9}$$

Using the families H and F we wish to construct a new PRF family $F' : \mathcal{K} \times \{0,1\}^{\ell_1} \to \{0,1\}^\ell$. We claim the following:

> Let $\mathcal{K} = \mathcal{K}_1 \times \mathcal{K}_2$, if $F' : \mathcal{K} \times \{0,1\}^{\ell_1} \to \{0,1\}^\ell$ is defined as
>
> $$F'_{(K_1,K_2)}(X) = F_{K_1}(H_{K_2}(X)),$$
>
> then F' is a PRF family.

This can be put more precisely as follows

Claim 2.1. *Let $\mathcal{A}$ be an arbitrary PRF adversary attacking F' which makes at most q queries to its oracle and runs for time t, then there exist a PRF adversary $\mathcal{B}$, such that*

$$\mathbf{Adv}_{F'}^{\mathrm{prf}}(\mathcal{A}) \leq \mathbf{Adv}_{F}^{\mathrm{prf}}(\mathcal{B}) + \frac{\epsilon q^2}{2} \tag{2.10}$$

moreover $\mathcal{B}$ also makes q queries and runs for time $O(t)$.

The statement of Claim 2.1 embodies a typical security statement. It would be worthwhile to analyze a bit why the above claim is equivalent to saying that F' is a PRF if F is a PRF and H a $\epsilon-$AU. Note that the statement claims that for any arbitrary PRF adversary $\mathcal{A}$ for the function F' there always exist a PRF adversary $\mathcal{B}$ for F such that their advantages are related as in Eq. (2.10). As we assume that F is a PRF hence the advantage of any efficient adversary (including $\mathcal{B}$) in attacking F is bound to be small, and as ϵ is also small hence the bound for the advantage claimed in Eq. (2.10) indeed suggests that the advantage of any efficient adversary (who uses reasonable number of queries and runs for a reasonable time) would have a small advantage in breaking F', which in turn suggests that F' is a PRF. Now we construct a proof of the above claim using the sequence of games.

Proof of claim: We construct a PRF adversary $\mathcal{B}$ which runs $\mathcal{A}$, the description of $\mathcal{B}$ is given in Figure 2.3.

Adversary $\mathcal{B}^{\mathcal{O}}$
1. $k_1 \xleftarrow{\$} \mathcal{K}_1$;
2. **While** $\mathcal{A}$ queries x, do the following:
2.1 $y \leftarrow H_{K_2}(x)$
2.2 $z \leftarrow \mathcal{O}(y)$
2.3 **return** z
3. **Until** $\mathcal{A}$ stops querying and returns a bit b
4. **return** b

Figure 2.3: The adversary $\mathcal{B}$.

Note that $\mathcal{B}$ is a PRF adversary for the function F and $\mathcal{A}$ is the same for the family F'. Hence, $\mathcal{B}$ has an access to an oracle, which can be an instance of the family F on a random function chosen uniformly at random from $\mathsf{Func}(\ell, \ell)$. In Figure 2.3, $\mathcal{B}$'s oracle is depicted by $\mathcal{O}$. $\mathcal{B}$ runs $\mathcal{A}$ in the sense that it provides answers to $\mathcal{A}$'s oracle queries (this is depicted in lines 2.1-2.3 in the description in Figure 2.3). When $\mathcal{A}$ finishes querying, it outputs a bit b and $\mathcal{B}$ in turn outputs the same bit b. We will assume that $\mathcal{A}$ never repeats a query, as it will not gain anything by repeating a query, as given either of the oracles the same query would return the same answer. This assumption makes our description a bit simpler, but in no way affects the generality of the result.

If the oracle $\mathcal{O}$ of $\mathcal{B}$ is a uniform random instance of the family F then $\mathcal{B}$ provides a perfect simulation for a random instance of the family F' to $\mathcal{A}$, also $\mathcal{B}$ outputs the same bit that $\mathcal{A}$ outputs. Thus, we can say that

$$\Pr\left[K_1 \stackrel{\$}{\leftarrow} \mathcal{K}_1 : \mathcal{B}^{F_{K_1}(.)} \Rightarrow 1\right] = \Pr\left[K_1 \stackrel{\$}{\leftarrow} \mathcal{K}_1, K_2 \stackrel{\$}{\leftarrow} \mathcal{K}_2 : \mathcal{A}^{F'_{K_1,K_2}(.)} \Rightarrow 1\right]. \tag{2.11}$$

Now we shall like consider the interaction of $\mathcal{A}$ with $\mathcal{B}$ with the help of a game. Assuming that $\mathcal{B}$ is provided with a random instance of F, $\mathcal{A}$'s interaction with $\mathcal{B}$ can be seen as in the game *Game***0** shown in Figure 2.4.

Initialization:
$K_1 \stackrel{\$}{\leftarrow} \mathcal{K}_1$;
$K_2 \stackrel{\$}{\leftarrow} \mathcal{K}_2$;
Respond to the sth query w^s of $\mathcal{A}$ as follows:
1. $d^s \leftarrow H_{K_2}(w^s)$;
2. $r^s \leftarrow F_{K_1}(d^s)$;
3. **return** r^s;

Figure 2.4: Game *Game***0**.

It is straight forward to see that

$$\Pr[K_1 \leftarrow \mathcal{K}_1, K_2 \leftarrow \mathcal{K}_2 : \mathcal{A}^{F'_{K_1,K_2}(\cdot)} \Rightarrow 1] = \Pr[\mathcal{A}^{Game\mathbf{0}} \Rightarrow 1], \tag{2.12}$$

and using Eq. (2.11) we have

$$\Pr\left[K_1 \stackrel{\$}{\leftarrow} \mathcal{K}_1 : \mathcal{B}^{F_{K_1}(.)} \Rightarrow 1\right] = \Pr[\mathcal{A}^{Game\mathbf{0}} \Rightarrow 1]. \tag{2.13}$$

Now we make a change in *Game***0**, by substituting the function $F_{K_1}()$, with a truly random function, and we name the changed game as *Game***1** which is shown in Figure 2.5.

In *Game***1** the function F is no more used, but it is replaced with a random function. The functionality of a random function whose domain and range is the set $\{0,1\}^\ell$ is provided by the subroutine $\delta()$. Note that the description of δ is similar as the description of the random function **RndF**() as shown in Figure 2.1. The subroutine $\delta(x)$ maintains a flag called `bad`, which is set to true if the function is called on the same input more than once also it maintains a table T, where $T[x]$ stores the value returned for x. In addition to the subroutine $\delta(x)$, the description in Figure 2.5 consists of a initialization procedure (lines 101-103) and the specific procedure with which a query w^s of $\mathcal{A}$ is responded.

Subroutine $\delta(x)$
01. $Y \xleftarrow{\$} \{0,1\}^\ell$;
02. **if** $x \in \mathsf{Dom}(\delta)$ **then**
03. $\mathsf{bad} \leftarrow \mathsf{true}$;
04. $\boxed{Y \leftarrow T[x]}$;
05. **end if;**
06. $T[x] \leftarrow Y$;
07. $\mathsf{Dom}(\delta) \leftarrow \mathsf{Dom}(\delta) \cup \{x\}$;
8. **return** Y;
Initialization:
101. $K_2 \xleftarrow{\$} \mathcal{K}_2$;
102. $\mathsf{Dom}(\delta) \leftarrow \emptyset$;
103. $\mathsf{bad} \leftarrow$ false;
Respond to the sth *query* w^s *of* $\mathcal{A}$ *as:*
201. $d^s \leftarrow H_{k_2}(w^s)$;
202. $r^s \leftarrow \delta(d^s)$;
203. **return** r^s;

Figure 2.5: Games *Game***1** and *Game***2**. The full description is of *Game***1**, *Game***2** is obtained from Game1 by removing the boxed entry in line **04**.

Given the description of Game 1 and the adversary $\mathcal{B}$, we conclude that

$$\Pr\left[\rho \xleftarrow{\$} \mathsf{Func}(\ell,\ell) : \mathcal{B}^{\rho(.)} \Rightarrow 1\right] = \Pr[\mathcal{A}^{Game\mathbf{1}} \Rightarrow 1], \tag{2.14}$$

as the game *Game***1** provides the same environment to $\mathcal{A}$ as $\mathcal{B}$ would have provided it if the oracle $\mathcal{O}$ for $\mathcal{B}$ was a random function.

Now, using Eq. (2.13), Eq. (2.14) and the definition of the PRF advantage for the adversary $\mathcal{B}$ we obtain

$$|\Pr[\mathcal{A}^{Game\mathbf{0}} \Rightarrow 1] - \Pr[\mathcal{A}^{Game\mathbf{1}} \Rightarrow 1]| = \mathbf{Adv}_F^{\mathrm{prf}}(\mathcal{B}). \tag{2.15}$$

Now, we make a small change to *Game***1** to obtain a new game *Game***2**. *Game***1** is transformed to *Game***2** by removing the instruction inside the box in line **04** of the subroutine $\delta(x)$. With this change, $\delta(x)$ is no more a function, as it may produce two different outputs when called on the same input twice. The interesting point to note is that *Game***1** and *Game***2** proceeds in the same way if the bad flag is not set, in other words if the function δ is only called with distinct inputs. The setting of the bad flag can be considered as a failure event, and thus using Lemma 2.1, we obtain

$$|\Pr[\mathcal{A}^{Game\mathbf{1}} \Rightarrow 1] - \Pr[\mathcal{A}^{Game\mathbf{2}} \Rightarrow 1]| \leq \Pr[\mathcal{A}^{Game\mathbf{2}} \text{ sets } \mathsf{bad}]. \tag{2.16}$$

Now, we do some syntactical changes to the game *Game***2** to obtain a new game *Game***3**. The *Game***3** is illustrated in Figure 2.6, in this game the function $\delta(\cdot)$ is eliminated, and it runs in three distinct phases. In the initialization phase, a random key is selected from $\mathcal{K}_2$, a multiset $\mathcal{D}$ is initialized to the empty set and the bad flag is set to false. The query w^s is responded with a string drawn uniformly at random from $\{0,1\}^\ell$. Finally there is a finalization stage which runs in two phases, in the first phase certain values are inserted in the multiset $\mathcal{D}$ and in the second phase it is checked if there is any collision between the values in $\mathcal{D}$, if a collision is found then the bad flag is set to true.

Initialization:
01. $K_2 \xleftarrow{\$} \mathcal{K}_2$; **02.** $\mathcal{D} \leftarrow \emptyset$; **03.** bad $\leftarrow$ false;
Respond to a query w^s of $\mathcal{A}$ as follows:
101. $r^s \leftarrow \{0,1\}^\ell$; **102.** **return** r^s;
Finalization:
<u>**First Phase:**</u> **201.** **for** $s \leftarrow 1$ to q, **202.** $d^s \leftarrow H_{k_2}(w^s)$; **202.** $\mathcal{D} \leftarrow \mathcal{D} \cup \{d^s\}$; **203.** **end for**; <u>**Second Phase:**</u> **if** (some value occurs more than once in $\mathcal{D}$) **then** bad $\leftarrow$ true;

Figure 2.6: Game *Game***3**.

It is easy to see that *Game***3** is just *Game***2** cast in a different way, the distribution of the responses that $\mathcal{A}$ obtains in *Game***3** is exactly the same that it would get from *Game***2**, thus

$$\Pr[\mathcal{A}^{Game\mathbf{2}} \Rightarrow 1] = \Pr[\mathcal{A}^{Game\mathbf{3}} \Rightarrow 1], \tag{2.17}$$

also

$$\Pr[\mathcal{A}^{Game\mathbf{2}} \text{ sets } \mathsf{bad}] = \Pr[\mathcal{A}^{Game\mathbf{3}} \text{ sets } \mathsf{bad}]. \tag{2.18}$$

Also, as we have assumed that $\mathcal{A}$ does not repeat any query thus Game**3** provides it with

an environment which is exactly same as the environment where $\mathcal{A}$ interacts with a random function with domain $\{0,1\}^{\ell_1}$ and range $\{0,1\}^{\ell}$, thus

$$\Pr[\mathcal{A}^{Game\mathbf{3}} \Rightarrow 1] = \Pr[\rho \xleftarrow{\$} \mathsf{Func}(\ell_1, \ell) : \mathcal{A}^{\rho()} \Rightarrow 1]. \tag{2.19}$$

Thus, using Eq. (2.12), Eq. (2.19) and the definition of PRF advantage of $\mathcal{A}$ we have

$$|\Pr[\mathcal{A}^{Game\mathbf{0}} \Rightarrow 1] - \Pr[\mathcal{A}^{Game\mathbf{3}} \Rightarrow 1]| = \mathbf{Adv}^{\mathrm{prf}}_{F'}(\mathcal{A}), \tag{2.20}$$

and from eqs. (2.15), (2.16), (2.17) and Eq. (2.18) it follows

$$|\Pr[\mathcal{A}^{Game\mathbf{0}} \Rightarrow 1] - \Pr[\mathcal{A}^{Game\mathbf{3}} \Rightarrow 1]| \leq \mathbf{Adv}^{\mathrm{prf}}_{F}(\mathcal{B}) + \Pr[\mathcal{A}^{Game\mathbf{3}} \text{ sets } \mathsf{bad}]. \tag{2.21}$$

Finally, using eqs. (2.20) and (2.21), we have

$$\mathbf{Adv}^{\mathrm{prf}}_{F'}(\mathcal{A}) \leq \mathbf{Adv}^{\mathrm{prf}}_{F}(\mathcal{B}) + \Pr[\mathcal{A}^{Game\mathbf{3}} \text{ sets } \mathsf{bad}]. \tag{2.22}$$

To get the desired bound we now need to compute $\Pr[\mathcal{A}^{Game\mathbf{3}} \text{ sets } \mathsf{bad}]$. To compute this probability we need to compute the probability of collisions in the multiset $\mathcal{D}$ in *Game***3**. According to the specification of *Game***3**, the multiset $\mathcal{D}$ contains the following elements:

$$\mathcal{D} = \{H_{K_2}(w^s) : 1 \leq s \leq q\}.$$

Where w^s is the sth query specified by $\mathcal{A}$. Now, as $\mathcal{A}$ does not repeat queries hence we have $w^s \neq w^{s'}$ for $1 \leq s \leq s' \leq q$. As, H is an ϵ-AU family, hence for all $s \neq s'$, we have

$$\Pr[H_{K_2}(w^s) = H_{K_2}(w^{s'})] \leq \epsilon. \tag{2.23}$$

Let COLD be the event that there is a collision in the set $\mathcal{D}$, then using Eq. (2.23) and the union bound we have

$$\begin{aligned} \Pr[\mathcal{A}^{Game\mathbf{3}} \text{ sets } \mathsf{bad}] &= \Pr[\mathsf{COLD}] \\ &\leq \epsilon\binom{q}{2} \\ &< \frac{\epsilon q^2}{2}. \end{aligned} \tag{2.24}$$

Hence from Eqs. (2.22) and (2.24)

$$\mathbf{Adv}_{F'}^{\mathrm{prf}}(\mathcal{A}) \leq \mathbf{Adv}_{F}^{\mathrm{prf}}(\mathcal{B}) + \frac{\epsilon q^2}{2}. \tag{2.25}$$

Moreover it is clear from the description of $\mathcal{B}$ that the number of queries asked by $\mathcal{B}$ and its running time is as claimed. □

2.7 Summary

In this Chapter we discussed the basic mathematical objects which we would use through out this thesis. In particular we discussed that binary strings of length n would be treated as elements in a finite field $\mathbb{F}_{2^n}$ and thus field operations would be applied to strings. We also discussed syntax of a block cipher and two important block cipher modes of operation, namely privacy only modes and authenticated encryption. We also discussed two important mathematical objects called pseudorandom functions and permutations and discussed how security of block ciphers can be defined with the use of pseudorandom permutations. Finally we gave a detailed example of a security proof using the technique of game playing. This technique would be repeatedly used later.

Chapter 3

Reconfigurable Hardware

It's better to die on your feet than live on your knees!

Emiliano Zapata

A significant part of the rest of the thesis deals with designing efficient hardware for storage encryption schemes. All our designs are directed towards Field Programmable Gate Arrays (FPGAs). To do a good use of FPGAs we need to know how they work, if we know the internal structure of them we can better exploit their

specific structure to achieve good performance of our implementations both in area and time. In this Chapter we give a brief introduction about the architecture of reconfigurable hardware focused on FPGAs. We start with a brief history of reconfigurable computing and then describe some basic architectural issues in FPGA. Finally we provide a description of the characteristics of Xilinx Spartan 3 FPGAs and also discuss some main features of Xilinx Virtex 5 and Lattice ICE40. Later in our work we have extensively used these families of FPGAs for our implementations.

3.1 Reconfigurable Computing: A Brief History

In very general terms reconfigurable computing can be described as a kind of computer architecture which combines the flexibility of software with the high performance of hardware. We are familiar with general purpose computing machines which are constructed using microprocessors. These machines consists of a fixed circuitry which can be exploited for various computing needs through the use of softwares. This paradigm provides a lot of flexibility in terms of the various tasks that a single static circuit can be made to perform. But this flexibility comes at the cost of low performance. On the other hand, one can have dedicated circuits or customs hardware for a specific task, as is obtained by application specific integrated circuits (ASICS). Such custom hardware does not provide flexibility but one can

obtain very high performance as the hardware can be constructed in a very optimized way which only supports a specific task. In reconfigurable computing both high performance and flexibility can be achieved. One of the most dominant paradigms of reconfigurable architectures are the field programming gate arrays (FPGAs). The principal difference of FPGAs compared to ordinary microprocessors is the ability to make substantial changes to the datapath itself in addition to the control flow. On the other hand, the main difference with custom hardware is the possibility to adapt the hardware during runtime by *loading* a new circuit on the reconfigurable fabric.

The initial ideas of reconfigurable computing are credited to Gerald Estrin [1]. In [44], Estrin himself talks about the circumstances which gave birth to this paradigm and the early researchers who were responsible for nurturing these ideas. In particular, Estrin says that the idea occurred to him as a response to a challenge posed by John Pasta in 1959. Pasta felt that at that time the computer manufactures were more eager to serve the growing market of computing machines by providing incremental improvements over the existing paradigm of computer architecture. Estrin in [44] says

> *...he (John Pasta) challenged me to propose new ways to organize computer systems in the hope that research advances in the public domain would lead to a surge of computer development in the private domain.*

As a response to Pasta's challenge Gerald Estrin published the initial ideas of reconfigurable computing in 1960 [43]. The architecture proposed in [43] consisted of a general purpose microprocessor interconnected with programmable logic devices, such that the entire system may be temporarily distorted into a problem oriented special purpose computer. Thus, the architecture achieved both the flexibility of software and the speed of hardware platforms. The architecture described in [43] was called a Fixed plus Variable Structure Computer (FpVSC). Immediately after publication of [43], Estrin and some of his students and colleagues published works describing various domains where a FpVSC can serve non-trivial computing needs [23, 45].

Though the ideas of reconfiguration were there from the early 1960s, they were not accepted as a viable paradigm for computing for many years, in particular, until the mid 1980s there were no significant impact of these ideas in the fast growing computing and semiconductor devices market. In the mid 1970s a device named Programmable Array Logic (PAL) was introduced commercially by Monolithic Memories Inc. The PALs can be considered as the first Programmable Logic Device (PLD). PALs though programmable could be programmed only once. The structure of earlier PALs consisted of logic gate arrays and they had no clocked memory elements. PALs can be considered as an early member of the commercial reconfigurable devices. PALs still continue in the market with some modifications and added functionalities, an example of an existing family is the TIBPAL16 [137] manufactured by Texas Instruments.

Later Generic Array Logic (GAL) was introduced in 1983 by Lattice Semiconductors. GAL appears as an improvement of PAL with the characteristic that they can be erased and reprogrammed. GALs were the precursors to more useful devices called Complex PLD (CPLD). A CPLD can emulate the computational power of hundreds of thousands of logic gates, they are still popular mostly due to their low cost, some CPLD devices can be bought for less than one dollar. Today's CPLDs are non-volatile electrically-erasable programmable devices which has a structure composed of several GAL blocks interconnected by a switching matrix used for programming the interconnection and the input/outputs pins. Cool-Runner II CPLD Family [143] is a CPLD still in use.

The rebirth of the reconfigurable computing paradigm took place in a big way in the mid 1980s. By that time the semiconductor industry has experienced considerable growth and there were numerous companies manufacturing custom chips (ASICs) for diverse clients. ASIC manufacturing as is today had always been a cost intensive business. If a company wanted to create its own chips it had to be an integrated device manufacturer (IDM), i.e., it had not only to design them but also assume responsibility of fabrication (commonly called fab). Fabrication requires a special facility in terms of complex material and human resources. Thus manufacturing their own chips was really out of question for cost constrained businesses. Moreover, as discussed in [3], in the mid 1980s Moore's law was in full swing, i.e., to stay in business the chip manufacturers had to double the number of transistors in their chips every 22 months. This meant frequent upgrade or complete change of the fabrication facilities and equipments. Manufacturers were having a tough time to win this race. This background along with the genius and far-sight of three engineers Ross Freeman, Jim Barnett and Barnie Vonderschmitt lead to the foundation of Xilinx Inc. Xilinx wanted to offer to the world a new way of seeing things, a "fab-less" chip which they called a Field Programmable Logic Array (FPGA), a single chip which can be reprogrammed to perform various tasks. The first commercial FPGA, the XC2064, was manufactured by Xilinx in 1985 [3]. Over the years the FPGAs technology has had a very fast development. The first FPGAs had just a matrix of Configurable Logic Blocks (CLBs) interconnected by an intricate array of switch matrices, nowadays modern FPGAs have embedded modules such as dedicated units for Digital Signal Processing, distributed RAM blocks etc. The advances in the integration capacity allow to have in the same chip the reconfiguration part of an FPGA and one or more microprocessors, this hybrid technology is called Configurable System on Chip (CSoC). Xilinx Virtex 2 Pro (discontinued) [149], Virtex 4 [147] and Virtex 5 [146] which include one or more embedded Power PC microprocessors are examples of CSoC technology. Actually there are many manufacturers of FPGAs such as Achronix, Actel, Altera, Lattice, Quick Logic and Xilinx. So the market of FPGAs is very large and the development of their technology is advancing every day.

Currently FPGAs are widely used for many applications not only to make fast prototypes, but are also used as components in embedded systems. Some areas where FPGAs are used are industrial control applications [106], network infrastructure security [33], image

processing [6], Video Processing [36] and cryptography [117].

3.2 Field Programmable Gates Arrays

Logic and interconnections are the main resources in an FPGA, even though modern FPGAs contain more embedded components such as multipliers or memory blocks. The logic part allow us to perform arithmetic and/or logical functions while interconnections consist of the physical routing between logic blocks. In the following subsections we describe in details the logic and interconnection elements in FPGAs.

3.2.1 Logic Elements

Every digital system, however complex it may be, can be represented using Boolean functions which relates the system inputs to the outputs. The Boolean functions representing a digital system can depend on their outputs (sequential circuit) or not (combinatorial circuit). A common and simple way to represent Boolean functions is by its truth table. A truth table of a Boolean function f is an enumeration of its outputs for all possible inputs. The truth table representation of a digital system is the computational heart of a FPGA. Lookup tables (LUTs) are hardware elements that can easily implement a truth table. LUTs have N inputs and only one output, i.e, each LUT can represent one of the 2^{2^N} possible Boolean functions on N variables. Functions whose number of inputs are greater than N can be implemented using several LUTs connected in cascade. The size of LUT has been widely studied [120], empirical studies have shown that the 4-input LUT structure gives a better trade-off between area and delay [15]. Traditionally the main computing elements in FPGAs have been 4 input LUTs. This is true for many families of FPGAs like Spartan 3, Virtex 2 pro, Virtex 4 etc. Recently Xilinx has released SRAM-based families of FPGAs such as Virtex 5, Virtex 6 [152] and Virtex 7 [152] with 6-input LUT architecture.

Physically a LUT can be implemented simply using one multiplexer and one memory. Figure 3.1 shows two different ways to implement a 4 input LUT.

Inside the FPGA, LUTs are grouped into configurable logic blocks (CLB), the number of LUTs in each logic block have also been investigated [35]. The organization of LUTs varies among families and manufacturers of FPGAs, for example in Xilinx FPGAs the elementary logic block is called slice, Altera FPGAs use logic blocks called Adaptive Logic Modules (ALMs), Actel uses logic blocks called VersaTile and Lattice uses Logic Cells. In addition to LUTs the logic blocks can contain more components like multiplexors, some additional logic, carry inputs or registers. In Figure 3.2 an architecture for a very simple CLB is shown, it has a single 4-input LUT and a D-type flip-flop (labeled as FFD1), the flip-flop serves the purpose of synchronizing the output. The additional xor gate labeled as XOR_A is to

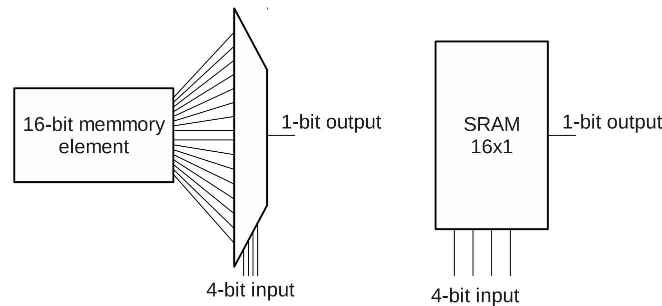


Figure 3.1: Two different ways to implement a 4 input LUT.

allow to handle a carry, Mux1 selects from the output of LUT or the xor operation between output of LUT and the carry and Mux2 is used to select an unregistered or registered output.

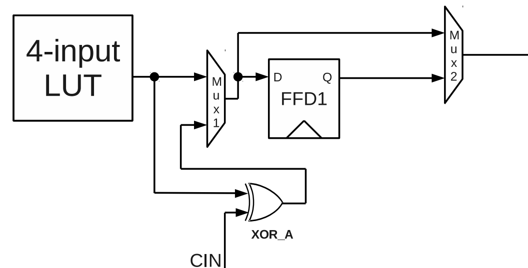


Figure 3.2: Simple Logic Block including a 4-input LUT, registered output and auxiliary XOR gate for carry chain.

3.2.2 Interconnection Resources

Logic blocks need to be interconnected with each other in order to realize complex digital systems, these interconnections within FPGAs are also configurable. Configurable blocks are generally placed in matrix structure and many vertical and horizontal tracks run between them. These tracks are used to make the interconnections. This specific placement is called the *island model* and is depicted in Figure 3.3.

There can be many ways to interconnect the logic blocks, we will describe some of them below.

Nearest neighbor

This is the simplest interconnection structure, it is illustrated in Figure 3.4. In this type of interconnection, all logic block have bidirectional connection with its nearest neighbors in

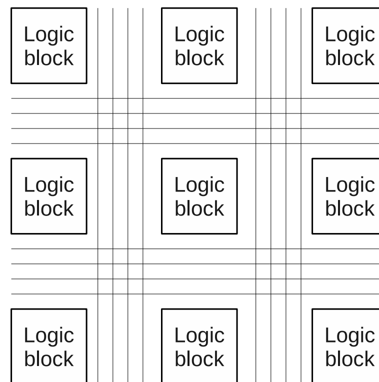


Figure 3.3: Island placement of logic block within an FPGA.

each direction: north, south, east and west. There are no interconnection resources which are capable of by passing logic blocks, so the signals must go through each logic block. And this increases the delay. Each CLB is connected to only one other CLB in each direction, this impose a limitation on the interconnection capabilities. But this kind of structure has the characteristic that logic blocks have direct connectivity between them. For that reason it is actually used in commercial FPGAs but hybridized with some other complex techniques that we explain next.

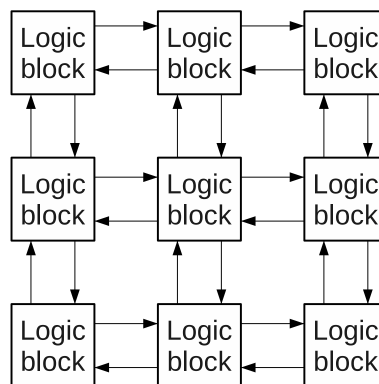


Figure 3.4: Nearest neighbor interconnection structure.

Segmented

This structure follows the model presented in Figure 3.3, i.e, in contrast to nearest neighbor, it has tracks independent of the logic blocks. The segmented structure is shown in Figure 3.5, the main components in a segmented type interconnection are as follows:

- **Tracks:** There are vertical and horizontal tracks grouped as buses. These tracks are connected with logic blocks through Connection Blocks (CBs), where the horizontal

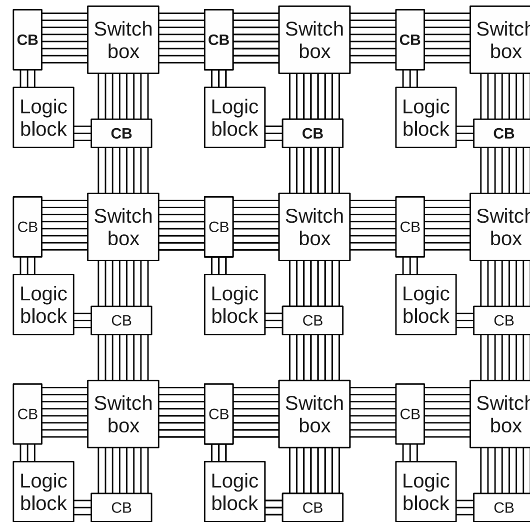


Figure 3.5: Routing structure with logic blocks placed in islands and using connection blocks and switch boxes.

and vertical tracks cross there are switch boxes to do interconnections between them.

- **Connection Blocks (CBs):** They allow inputs and outputs of logic blocks reach both horizontal and vertical tracks, each input or output can be connected to several tracks as is shown in the left side of Figure 3.6. We can see that inputs/outputs can be connected to arbitrary tracks, one only have to select which connections are activated.

- **Switch Boxes:** These are located in the crossings between vertical and horizontal tracks, in the right side of Figure 3.6 we present a detailed view of a switch box. Programmable switches allow to lay out routes between near on far logic blocks, depending on their design for example a connection could be to turn the corner in either direction or continue straight. Switch box design is a research area in itself, some interesting work dealing with switch-box design can be found in [46–50].

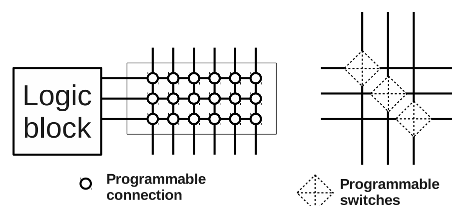


Figure 3.6: **Left side:** Connection Block. **Right side:** Switch Box.

Segmented architecture can be hybridized with nearest neighbor structure to increase flexibility and performance. Connection of CLBs using a segmented architecture requires the

signal to go through connection and switch boxes which may increase delay, thus, in certain scenarios having direct connections between logic blocks independent of tracks can be very useful. In the Figure 3.7 a hybrid interconnection architecture is shown. In Figure 3.7 one can see that the logic blocks are connected through tracks as in a segmented interconnection, in addition it has two other kind of connections in the Figure labeled as "direct-connect" and "long-interconnection". A direct-connect is a connection as in the nearest neighbor connection which connects two neighboring logic blocks. The long interconnection wires between switch boxes allow to by-pass intermediate switch boxes. An application of direct interconnection between logic blocks is efficient carry propagation. In a direct connection architecture the carry signal goes without passing through switch boxes or connections blocks, thus reducing the delay of the carry signal. If the implementation requires to connect logics blocks far apart, the long wires between switch boxes can be useful because they avoid some of the intermediate switch boxes reducing the delay of the signals.

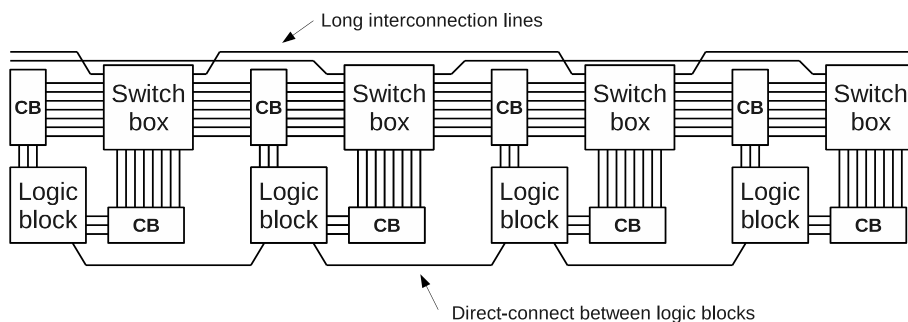


Figure 3.7: Segment structure hybridized with nearest neighbor structure.

Hierarchical

A proper grouping of logic blocks can improve the routing capabilities and reduce delays. Some times the logic blocks are clustered in a hierarchical fashion, which is called the hierarchical interconnection structure. The basic elements of this architecture are similar to the segmented architecture, and include logic blocks into which logic is mapped and connection blocks and switch blocks through which connection is realized. The difference is that the logic blocks and the connection forms a hierarchy. For example in Figure 3.8 a basic hierarchical structure is shown. In the architecture shown in Figure 3.8 the basic cluster consists of four logic blocks, we call this cluster of four logic block as a level 0 element. Four of these level 0 elements forms a level 1 element. Thus, a level 1 element is composed of sixteen logic blocks. Further, four level 1 elements are grouped to form a level 2 element which contains 64 logic blocks. Theoretically such an hierarchy can be organized in different ways, like a single logic block can be defined as a level 0 element and the whole FPGA can

be considered as a single element in level n, for some value of n. The connection resources such as connection blocks, switch boxes, long wires, horizontal and vertical tracks and direct connections between groping levels can be placed in any of the levels and their complexities can vary on different levels. Intuitively the lower level elements require lower flexibility as there are fewer connections to route. At each level routing is first performed locally. If a signal has a connection with another element beyond its level then it is routed out through its connection block. This architecture requires less switches to route and thus can implement much faster logic with much less interconnection resources as compared to the standard segmented model. This interconnection structure has been widely studied [39, 88] and has been adopted in commercial FPGAs.

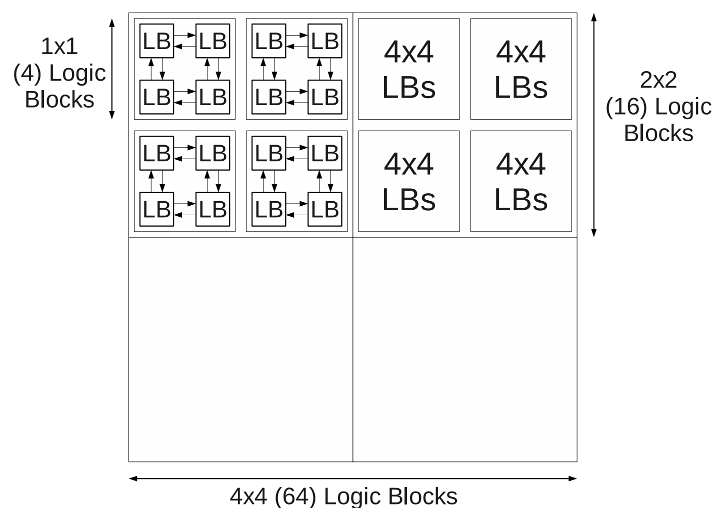


Figure 3.8: Hierarchical structure with basic cluster of four logic blocks.

In the three subsequent sections we shall describe some important characteristics of three specific families of FPGAs namely Spartan 3, Virtex 5 and Lattice ICE40.

3.3 Xilinx Spartan 3 FPGAs

The information presented in this Section is an extract of the *Spartan-3 FPGA Family Data Sheet* [145]. Spartan 3 FPGAs are low-cost devices, so they are a good option for developing fast prototypes of hardware designs. The main features of this family are listed below:

1. Advanced 90-nanometer process technology.
2. Up to 633 Input/Output pins.
3. Logic resources:

(a) 4-input LUTs.
(b) Abundant logic cells with shift register capability.
(c) Wide, fast multiplexers.
(d) Fast look-ahead carry logic.
(e) Dedicated 18 x 18 multipliers.

4. Up to 1,872 Kbits of total block RAM.

3.3.1 Configurable Logic Blocks

In Spartan 3 the Configurable Logic Blocks (CLBs) are the main resource to implement both combinatorial and synchronous circuits. Within each CLB there are four interconnected slices as shown in Figure 3.9. As depicted in Figure 3.9, the slices are grouped in pairs and are placed as columns with an independent carry chain whose input and output are marked as CIN and COUT respectively. In Figure 3.9, the labels X and Y in the boxes representing the slices indicates the row and column numbers of the of slices respectively within the matrix of CLBs. X counts up from left side of the matrix to the right, Y starts from bottom of the matrix. Figure 3.9 shows the CLB located in the lower left hand corner of the matrix. The slices labeled with an even X number are called SLICEM (left-hand of the figure) and the slices labeled with an odd X value are celled SLICEL (right-hand side of the figure). The slices located in the same column have very fast carry propagation between them. In the same Figure we also show the interconnection resources as an hybrid structure, i.e, there are tracks and connections between CLBs in the neighborhood.

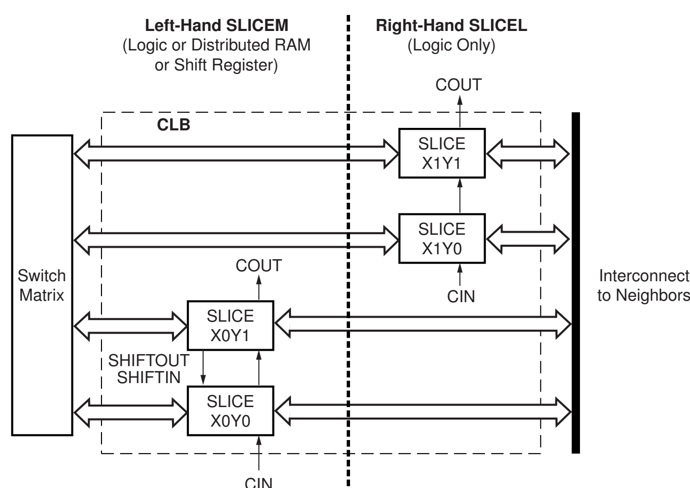


Figure 3.9: CLB of Spartan 3 FPGAs.

Resources within an Slice

Both SLICEL and SLICEM have the following elements in common: two logic function generators (LUTs), two storage elements, wide-function multiplexers, carry logic and arithmetic gates. In the Figure 3.10 a simplified diagram of SLICEM is shown. To convert the diagram in Figure 3.10 to a SLICEL, one just need to eliminate the components and lines in blue.

LUTs in SLICEM also have the functionalities of storing data using distributed RAM and shifting data with a 16-bit shift register. LUTs are RAM-based, in SLICEM they can be configured as a function generator, 16-bit distributed memory or 16-bit shift register. The storage element is a D-type Flip Flop which is programmable since one can choose registered output YQ or non-registered output Y. Width-functions multiplexers combine LUTs to allow the implementations of more complex Boolean functions, there is an F5MUX in each slice accompanied by an FiMUX which takes on the names F6MUX, F7MUX, F8MUX depending of the position of the slice in CLB. Extra logic gates and control multiplexer to handle the carry chain allow us to get fast and efficient implementations of math operations. For more information of how use the components mentioned above refer to [148].

3.3.2 Interconnection Resources

Inside the Spartan 3 devices there are four types of interconnections: Long lines, Hex lines, Double lines, and Direct lines. We will briefly described the interconnection resources below.

Switch Matrix

The main functional elements which are to be interconnected in a Spartan 3 FPGA are the configurable logic blocks (CLB), the input/output buffers (IOB), the digital clock manager (DCM), the block RAMs and the multipliers. The switch matrix (or the switch box) connects the different kinds of these elements across the device. A single switch matrix connected to a functional element such as CLB, IOB or DCM is defined as an *interconnect tile*. Elements like block RAMs and multipliers may be connected to more than one switch matrix. In such cases the interconnect tile consists of the functional element and all the switch matrices attached to it. In Figure 3.11 the types of interconnect tiles are illustrated. As depicted in the Figure the 18kB RAM along with the four switch boxes to which it is connected forms an interconnect tile, similarly for the multiplier. An array of interconnect tiles defines a device. The different interconnection resources like the tracks and the wires are located in between each interconnect tile. This structure is depicted in Figure 3.12 where the bold lines shows the channels of interconnection resources.

Spartan 3 contains the following interconnection resources: Horizontal and vertical long

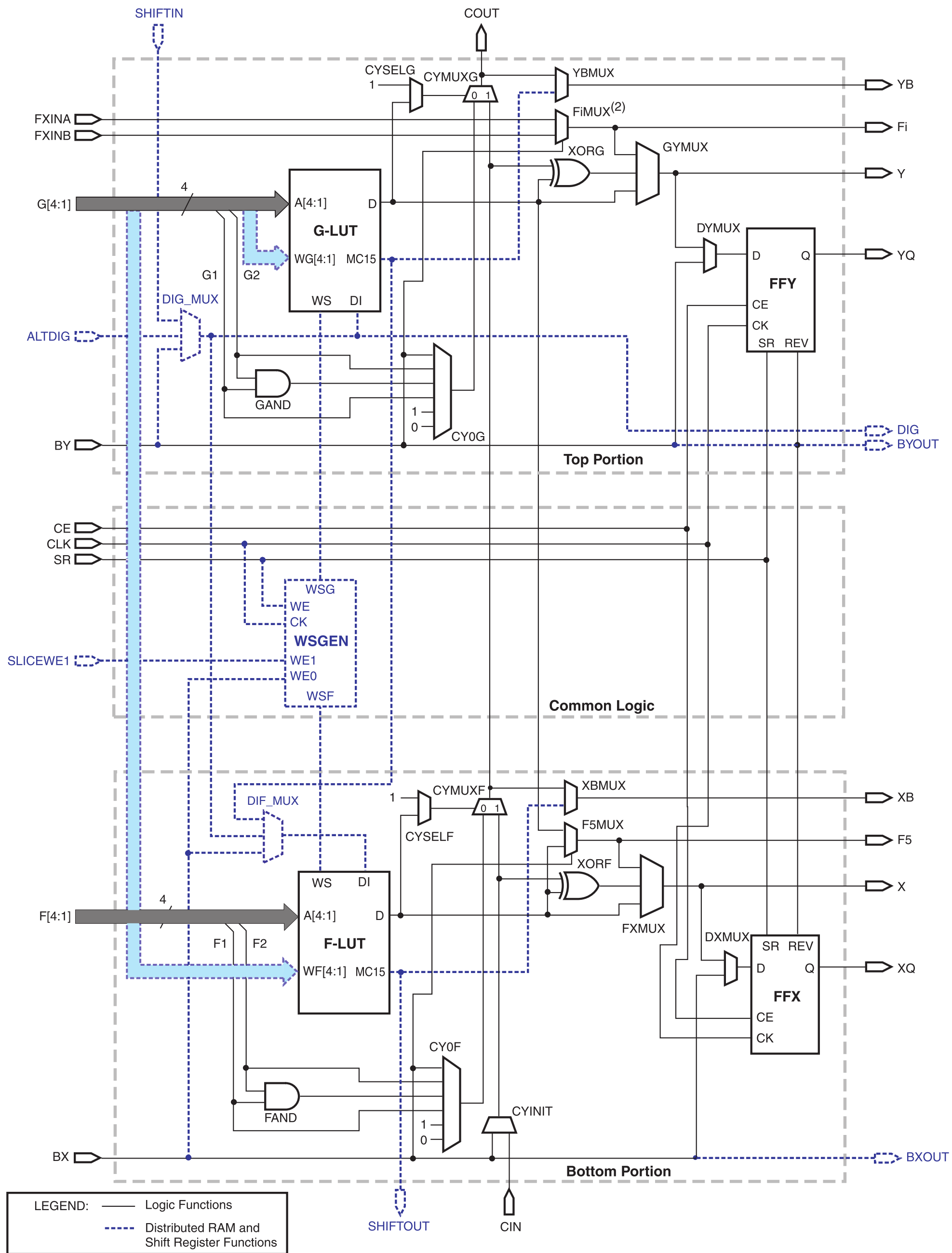


Figure 3.10: SLICEM of Spartan 3 FPGAs. Options to invert signal polarity as well as other options that enable lines for various functions are not shown. The index i can be 6, 7, or 8, depending on the slice. In this position, the upper right-hand slice has an F8MUX, and the upper left-hand slice has an F7MUX. The lower right-hand and left-hand slices both have an F6MUX.

lines, Hex lines, Double lines and Direct lines. We describe briefly these resources next.

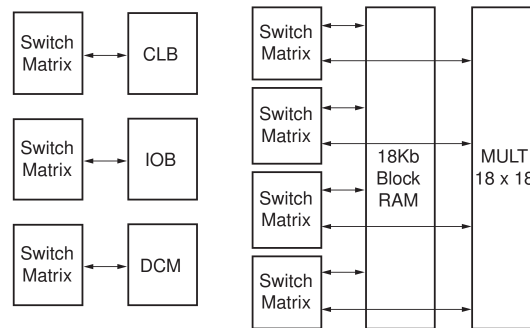


Figure 3.11: Interconnect tiles available in Spartan 3 FPGAs.

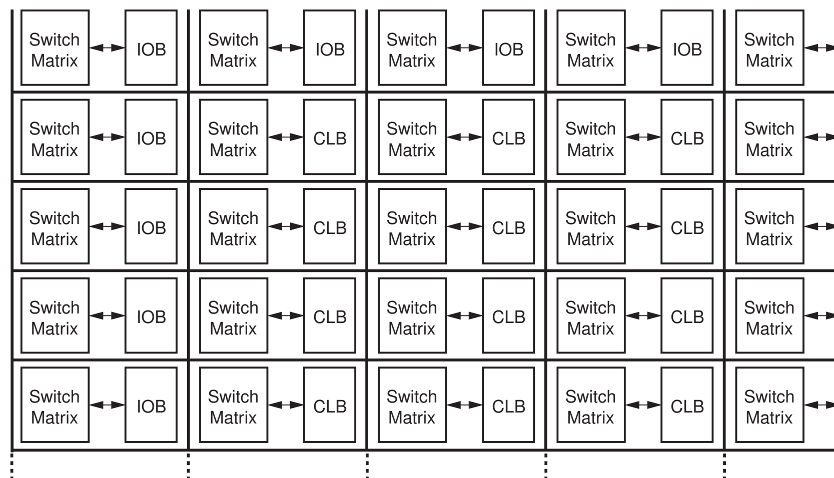


Figure 3.12: Array of Interconnect Tiles in an FPGA.

Horizontal and Vertical Long Lines: There are 24-wire long lines that span both vertically and horizontally and connects to one out of every six interconnect tiles (Figure 3.13). At any tile, four of the long lines send or receive signals from a switch matrix. These lines are well-suited for carrying high frequency signal with minimal loading effects due their low capacitance.

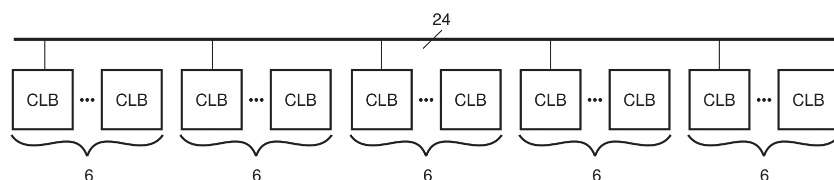


Figure 3.13: Horizontal and Vertical Long Lines.

Hex Lines: Each set of eight hex lines are connected to one out of every three tiles, both horizontally and vertically (Figure 3.14). Thirty-two hex lines are available between any

given interconnect tile. Hex lines are only driven from one end of the route.

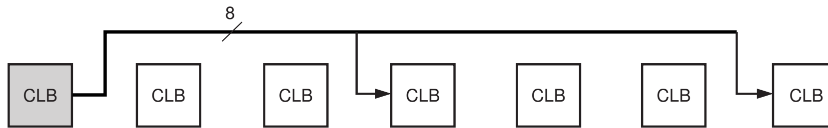


Figure 3.14: Hex Lines.

Double Lines: They connect an interconnect tile to its neighbors in all four directions both vertically and horizontally (Figure 3.15). There are thirty-two double lines available between any given interconnect tile. Double lines add more flexibility compared to long lines and hex lines.

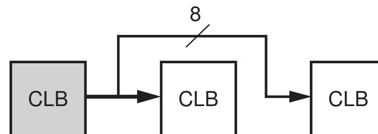


Figure 3.15: Double Lines.

Direct Lines Direct connect lines route signals to neighboring tiles vertically, horizontally, and diagonally (Figure 3.16). These lines most often drive a signal from a "source" tile to a double, hex, or long line and conversely from the longer interconnect back to a direct line accessing a "destination" tile.

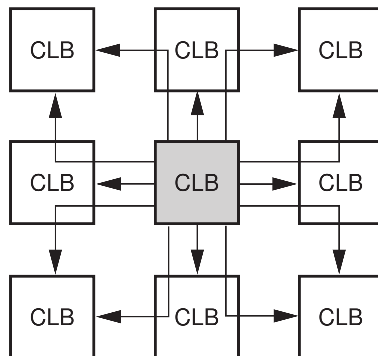


Figure 3.16: Direct Lines.

3.4 Xilinx Virtex 5 FPGAs

The information shown in this section was taken from *Virtex-5 FPGA User Guide* [150]. The main features of the Virtex 5 family of FPGAs are as follows:

- Built on a 65-nm state-of-the-art copper process technology.
- Logic resources:
  - Real 6-input look-up table (LUT) technology.
  - Dual 5-LUT option.
  - Improved reduced-hop routing.
  - 64-bit distributed RAM option.
  - SRL32/Dual SRL16 option.
- True dual-port RAM blocks.
- Advanced DSP48E slices:
  - 25 x 18, two's complement, multiplication.
  - Optional adder, subtracter, and accumulator.
  - Optional pipelining.
  - Optional bitwise logical functionality.
  - Dedicated cascade connection.
- PowerPC 440 Microprocessors (FXT Platform only).[1]

A glaring difference of Virtex 5 family compared to the Spartan 3 family is that the LUTs are 6 input, in addition these FPGAs are equipped with some special slices called DSP slices moreover a PowerPC microprocessor is embedded in the FPGA. In this Section we describe the CLBs and the interconnection resources of a Virtex 5 FPGA.

3.4.1 Configurable Logic Blocks

In Virtex 5 FPGAs each CLB contains two slices placed in columns, they are labeled as SLICE(0) for slice in the right side and SLICE(1) for slice in the left side, as is shown in Figure 3.17. Each slice has an independent carry chain. To locate slices in the slices matrix, all slices are labeled with an X followed by the column number and a Y followed by the row number of the slice (Figure 3.18).

[1]The Vitex 5 family of FPGAs is divided into five platforms: LX high-performance general logic applications, LXT high-performance logic with advanced serial connectivity, SXT high-performance signal processing applications with advanced serial connectivity, TXT high-performance systems with double density advanced serial connectivity and FXT high-performance embedded systems with advanced serial connectivity.

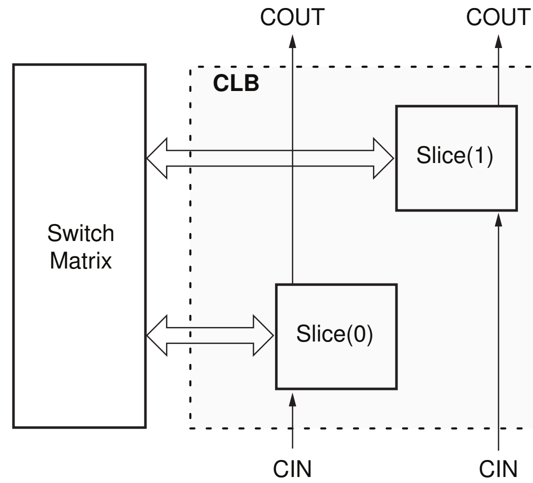


Figure 3.17: Slices within a CLB.

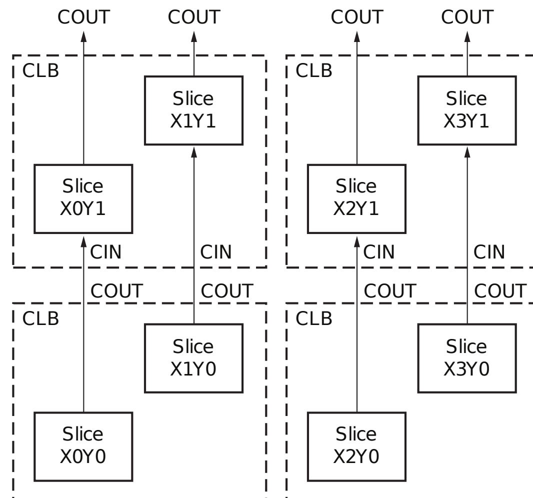


Figure 3.18: Slices position into the CLBs matrix.

Resources within an Slice

Every slice contains four 6-input LUTs, four storage elements, wide-function multiplexers and carry logic. Same as Spartan 3, LUTs of slices of type SLICEM support two additional functionalities: distributed RAM and shifting data with 32-bit registers. The function generators in Virtex 5 are constructed as is shown in Figure 3.19, they have six inputs, two outputs and internally two 5-input LUTs and one multiplexer. These resources allow them to compute any arbitrary 6-input Boolean function (combining the output of the two LUTs and using the select input of the multiplexer as the most significant bit of the input of the function) or two arbitrary defined 5-input Boolean functions (using 5-input LUTs independently), as long as these two functions share common inputs and each of these functions

have their own output line (labeled in Figure 3.19 as $D6$ and $D5$). Signals from the function generators can exit the slice (through A, B, C, D output for O6 or AMUX, BMUX, CMUX, DMUX output for O5).

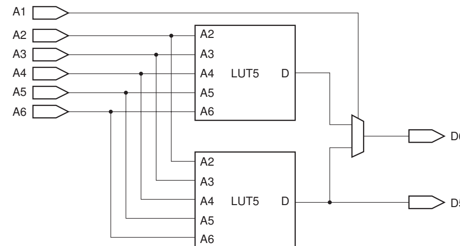


Figure 3.19: Internal view of LUT of Virtex 5 FPGAs.

In Figure 3.20 an SLICEM is shown, we can see several multiplexers and an additional Xor gates which allow to handle the carry chain. The multiplexer also can be used to construct large multiplexers and functions with more than six inputs.

LUTs in SLICEM can be configured as a 32-bit shift register and using the internal multiplexers, one can implement even a 128-bit shift register using only one SLICEM using the four available LUTs configured as a 32-bit shift register. Also using a single slice one can implement a 16×1 multiplexer. For more information see Chapter 5 of [150].

DSP slices or DSP48E deserve an special mention, they are slices specialized in Digital Signal Processing (DSP) and incorporate some tools such as an ALU specialized in DSP operations. A general diagram of a DSP48E is shown in Figure 3.21, a DSP48E is very useful to implement integer arithmetic insomuch as it contains a 25×18-bit signed multiplier (or 24×17-bit unsigned multiplier), a 48-bit-two's-complement adder and 48-bit logic unit with Bit-wise logic operations two-input AND, OR, NOT, NAND, NOR, XOR, and XNOR. Signal labeled with an $*$ in Figure 3.21 refer to signals connected to a high speed bus which interconnects DSP48E slices between them. More information about DSP48E slices can be found in *Virtex-5 FPGA XtremeDSP Design Considerations* [151].

Interconnection Resources

The information available today about interconnection structure of Virtex 5 FPGAs is very limited. They use a new diagonally symmetric interconnect pattern which enhances performance by reaching more places in fewer hops. That pattern allows for more logic connections

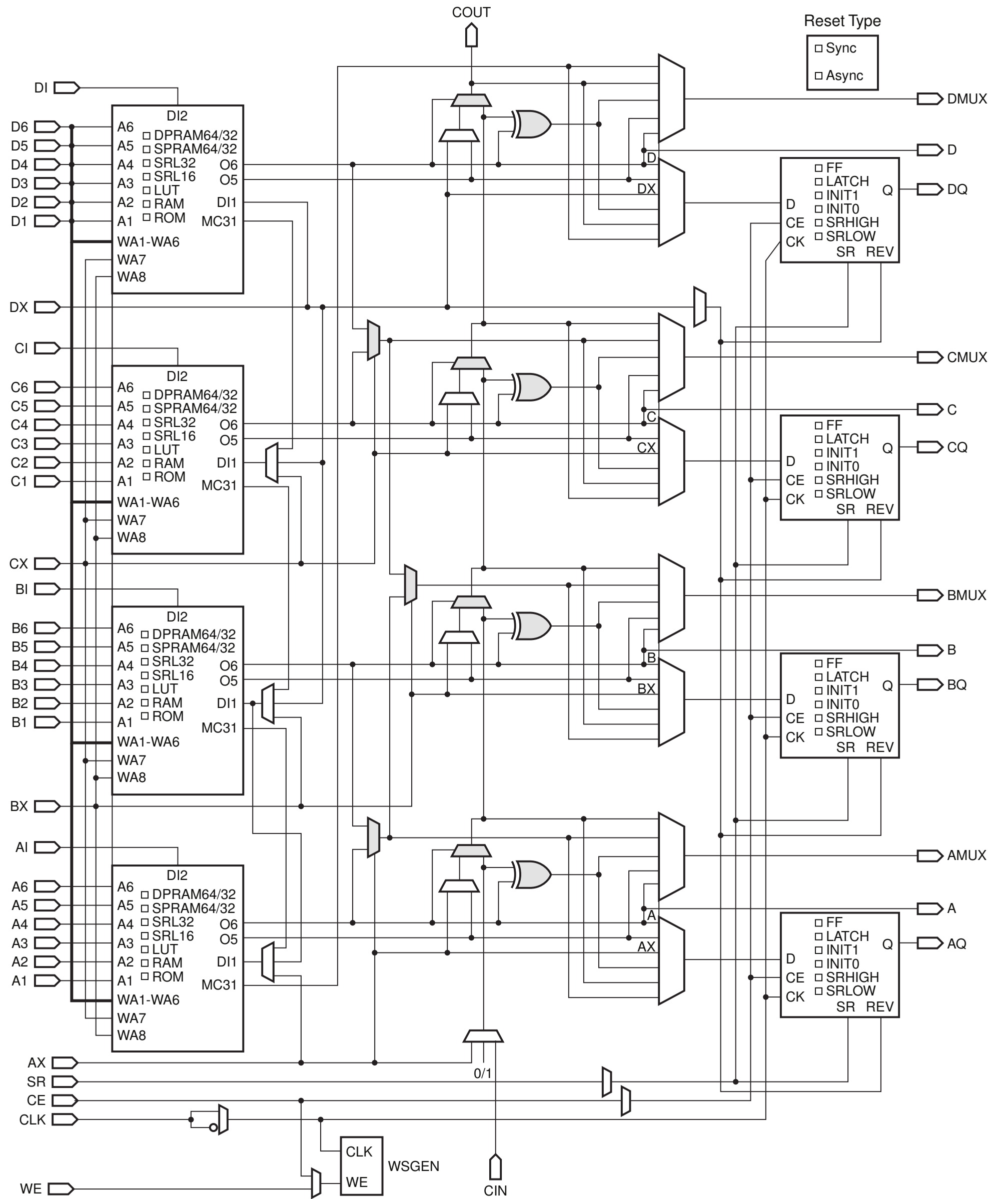


Figure 3.20: SLICEM of Virtex 5 FPGAs.

to be made within two or three hops, information taken from *Achieving Higher System Performance with the Virtex-5 Family of FPGAs* [142].

3.5 Lattice ICE40 FPGAs

The lattice ICE40 FPGAs were developed to deliver the lowest static and dynamic power consumption of any comparable CPLD or FPGA device, hence there are suitable for mobile

*These signals are dedicated routing paths internal to the DSP48E column. They are not accessible via fabric routing resources.

Figure 3.21: Slice DSP48E.

and battery powered applications. In this Section we will give a brief description of the architecture and characteristics of ICE40 FPGAs starting with logic resources followed by the routing resources available in them. The information about this family of FPGAs was taken from *iCE40 Family Handbook Ultra Low-Power mobileFPGATM LP, HX* [89]. Some general features of LATTICE ICE40 are:

- Proven, high-volume 40 nm, low-power CMOS technology.
- Up to 7,680 Logic Cells (LUT+Flip-Flop).
- Two-port 4Kbit RAM blocks (RAM4K).
- Up to 206 Programmable I/O.
- 4-input LUTs.

3.5.1 Programmable Logic Block

The basic logic resource in ICE40 FPGAs is the Logic Cell (LC), and a Programmable Logic Block contains 8 LCs interconnected between them through the carry logic resources as is shown in Figure 3.22. ICE40 devices contain up to 8000 LCs, each LC has three elements as can be seen in Figure 3.22, these elements are:

- A 4-input LUT (LUT4) capable to build any combinatorial logic function of up to four inputs, also it can behave as a 16×1 ROM. To construct wider functions several LUTs should be combined and cascaded.

- A D-type Flip-Flop with an optional clock enable and reset control input.
- Carry Logic, it accelerates the logic efficiency and performance of arithmetic functions and some wide cascaded logic functions.

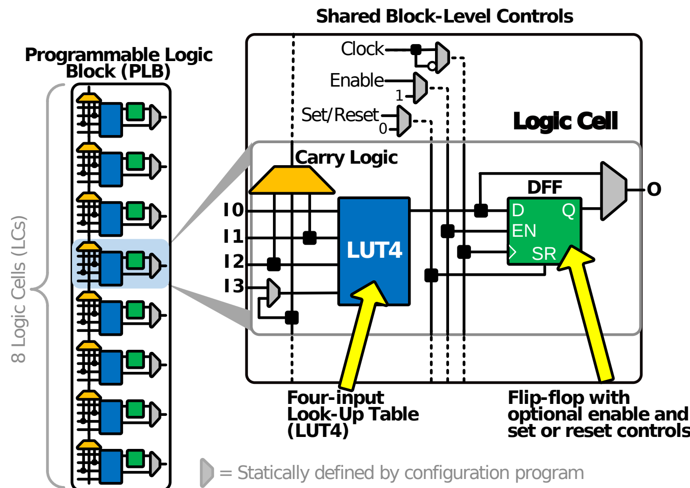


Figure 3.22: Programmable Logic Block and Logic Cell of Lattice ICE40 FPGAs.

Interconnection Resources

Same as Virtex 5 FPGAs the information about the connection resources for ICE40 is scarce. In the Figure 3.23 we show the placement of PLBs within the device, the PLBs are accommodated in a matrix. The carry chain can span across multiple PLBs, as PLBs in the same column are connected by the carry chain resources. PLBs can reach I/O banks and block RAMs through programmable interconnections.

3.6 Basics of FPGA Programming

For implementing a system in an FPGA the description of the digital system is first written using a special type of programming language called a hardware description language (HDL). This description can be processed by specialized tools to either map the design to a real hardware or to obtain characteristics of the design in various hardware platforms using simulations. In this section we briefly explain the various phases of design using FPGAs.

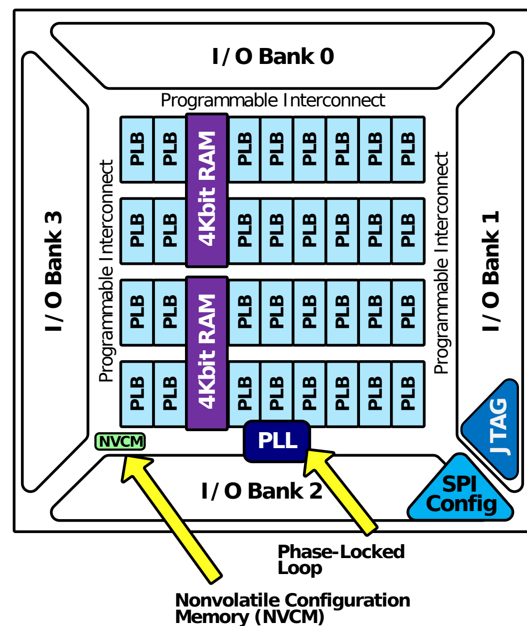


Figure 3.23: Internal view of a Lattice ICE40.

3.6.1 Hardware Description Languages

A Hardware Description Language (HDL) is a language which allow us to describe digital circuits, an HDL must satisfy the following requirements [109]:

1. **Support for Design Hierarchy.** A HDL must allow description of the hardware including the functionalities of its subcomponents. Finally the design must look as a hierarchy of components where each component of the hierarchy has its own subcomponents until the lowest level. In lowest level the components are described using their functionality. Designing using the hierarchy of components allow the reutilization of components and the use of libraries.

2. **Library Support.** a HDL must have a mechanism for accessing various libraries. The libraries can contain an interface description of a design, and when several descriptions and models are correct and complete, they should be placed in to the library, and the libraries should be accessible to different designers.

3. **Sequential Statements.** An important requirement of a HDL is the support for concurrency operations but sometimes to describe the functionality of certain components we need sofware-like sequential statements. Decisional statements like *if then else, case, when* and loop statements as *for do, while* should be present in the specific syntax of a HDL, the execution of these statements is procedural.

4. **Generic Design.** The operation of the hardware components is also influenced by the physical features of the target where the design is implemented, but it is desirable that the description of the component using a HDL should be generic, i.e., it should be functional, for example, for many families of different manufacturers of FPGAs.

5. **Type Declaration and Usage.** The language must provide various data types. In addition to the must used bit-wise and boolean types,a HDL should allow declaration of other data types like floating point, integer, enumerate and also some data structures vectors. If the HDL does not directly support these kind of types then the users should be able to define them and put them into a library, so that other designers can used them. The operations also can be redefined for the new data types (for example addition, subtraction, AND,OR,ETC).

6. **Use of Subprograms.** The possibility to handle functions and procedures is another requirement for a HDL, it should allow to define functions and procedures, explicit type conversions, logic unit definitions, operator redefinitions, new operation definitions, etc.

7. **Timing Control.** The handling of the time at all levels of the design is a very important requirement of a HDL. In synchronous digital system the clock is the heart of the system. So the HDL should provide a way to fix the behavior of some signal to the clock, for example assign some value when the clock transition from low level to high level takes place.

In contrast to traditional programming languages the HDLs define specific instructions for real parallelism as is intrinsic in hardware. A HDL allows us to define some things that are not allowed in a programming language, such as the instantiation of multiple components of the same type or specify manually the interconnections between components. Other important feature of HDL is that the assignments can occur in parallel, and the propagation of the signal takes time (propagation delay). When we are designing hardware every data structure should be static because we are defining the hardware itself while in software we are using the hardware.

Some existing HDLs are listed below:

- **AHPL.** A hardware Programming Language.
- **CDL.** Computer Design Language.
- **CONLAN.** CONsensus LANguage.
- **IDL.** Interactive Design Language.
- **ISPS.** Instruction Set Processor Specification.

- **TEGAS.** Test Generation and Simulation.
- **TI-HDL.** Texas Instruments Hardware Description Language.
- **ZEUS.**
- **ABEL.** Advance Boolean Expression Language.
- **Verilog.**
- **VHDL.** Very High Speed Integrated Circuits Hardware Description Language.

For the hardware implementations in this thesis we use VHDL because it is an IEEE standard for HDLs and the manufacturers of FPGAs offer good support for it, which include tools and VHDL libraries to their devices. For example the Xilinx ISE and Lattice iCEcube2 design software.

3.6.2 Design Flow

The basic steps to implement circuits in FPGAs is shown in Figure 3.24, a more detailed explanation about design flow can be found in *Xilinx Synthesis and Simulation Design Guide* [144]. A brief description of each step is given below:

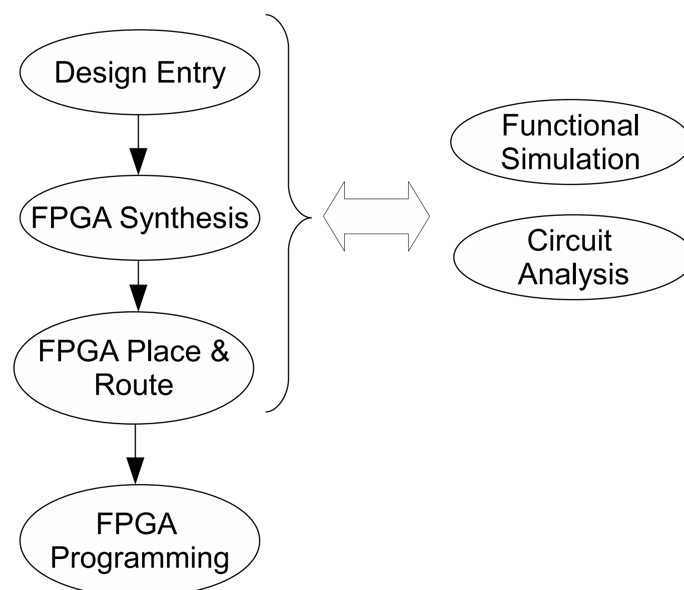


Figure 3.24: Design Flow.

1. **Design Entry:** A design must be described using an HDL or some other schematic tool. The description preferably must be modular and hierarchical. Generally, Register Transfer Level (RTL) is the abstraction level used to describe circuits with a HDL. RTL refers to how transfer of data happens between registers, logic units and buses. Nowadays sometimes an Electronic System Level (ESL) is used as the abstraction level [57]. In ESL a designer is only concerned with the functionality of the design and describes the design that is going to be implemented. The algorithm is described using a procedural language like C. In this abstraction level is not necessary that the designer handles the clock, timing and other low level hardware details.

2. **Functional Simulation:** The most basic simulation of a design is called the behavioural. It uses the VHDL code and verify its behaviour. It is recommendable to do a simulation for each component present in the hierarchy of the design.To simulate the components a test bench is done using VHDL. A test bench is a file with the description of wave forms to feed the component under testing. There are other kinds of simulations, post synthesis and post place & route, they are more advanced than behavioural simulations since these simulations include gate and wires delays. So the kind of simulation performed depend of which steps of the Figure 3.24 have been done.

3. **Synthesis:** Synthesis is a process which converts the design entry specifications to electronic components (multiplexers, counters, registers, etc), and finally maps these components to blocks of the FPGA device. A netlist of basic gates is prepared from HDL/schematic design entry, which is further optimized at gate level.

4. **Place and Route:** This phase decides the placement of logic cells on the target FPGA. This also involves wiring input and output of these logic cells through wiring and switch boxes available within FPGA. This phase needs the design description and the information about the architecture of FPGA. This process uses heuristics algorithms to find the best routing for a given design, the use of timing or area constraints is an important in this phase issue to get a good performance. In this phase it is also important to define the I/O pin constraints, i.e, to assign the inputs and outputs defined in the design into the real pins available in FPGA.

5. **Circuit Analysis:** It verifies the design using different performance metrics such as throughput or throughput per area. Also a timing simulation can be performed, this kind of simulation corroborates the correctness of the circuit taking into account the intrinsic features of the device such as the delays of the component of a slice. There are also some tools which allows estimation of the power consumption of the circuit.

3.7 Summary

In this Chapter we provided a brief introduction to the paradigm of the configurable computing. Reconfigurable hardware, in particular FPGAs are currently widely used for building prototypes of cryptographic systems and also in some cases FPGAs can be widely deployed in commercial devices. In this Chapter we provided discussions about the architecture of Virtex 5, Spartan 3 and Lattice ICE40 FPGAs. We have extensively used these families to build prototypes of disk encryption schemes.

Chapter 4

Tweakable Enciphering Schemes

The philosophers have only interpreted the world, in various ways. The point, however, is to change it.

Karl Marx

Tweakable enciphering schemes are a block cipher mode of operation which provides adequate security and functionality for its use in the application of encrypting block oriented storage media. In this Chapter we present various aspects of tweakable enciphering schemes. In Section 4.1 we introduce tweakable block ciphers which serves as basic motivation for tweakable encryption. In Section 4.2 we formally define the syntax and security of tweakable enciphering schemes. In Section 4.4 we argue about the suitability of tweakable enciphering schemes for the application of disk encryption. In Section 4.5 we describe the constructions of some of the existing TES and finally in Section 4.6 we discuss some recent standardization activities surrounding the application of disk encryption.

4.1 Tweakable Block Ciphers

Block ciphers as defined in Section 2.3 are deterministic algorithms which encrypt fixed length strings. When block ciphers are put to use for real encryption by a mode of operation, then the determinism inherent in a block cipher is broken using an initialization vector or a nonce which helps in providing some kind of state information or randomness. Informally, among other objectives, the use of IVs (or nonces) in modes serves to provide a functionality that the same plain text when encrypted multiple times (using different IVs) yields different ciphertexts. This variability obtained in ciphertexts serves better in achieving security. Tweakable block ciphers are an attempt to achieve a similar functionality for stand alone block ciphers. In a tweakable block cipher (TBC) the variability of the cipher produced is gained by extending the usual syntax of a block cipher and allowing it to take an extra input (other than the key and the message) called a *tweak*. Thus when the same plain text is

encrypted with a TBC with the same key and message but different tweaks then the outputs are different. The notion of TBC was first introduced in [93], the extended version of this work was also recently published in [94].

Formally a TBC is a map

$$\widetilde{E} : \mathcal{K} \times \mathcal{T} \times \{0,1\}^n \to \{0,1\}^n,$$

where $\mathcal{K}$ is the key space and $\mathcal{T}$ is the tweak space, for each $K \in \mathcal{K}$ and $T \in \mathcal{T}$, $\widetilde{E}_K^T(\cdot)$ is a permutation of $\{0,1\}^n$. An n-bit message M is encrypted by a TBC under a secret key K and a non-secret tweak T to obtain an n-bit ciphertext $C = \widetilde{E}_K^T(M)$. TBCs are constructed using block ciphers and the tweak is used in some way such that the property of PRP or SPRP are preserved.

There are several constructions available for TBCs. Let $E_K : \{0,1\}^n \to \{0,1\}^n$ be a secure block cipher and $T \in \{0,1\}^n$, then the four functions listed below are all secure TBCs

(a) $\widetilde{E}_K^T(M) = E_K(T \oplus E_K(M))$

(b) LRW construction: $\widetilde{E}_{K_1,K_2}^T(M) = E_K(M \oplus h_{K_2}(T)) \oplus h_{K_2}(T)$

(c) XE construction: $\widetilde{E}_K^T(M) = E_K(M \oplus E_K(T))$

(d) XEX construction: $\widetilde{E}_K^T(M) = E_K(M \oplus E_K(T)) \oplus E_K(T)$

The constructions (a) and (b) were proposed in [93], whereas the constructions (b) and (c) were proposed in [118]. In (b) in addition to the block cipher $E_K()$ a function h is used, which is required to be $\epsilon-$AXU. The definition of an $\epsilon-$AXU is similar to $\epsilon-$AU as provided in Definition 2.1.

Definition 4.1. *Let $h : \mathcal{K} \times \{0,1\}^\ell \to \{0,1\}^n$ be a family of functions. h is called an $\epsilon-$almost xor universal ($\epsilon-$AXU) family if for all $w, w' \in \{0,1\}^\ell$ where $w \neq w'$ and all $z \in \{0,1\}^n$,*

$$\Pr[K \xleftarrow{\$} \mathcal{K} : h_K(w) \oplus h_K(w') = z] \leq \epsilon.$$

If $E_K()$ is secure in the sense of a PRP then, constructions (a) and (c) are tweakable PRPs and if $E_K()$ is secure in the sense of a SPRP then constructions (b) and (d) are secure as tweakable SPRPs. Generalizations of some of these constructions were provided in [30, 104].

One of the motivations to introduce TBC was to construct secure modes of operation starting from them. This has been achieved to a great extent as some modes of operations like OCB, PMAC etc. have been constructed using TBC as primitives [118]. In many cases use of a TBC has helped to make both the description and analysis of modes much simpler.

4.2 Tweakable Enciphering Schemes: Definitions and Security Notions

TES are motivated by TBCs, TES can be seen as TBCs with a domain which contain arbitrary long (possibly of variable lengths) strings. Formally, a *Tweakable Enciphering Scheme (TES)* is a function $\mathbf{E} : \mathcal{K} \times \mathcal{T} \times \mathcal{M} \rightarrow \mathcal{M}$, where $\mathcal{K} \neq \emptyset$ and $\mathcal{T} \neq \emptyset$ are the key space and the tweak space respectively. The message and the cipher spaces are $\mathcal{M}$. In general we assume that $\mathcal{M} = \cup_{i>0}\{0,1\}^i$, but in certain scenarios $\mathcal{M}$ may be restricted to contain strings of some predefined lengths.

We shall sometimes write $\mathbf{E}_K^T(.)$ instead of $\mathbf{E}(K,T,.)$. The inverse of an enciphering scheme is $\mathbf{D} = \mathbf{E}^{-1}$ where $X = \mathbf{D}_K^T(Y)$ if and only if $\mathbf{E}_K^T(X) = Y$. An important property of a tweakable enciphering scheme is that it is length preserving, i.e., for every $x \in \mathcal{M}$ and every $T \in \mathcal{T}$, $|\mathbf{E}_K^T(x)| = |x|$. Same as in TBCs, the tweak, used in a TES is not the same as a nonce, as the tweak can be repeated, it is only meant to provide variability in the ciphertext.

Security of TES: Let $\text{Perm}^{\mathcal{T}}(\mathcal{M})$ denote the set of all functions $\boldsymbol{\pi} : \mathcal{T} \times \mathcal{M} \rightarrow \mathcal{M}$ where $\boldsymbol{\pi}(\mathcal{T},.)$ is a length preserving permutation. Such a $\boldsymbol{\pi} \in \text{Perm}^{\mathcal{T}}(\mathcal{M})$ is called a tweak indexed permutation. For a tweakable enciphering scheme $\mathbf{E} : \mathcal{K} \times \mathcal{T} \times \mathcal{M} \rightarrow \mathcal{M}$, we define the advantage an adversary $\mathcal{A}$ has in distinguishing $\mathbf{E}$ and its inverse from a random tweak indexed permutation and its inverse in the following manner.

$$\begin{aligned}\mathbf{Adv}_{\mathbf{E}}^{\pm\widetilde{\text{prp}}}(\mathcal{A}) = \Big|&\Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\mathbf{E}_K(.,.),\mathbf{E}_K^{-1}(.,.)} \Rightarrow 1\right] - \\ &\Pr\left[\boldsymbol{\pi} \xleftarrow{\$} \text{Perm}^{\mathcal{T}}(\mathcal{M}) : \mathcal{A}^{\boldsymbol{\pi}(.,.),\boldsymbol{\pi}^{-1}(.,.)} \Rightarrow 1\right]\Big| \end{aligned} \tag{4.1}$$

We assume that an adversary never repeats a query, i.e., it does not ask the encryption oracle with a particular value of (T,P) more than once and neither does it ask the decryption oracle with a particular value of (T,C) more than once. Furthermore, an adversary never queries its deciphering oracle with (T,C) if it got C in response to an encipher query (T,P) for some P. Similarly, the adversary never queries its enciphering oracle with (T,P) if it got P as a response to a decipher query of (T,C) for some C. These queries are called *pointless* as the adversary knows what it would get as responses for such queries.

TES are generally constructed using a block cipher as the basic primitive. Thus, the basic construction goal is to extend the domain of the block cipher to a domain which can contain arbitrary long strings of variable lengths, also one needs to incorporate the tweak. There have also been attempts to design TES from scratch, i.e., to design a tweakable block cipher with arbitrary block lengths. In this thesis we will concentrate only on the block-cipher based constructions, as they have been widely studied and also are known to be secure.

A design of a TES is done keeping in mind the considerations of efficiency usability and

security. We elaborate on these considerations next:

1. **Efficiency :** As said before, most known TES constructions are block cipher based. The block cipher based constructions also sometimes use certain kinds of hash functions which are realized using finite field multiplications. Thus the basic efficiency is measured by the number of block cipher calls and finite field multiplications required in the construction. The ease of parallel implementations is also an important consideration. A scheme which is amenable to parallelization can have a pipelined implementation in hardware (or can use instruction pipelining in software) thus giving rise to more efficient schemes.

2. **Usability :** Various kinds of usability requirements are considered for designing TES. An important consideration is the input message lengths. It turns out that a TES which is designed for only fixed length messages is less complex (and hence more efficient) than schemes which can handle variable lengths. Hence it is useful to have specifications only meant for fixed length messages in application scenarios where this restriction can be tolerated. Thus, some designers who have proposed schemes for arbitrary long messages also have proposed some modifications which can be used only for fixed length messages. Another usability consideration is the number of key bits necessary. Keys in any application needs to be stored securely, thus it is assumed that the more key bits a scheme requires it becomes more costly to use it, as one needs to bear the cost of storing the key securely. Hence, one of the important goals in the designs is to use minimal amount of key material.

3. **Security:** A block cipher based TES design is generally associated with a security proof. The security proof proves an upper bound on the $\pm\widetilde{\text{prp}}$ advantage (see Eq. 4.1) of an arbitrary adversary. Generally two different bounds are proved, namely the *information theoretic bound* and the *complexity theoretic bound*. While proving the information theoretic bound, the block cipher used in the construction is thought to be secure in the sense of a random permutation/function.The information theoretic bound generally depends on the query complexity[1] (σ) of the adversary and is independent of its running time. For all TES, generally an information theoretic bound of the form $O(\sigma^2)/2^n$ is known, such a bound is commonly called a quadratic security bound. An important goal is to design schemes where the security bound is small. The complexity theoretic bound considers the block cipher as a pseudorandom permutation/function and hence this bound is more than the information theoretic bound. Once the information theoretic bound is known, the corresponding complexity theoretic bound can be easily derived from it using some standard techniques [65].

[1]The query complexity (σ) of an adversary is defined as the number of n blocks of queries made by the adversary to its oracles, where n is the block length of the underlying block cipher used to construct the TES.

4.3 A Brief History of the Known Constructions of Tweakable Enciphering Schemes

In this section we provide a brief history of the TES constructions known till date. Later in Section 4.5 we provide details of some existing constructions.

The first work to present a scheme which is very near to a TES was by Naor and Reingold [107]. This work provides a construction of a strong pseudorandom permutation (SPRP) on messages of arbitrary length. The construction consists of a single encryption layer in between of two invertible universal hash functions. They did not provide a tweakable SPRP (which is the requirement for a TES) since their work predates the notion of tweaks which was introduced much later in [93].

The first construction of a tweakable SPRP was presented in [65] which was called the CMC mode of operation. Also, in [65] it was first pointed out that tweakable SPRP is a suitable model for low level disk encryption and thus TES should be used for this application. As mentioned, in this application, the disk is encrypted sector-wise and the sector address corresponds to the tweak (we elaborate more on this in Section 4.4). CMC consists of two layers of CBC encryption along with a light weight masking layer. The sequential nature of CBC encryption makes CMC less interesting from the perspective of efficient implementations.

Using the same philosophy of CMC a parallel mode called EME was proposed in [66]. In EME the CBC modes of CMC are replaced by electronic code book (ECB) layers which are amenable to efficient parallelization. EME has an interesting limitation that it cannot securely encrypt messages whose number of blocks are more than the block length of the underlying block-cipher, for example, if AES is used as the block cipher in EME then it can securely encrypt messages containing less than 128 blocks. This limitation was fixed in the mode EME* [66].

The modes CMC, EME, EME* have been later classified as *encrypt-mask-encrypt* type modes. As they use two encryption layers along with a masking layer. For encrypting m blocks of messages, these modes require around $2m$ block cipher calls, and the block cipher calls are the most expensive operation used by these modes.

A different class of constructions which have been named as the *hash-counter-hash* type, consist of two universal hash functions with a counter mode of encryption in between. The first known construction of this kind is XCB [101, 102]. In [101] the security of the construction was not proved, which was done later in [102].

HCTR [141] is another construction of this category which uses the same methodology of XCB. A serious drawback of HCTR was that the security proof provided in [141] only guaranteed that the security degrades by a cubic bound on the query complexity of the adversary. As quadratic security bounds for TES were already known, so HCTR seemed

to provide very weak security guaranties compared to the then known constructions. In an attempt to fix this situation HCH [31] was proposed, which modified HCTR in various ways to produce a new mode which used one more block cipher call than HCTR but provided a quadratic security guarantee. HCH offered some more advantages over HCTR in terms of the number of keys used, etcetera. In [105] another variant of HCTR was proposed which provides a quadratic security bound and later in [28] a quadratic security bound of the original HCTR construction (as proposed in [141]) was proved.

ABL [100] is another construction of the *hash-counter-hash* type, but it is inefficient compared to the other members of this category. The constructions of this type require both finite field multiplications along with block cipher calls. The efficient members of this category use about m block cipher calls along with $2m$ finite field multiplications for encrypting a m block message.

The paradigm proposed for the original Naor and Reingold construction [107] has also been further used to construct TES and they are categorized as *hash-encrypt-hash* type. Examples of constructions of this category are PEP [29], TET [64], HEH [122]. Like the *hash-counter-hash* constructions these modes also require about m block cipher calls and $2m$ finite field multiplications for encrypting a m block message.

In [128] significant refinements of constructions of *hash-counter-hash* and *hash-encrypt-hash* constructions were provided. The main idea in [125] is using a new class of universal hash functions called the Bernstein-Rabin-Winnograd (BRW) polynomials. The BRW polynomials provide a significant computational advantage, as they can hash a m block message using about $m/2$ multiplications in contrast to usual polynomials which require m multiplications. These new modes proposed in [125] are called HEH[BRW] and HMCH[BRW]. These modes can also be instantiated using usual polynomials, and such instantiations result in the modes HEH[Poly] and HMCH[Poly]. The TES using BRW polynomials are discussed in Chapter 6.

4.4 Tweakable Enciphering Schemes and Disk Encryption

The problem of low level disk encryption, as introduced in Chapter 1 consists of encrypting individual sectors/blocks in a block oriented storage media. As the sector length is fixed (generally 512 or 4096 bytes), thus, a length preserving deterministic encryption scheme should be used. [2]

TESs has all these required properties, also as TES are tweakable it can provide ample cipher text variability which makes them more suitable for the application. The sector address is generally considered to be the tweak, hence when two different sectors are encrypted with

[2]Though the restriction of length preserving encryption scheme has been stressed in the literature and standards till date, in Chapter 10 of this thesis we argue that this requirement may not be that necessary as is thought of.

two different tweaks, this makes the cipher texts in two different sectors different even if they store the encryption of the same information. This property can thwart many kinds of attacks.

TESs provide security in the sense of a strong pseudorandom permutation on the whole sector. This is quite a strong notion of security for deterministic ciphers. The security definition guarantees that the cipher texts produced by a TES are indistinguishable from random strings, moreover the security definition suggests that if a single bit in the ciphertext is changed then on decryption a string indistinguishable from a random string is produced. The second property serves as a form of authentication, as there is no way for an adversary to change a ciphertext stored in a sector so that it would get decrypted to something meaningful. It is to be noted that in a length preserving encryption scheme it is not possible to provide the service of data authentication as is provided by authenticated encryption (AE) schemes. As discussed in Section 2.5.2, an AE scheme outputs a tag which serves as a footprint of the message encrypted, and on decryption this tag can be checked to see if any adversarial tampering of the ciphertext has taken place. This though is the most preferred form of authentication, but as is obvious the tag adds to the length of the ciphertext and always makes the ciphertext longer than the plaintext. Thus, authentication as provided by AE schemes cannot be provided by any length preserving encryption scheme. So though TES do not provide true authentication an in AEs but as stated above, ciphertext tampering in TES can be detected by the high level applications which uses the data.

4.5 Description of some TES

In this section we provide detailed a description of some of the existing TES. Most of the algorithms in their original proposals are stated in the highest possible generality, i.e., the designers of some of the proposals have tried to incorporate messages and tweaks of arbitrary lengths. However this generality is not required in practice for the disk encryption problem, as the message in this case is always of fixed length (512 bytes or 4096 bytes), which is again a multiple of the most used block length (128 bits) of a block cipher. The tweak, which is the sector address, is also of fixed length and it can be restricted to one block. Hence, the description of the algorithms provided in this Section assume above length restrictions for both the plaintexts and the tweaks.

4.5.1 Hash-Counter-Hash

We begin with the description of the Hash-Counter-Hash type algorithms. We describe in details three schemes, namely, HCTR [141], HCH [31], XCB [101].

This class of TES constructions utilize a variant of the Wegman-Carter polynomial hash [24]

combined with a counter mode of operation. The scheme described in Fig. 4.1 is the encryption procedure using HCTR. Lines 1 and 5 of the HCTR algorithm compute the $H_h(.)$ hash function. For a plaintext string $P = P_1||P_2||\cdots||P_m$, where $P_i \in \{0,1\}^n$, $H_h(P)$ is defined as

$$H_h(P) = P_1 h^{m+1} \oplus P_2 h^m \oplus \ldots \oplus P_m h^2 \oplus \mathsf{bin}_n(|P|)h, \tag{4.2}$$

where h is an n-bit hash key. In addition to the two hash layers, HCTR performs a counter mode of operation in line 4. Given an n-bit string S, a key K and m blocks of plaintext/ciphertext $A_1, A_2, \ldots, A_m$ ($A_i \in \{0,1\}^n$), the counter mode is defined as,

$$\begin{aligned} \mathsf{Ctr}_{K,S}(A_1, \ldots, A_m) \;&=\; (A_1 \oplus E_K(S \oplus \mathsf{bin}_n(1)), \ldots, A_m \oplus \\ &\quad\; E_K(S \oplus \mathsf{bin}_n(m))). \end{aligned} \tag{4.3}$$

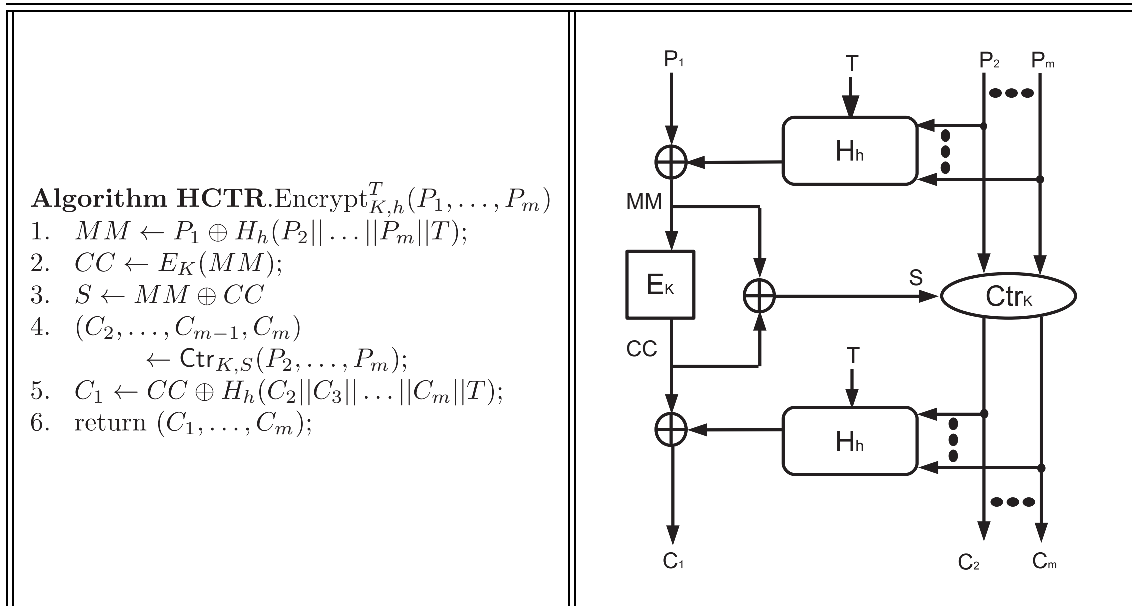


Figure 4.1: HCTR, encryption algorithm and block diagram.

The HCH algorithm is described in Figure 4.2. As we can see in Figure 4.2 the HCH algorithm is very similar to that of HCTR. The main difference is the way the tweak is handled and an extra encryption before the counter mode. The structural difference can be easily observed by comparing Figure 4.2 and Figure 4.1. HCH also uses a polynomial hash similar to that of HCTR. For $P = P_1||P_2||\cdots||P_m$, where $P_i \in \{0,1\}^n$ the hash function $H_{R,Q}(\cdot)$ is defined as

$$H_{R,Q}(P) = Q \oplus P_1 \oplus P_2 R^{m-1} \oplus P_3 R^{m-2} \oplus \cdots \oplus P_{m-1} R^2 \oplus P_m R. \tag{4.4}$$

The counter mode of HCH is same as that described in Eq. (4.3). HCHfp is an improvement over HCH which works only for messages whose lengths are multiple of n (the block size

of the block cipher). Compared to HCH, HCHfp saves one block cipher call but require an extra key for the polynomial hash as in HCTR. To obtain HCHfp from the algorithm of HCH presented in Figure 4.2 we just eliminate the computation of R and it is obtained as an input. Moreover, as all the blocks in the messages are complete (all the blocks in the message have n bits) padding is not necessary for the last block. The computation of Q is changed as $Q \leftarrow E_K(T)$. Following these changes HCHfp is obtained. The net saving that HCHfp gets over HCH is the block cipher call to produce R, but this saving is obtained at the cost of the key size, as in HCHfp R is obtained as an input key.

The XCB scheme is illustrated in Figure 4.3. XCB uses the hash $\boldsymbol{h}_h(P,T)$, where P is as defined above and T is an n bit string. The hash $\boldsymbol{h}_h()$ is defined as

$$\boldsymbol{h}_h(P,T) = P_1 h^{m+2} \oplus P_2 h^{m+1} \oplus \cdots P_m h^3 \oplus T h^2 \oplus (\mathsf{bin}_{\frac{n}{2}}(|P|)||\mathsf{bin}_{\frac{n}{2}}(|T|))h, \tag{4.5}$$

The counter mode in XCB is a bit different from the one in Eq. (4.3). Let us define a function $\mathsf{incr}_i(S)$, where i is a positive integer and S an n-bit string. Let $S = S_l||S_r$, where S_r is 32 bits long and S_l is $(n-32)$ bits long. Then we define

$$\mathsf{incr}_i(S) = S_l||[(S_r + i) \pmod{2^{32}}].$$

The counter mode of line 6 of the XCB algorithm is defined as

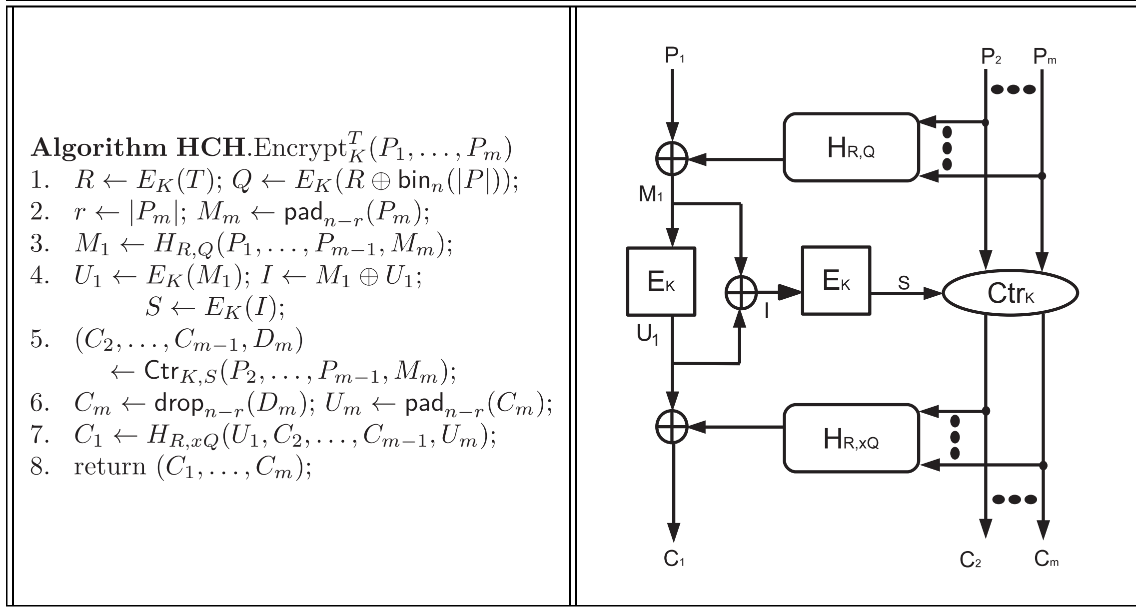
$$\bar{\mathsf{Ctr}}_{K,S}(A_1,\ldots,A_m) = (A_1 \oplus E_K(\mathsf{incr}_1(S)),\ldots,A_m \oplus E_K(\mathsf{incr}_m(S))) \tag{4.6}$$

4.5.2 Encrypt-Mask-Encrypt

The candidate for the Encrypt-Mask-Encrypt category that we describe here is called EME which stands for ECB-Mask-ECB. As the name suggests, the EME algorithm consists of two ECB layers with a layer of masking in-between. The description of the EME algorithm is provided in Figure 4.4. This TES uses the variants of ECB described in the algorithm in Fig. 4.4 as First-ECB and Last-ECB. These functions are defined as follows:

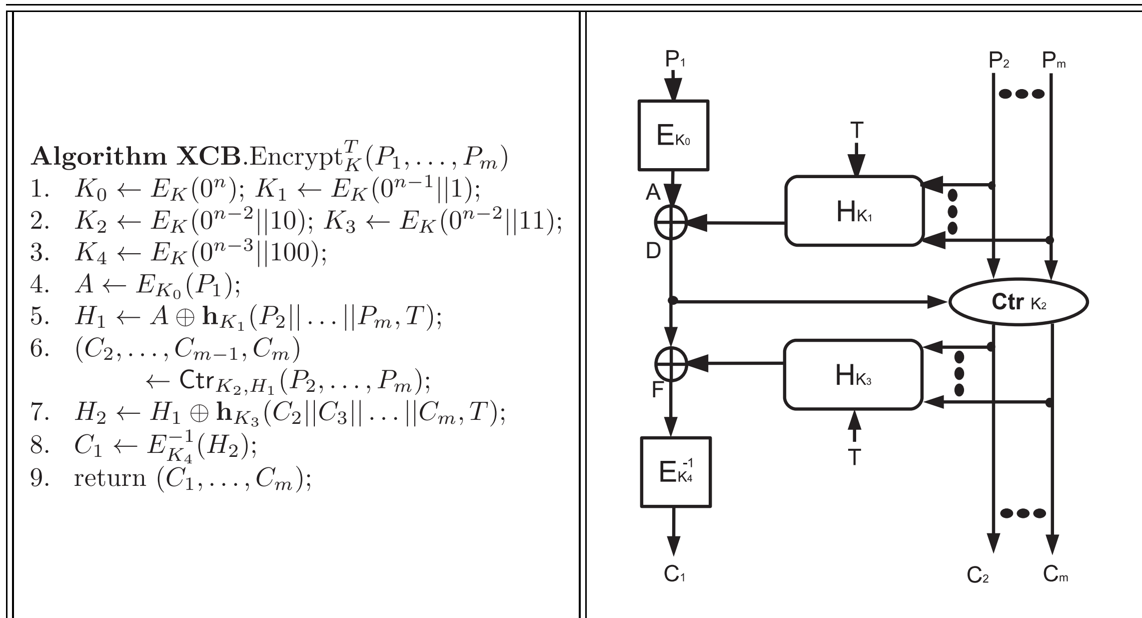
$$\begin{aligned}
\text{First-ECB}(X_1,\ldots,X_m:L) &= (E_K(X_1 \oplus L), E_K(X_2 \oplus xL),\ldots,E_K(X_m \oplus x^{m-1}L)) \\
\text{Last-ECB}(X_1,\ldots,X_m:L) &= (E_K(X_1) \oplus L, E_K(X_2) \oplus xL,\ldots,E_K(X_m) \oplus x^{m-1}L)
\end{aligned}$$

EME has some message length restrictions. If the block length of the underlying block cipher is n, then the message length should always be a multiple of n. Moreover, EME cannot encrypt more than n blocks of messages. This means that if an AES-128 is used as the underlying block-cipher, then EME cannot encrypt more than 2048 bytes (2 KB) of data. This message length restriction was removed in a construction called EME* [63] which



Algorithm HCH.Encrypt$_K^T(P_1, \ldots, P_m)$
1. $R \leftarrow E_K(T)$; $Q \leftarrow E_K(R \oplus \mathsf{bin}_n(|P|))$;
2. $r \leftarrow |P_m|$; $M_m \leftarrow \mathsf{pad}_{n-r}(P_m)$;
3. $M_1 \leftarrow H_{R,Q}(P_1, \ldots, P_{m-1}, M_m)$;
4. $U_1 \leftarrow E_K(M_1)$; $I \leftarrow M_1 \oplus U_1$;
   $S \leftarrow E_K(I)$;
5. $(C_2, \ldots, C_{m-1}, D_m)$
   $\leftarrow \mathsf{Ctr}_{K,S}(P_2, \ldots, P_{m-1}, M_m)$;
6. $C_m \leftarrow \mathsf{drop}_{n-r}(D_m)$; $U_m \leftarrow \mathsf{pad}_{n-r}(C_m)$;
7. $C_1 \leftarrow H_{R,xQ}(U_1, C_2, \ldots, C_{m-1}, U_m)$;
8. return $(C_1, \ldots, C_m)$;

Figure 4.2: HCH, encryption algorithm and block diagram. The function $\mathsf{pad}_{n-r}(P_m)$ adds $(n-r)$ zeros to the end of P_m.



Algorithm XCB.Encrypt$_K^T(P_1, \ldots, P_m)$
1. $K_0 \leftarrow E_K(0^n)$; $K_1 \leftarrow E_K(0^{n-1}||1)$;
2. $K_2 \leftarrow E_K(0^{n-2}||10)$; $K_3 \leftarrow E_K(0^{n-2}||11)$;
3. $K_4 \leftarrow E_K(0^{n-3}||100)$;
4. $A \leftarrow E_{K_0}(P_1)$;
5. $H_1 \leftarrow A \oplus \mathbf{h}_{K_1}(P_2|| \ldots ||P_m, T)$;
6. $(C_2, \ldots, C_{m-1}, C_m)$
   $\leftarrow \mathsf{Ctr}_{K_2,H_1}(P_2, \ldots, P_m)$;
7. $H_2 \leftarrow H_1 \oplus \mathbf{h}_{K_3}(C_2||C_3|| \ldots ||C_m, T)$;
8. $C_1 \leftarrow E_{K_4}^{-1}(H_2)$;
9. return $(C_1, \ldots, C_m)$;

Figure 4.3: XCB, encryption algorithm and block diagram.

requires more block-cipher calls than EME. In [123] the EME mode was further generalized.

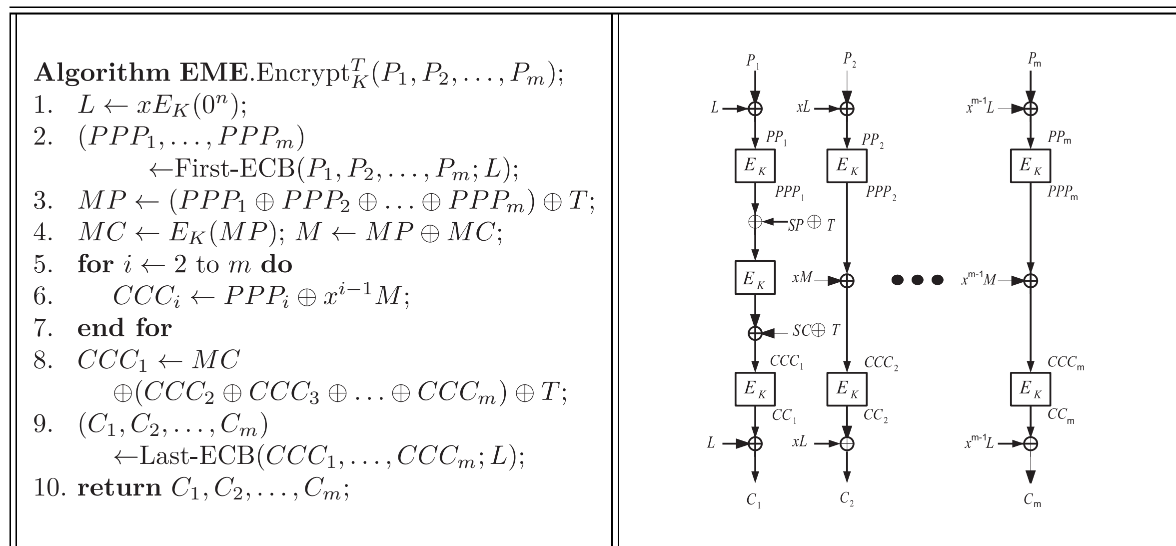


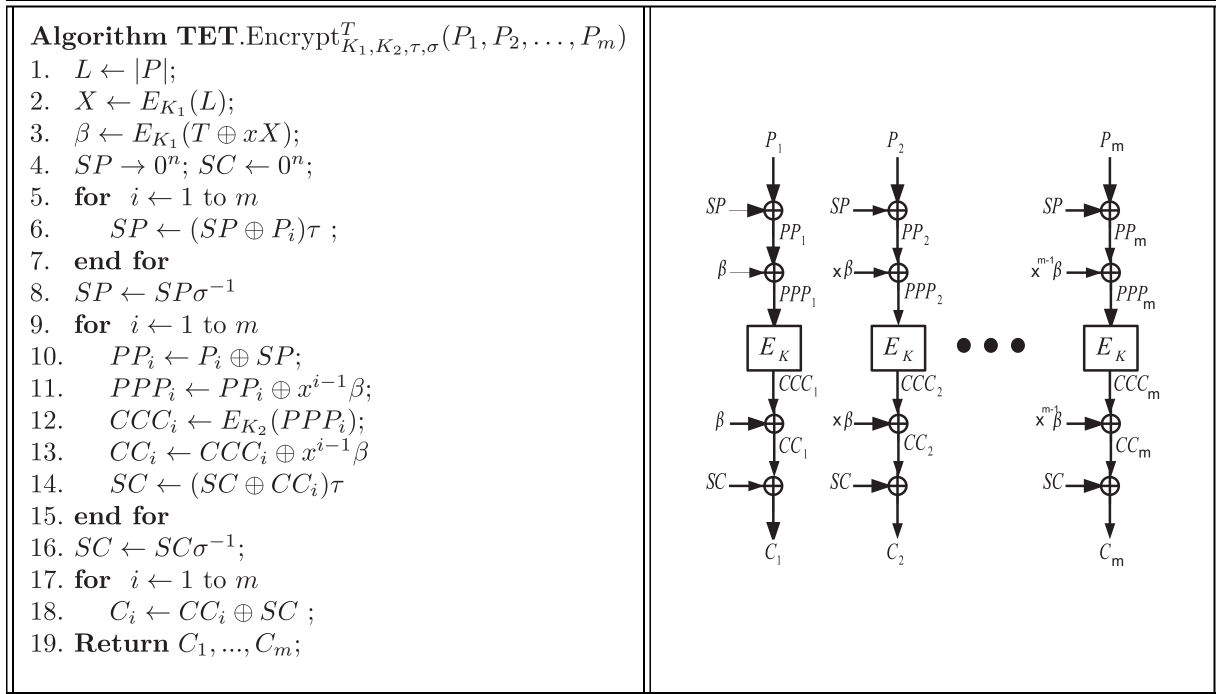
Figure 4.4: EME, encryption algorithm and block diagram.

4.5.3 Hash-Encrypt-Hash

TET and HEH, fall under the category Hash-Encrypt-Hash. As described in the corresponding algorithms in Figure 4.5 and Figure 4.6, these two schemes also use polynomial hash functions similar to the ones included in HCH and HCTR. TET uses two layers of blockwise invertible universal hash functions with an ECB layer of encryption between them. To compute the hash function, TET requires a hash key τ which must meet certain properties. To ensure invertibility of the hash function, τ must be such that for an m block message, $\sigma = \sum_{i=1}^{m} \tau^i \neq 0$, with $\sigma \in GF(2^n)$. For this to be true one requires different hash keys for different message lengths. The authors propose a way to generate the hash key τ from a master key. The encryption algorithm also requires the value of σ^{-1}. This makes TET rather complicated for applications which require encryption of variable length messages. In this Section, we only present a fixed length version of the TET algorithm. Also we assume a fixed value of τ and that σ^{-1} has been pre-computed offline.

HEH is a significant improvement over TET, where the requirement of the inverse computation has been completely removed. The algorithm for HEH as described in Figure 4.6, is also meant for fixed length messages. The specific version of HEH that we present in Figure 4.6 has been named by the authors in [122] as HEHfp.

We summarize in Table 4.1 the characteristics of the various modes that we described. Table 4.1 shows the number of basic operations required by each TES, and also the number of keys associated to them. The security provided by each mode is similar in nature, *i.e.* every



Algorithm TET.Encrypt$^T_{K_1,K_2,\tau,\sigma}(P_1, P_2, \ldots, P_m)$

1. $L \leftarrow |P|$;
2. $X \leftarrow E_{K_1}(L)$;
3. $\beta \leftarrow E_{K_1}(T \oplus xX)$;
4. $SP \rightarrow 0^n$; $SC \leftarrow 0^n$;
5. **for** $i \leftarrow 1$ to m
6. $\quad SP \leftarrow (SP \oplus P_i)\tau$;
7. **end for**
8. $SP \leftarrow SP\sigma^{-1}$
9. **for** $i \leftarrow 1$ to m
10. $\quad PP_i \leftarrow P_i \oplus SP$;
11. $\quad PPP_i \leftarrow PP_i \oplus x^{i-1}\beta$;
12. $\quad CCC_i \leftarrow E_{K_2}(PPP_i)$;
13. $\quad CC_i \leftarrow CCC_i \oplus x^{i-1}\beta$
14. $\quad SC \leftarrow (SC \oplus CC_i)\tau$
15. **end for**
16. $SC \leftarrow SC\sigma^{-1}$;
17. **for** $i \leftarrow 1$ to m
18. $\quad C_i \leftarrow CC_i \oplus SC$;
19. **Return** $C_1, ..., C_m$;

Figure 4.5: TET, encryption algorithm and block diagram.

mode depicted in Table 4.1 has a quadratic security bound. [3]

Table 4.1: Summary of the characteristics of the TES described. We consider a fixed length m-block message for all the schemes.

Mode	Type	BC Calls	Field Mult.	no. of keys
HCH	Hash-Counter-Hash	$m+3$	$2(m-1)$	1
HCHfp	Hash-Counter-Hash	$m+2$	$2(m-1)$	2
HCTR	Hash-Counter-Hash	m	$2(m+1)$	2
XCB	Hash-Counter-Hash	$m+6$	$2(m+1)$	1
EME	Encrypt-Mask-Encrypt	$2(m+1)$	0	1
TET	Hash-Encrypt-Hash	$m+2$	$2m+2$	3
HEH	Hash-Encrypt-Hash	$m+1$	$2m$	2

[3] Note that in the original paper [141] where HCTR was proposed, the author showed a cubic bound which was improved in [28]. The original paper [101], where XCB was proposed did not have a security proof. The security proof of XCB was recently reported in [102].

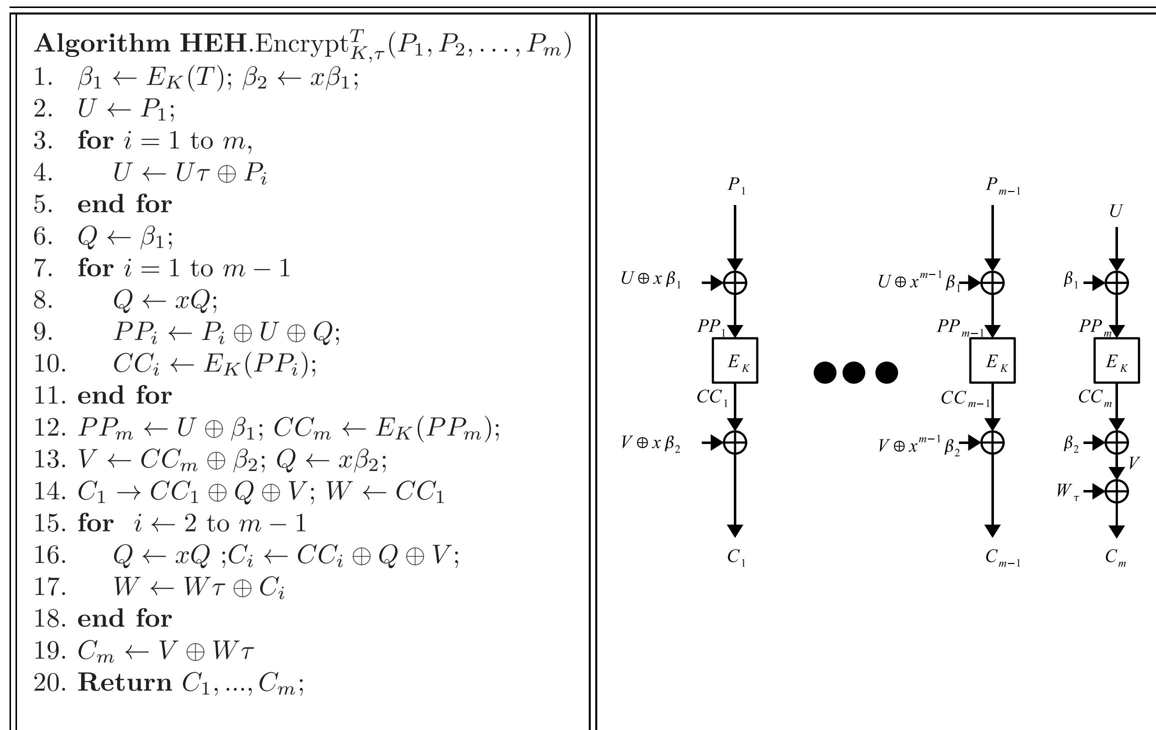


Figure 4.6: HEH, encryption algorithm and block diagram.

4.6 Activities of IEEE SISWG

The current activities surrounding the problem of storage encryption have been broadly directed by an active standardization procedure being performed by IEEE working group on security in storage (SISWG). SISWG has been working towards security solutions in various storage media. SISWG has four task groups: 1619 which deals with narrow block encryption, 1619.1 which deals with tape encryption, 1619.2 deals with wide block encryption and 1619.3 which deals with key management. The task groups 1619 and 1619.2 are responsible for standardizing encryption algorithms for fixed-block storage devices and hard disks fall under this category. Both the task groups (1619 and 1619.2) have concentrated only on length preserving encryption stressing this to be an important criteria for disk encryption. The task group 1629.1 has standardized authenticated encryption for the purpose of tape encryption, as in tapes there exist extra space for the authentication tag.

The task of 1619 has been completed and they have come up with the standard document 1619-2007 which specifies a mode called XTS-AES [42]. XTS is derived from a previous construction of Rogaway [118] which was called XEX. As we mentioned in Section 4.1 XEX is a tweakable block cipher and in [118] it was shown how such a tweakable blockcipher can be used to construct authenticated encryption schemes and message authentication codes. XTS

is different from XEX by the fact that it uses two keys and later it was pointed out in [92] that the usage of two keys was unnecessary. The XTS can be seen as an electronic code book mode of tweakable block-ciphers where each block uses a different tweak, hence the name “narrow block mode". XTS is efficient, fully parallelizable, and possibly does not have any patent claims. But it is questionable whether XTS provides adequate security for the disk encryption problem. XTS is deterministic and there is no scope of authentication, also trivial mix and match attacks are applicable to XTS. These weaknesses are acknowledged in the standard document itself. The document gives a proof of the security of XTS (the validity of the proof has also been contested in [92], where a better proof has been provided), but it only proves XTS as a secure tweakable block-cipher, this assurance is possibly not enough to use XTS for the disk encryption application. This concern has been voiced in other public comments like in [136]. In spite of these criticisms XTS has been standardized by NIST [42].

1619.2 is working on wide block modes, which means they would standardize a mode of operation which behaves like a (tweakable) block-cipher with a block length equal to the data length of a sector. This notion is satisfied by tweakable enciphering schemes (TES), and the security guarantees provided by TES seem adequate for the disk encryption scheme as the ciphertexts produced by them are indistinguishable from random strings and are also non-malleable [65]. Thus, the wide block modes would be much more interesting in terms of security than the XTS mode. But, TES are much more inefficient than XTS. 1612.2 has chosen two modes EME2 (a variant of EME [66]), and XCB [101] for standardization, but the final standard document is not yet out. Among many available TES the reason for choosing EME2 and XCB is not clear. At least the studies presented in this thesis show XCB to be the least efficient mode and both XCB and EME are covered by patent claims.

4.7 Final Remarks

In this Chapter we introduced various aspects of TES including their definition, security and some existing constructions. We also argued about the suitability of TES for the application of disk encryption and the recent standardization activities related to disk encryption. In the next Chapter we provide detailed analysis of some TES from the perspective of hardware implementations and also provide some hard experimental results on the implementations.

Chapter 5

Baseline Hardware Implementations of Tweakable Enciphering Schemes

Emancipate yourself from mental slavery,
none but ourselves can free our mind.

Bob Marley

In spite of numerous activities directed towards design of efficient and secure TES and an active standardization effort [71], little experimental data regarding these schemes were known prior to this work. Authors had mostly used heuristic efficiency arguments based on operation counts to compare between different modes. Such arguments based solely on operation counts, however, are only valid if one assumes a software implementation of the mode. In hardware efficiency estimates based only on operation counts are inadequate, as in hardware there is a possibility of using parallelism which cannot be captured by only operation counts. In this Chapter we present optimized hardware implementation of six TES. The modes we chose are HCH, HCTR, XCB, EME, TET and HEH. Also we provide performance data for a variant of HCH, called HCHfp, which is particularly useful for disk encryption. The rationale behind the choice of these specific modes is discussed next.

The modes that we left out in this study are CMC, PEP and ABL. CMC which use two layers of CBC type encryption cannot be pipelined. PEP and ABL are particularly inefficient compared to their counterparts. We also do not present the implementation of EME* which is a modification over EME, the structure of EME and EME* are same from the hardware implementation perspective. There have been some recent proposals of TES which uses specialized polynomials, these schemes HMCH[BRW] and HEH[BRW] [128] have been dealt with separately in Chapter 6.

For all the implementations we use AES-128 [38] as the underlying block-cipher. Whenever required we use a fully parallel Karatsuba multiplier to compute the hash functions. We carefully analyze and present our design decisions and finally report hardware performance data of the six modes. Our implementations show that in terms of area HCTR, HCH, TET, HEH and XCB require more area than EME. HCTR performs the best in terms of speed followed by HEH, HCHfp, EME, TET, HCH and XCB.

Most of the material presented in this Chapter have been published in a concise form in [97]. As per our knowledge [97] is the first work which reports extensive performance data of TES in various platforms.

5.1 Design Decisions

For implementing all six schemes we chose the underlying block cipher as AES-128. As it was mentioned before, the designs that we present here are directed towards the application of disk sector encryption. In this specific application the messages are all of fixed length and we consider them to be multiples of 128 bits. In particular, our designs are optimized for applications where the sector length is fixed to 512 bytes. As the sector address is considered to be the tweak, thus the tweak length itself is considered to be fixed and equal to one block length of the block cipher.

The speed of a low level disk encryption algorithm must meet the current possible data rates of disk controllers. With emerging technologies like serial ATA and Native Command Queuing (NCQ) the modern day disks can provide data rates around 3Giga-bits per second [130]. Thus, the design objective should be to achieve an encryption/decryption speed which matches this data rate.

The modes HCH, HCTR, XCB, TET and HEH use two basic building blocks, namely, a polynomial xor universal hash and the block-cipher. EME requires only a block-cipher. Since AES-128 was our selection for the underlying block-cipher, proper design decisions for the AES structure must meet the desired speed. Out of many possible designs reported in the literature [55, 60, 69] we decided to design the AES core so that a 10-stage pipeline architecture could be used to implement the different functionalities of the counter mode, the electronic code book (ECB) mode and the encryption of one single block that we will call in the rest of this work *single mode*.

This decision was taken based on the fact that the structure of the AES algorithm admits to a natural ten-stage pipeline design, where after 11 clock cycles one can get an encrypted block in each subsequent clock-cycle. It is worth mentioning that in the literature, several efficient designs with up to 70 pipeline stages have been reported [76], but such designs would increase the latency, i.e., the total delay before a single block of cipher-text can be produced. As the message lengths in the target application are specifically small (in particular the most used sector size is of 512 bytes), such pipeline designs are not suitable for our target application.

The main building block needed in the polynomial hash included in the specification of the HCH, HCTR, XCB and TET modes, is an efficient multiplier in the field $GF(2^{128})$. Out of many possible choices we selected a fully parallel Karatsuba multiplier which can multiply two 128-bit strings in a single clock-cycle at a sub-quadratic computational cost [116]. This

time efficient multiplier occupies about 1.4 times the hardware resources required by one single AES round.[1]. Because of this, the total hardware area of EME (which does not require multipliers) is significantly lesser than that required by the other modes that we study. A more compact multiplier selection would yield significantly lower speeds which violates the design objective of optimizing for speed.

It is noted that the specifications of HCTR, XCB, TET and HCHfp algorithms imply that one multiplicand is always fixed, thus allowing the usage of pre-computed look up tables that can significantly speed up the multiplication operation. Techniques to speed up multiplication by look-up tables are discussed in [12, 99, 103, 132] for the software platform scenario. These techniques can be somehow be extended to hardware implementations also. However, there is a tradeoff in the amount of speed that can be obtained by means of pre-computation and the amount of data that needs to be stored in tables. Significantly higher speeds can be obtained if one stores large tables. This speedup thus comes with an additional cost of area and also the potentially devastating penalty of secure storage. Moreover, if pre-computation is used in a hardware design then the key needs to be hardwired in the circuit which can lead to numerous difficulties in key setup phases and result in lack of flexibility for changing keys. Because of the above considerations, we chose not to store key related tables for our implementations. Thus the use of an efficient but large multiplier is justified in the scenario under analysis. [2] Regarding storage of key related materials we make an exception to this in case of TET. TET requires computation of a inverse, which is a particularly expensive arithmetic operation. In case of TET we store the hash key τ along with the pre-computed value of σ^{-1}, this storage helps us to get rid of a field inversion circuit but does not help us to speed up multiplications.

We implemented the schemes on a FPGA device which operates at lower frequencies than true VLSI circuits. Thus the throughput that we obtain probably can be much improved if we use the same design strategies on a CMOS VLSI technology. Our target device was a XILINX Virtex 4, xc4v1x100-12FF1148.

5.2 Implementation of Basic Blocks

In this Section we describe how the basics building blocks TES were implemented, i.e., the AES and polynomial hash using a Karatsuba multiplier.

[1]For specific experimental details see Table 5.1.

[2]The same design decision was taken in [99].

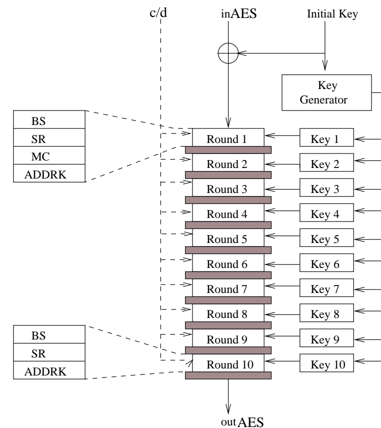


Figure 5.1: AES Pipeline Architecture

5.2.1 The AES Design

In general terms, the AES algorithm is a sequence of pre-defined transformations which process the input data, depending on the key length used, in 10, 12 or 14 iterations called rounds. The last round differs from the others. AES uses one of three cipher key-lenghts 128, 192 or 256 bits, and has a fixed block length of 128 bits. For our implementations we choose the AES with key length of 128 bits, such an AES operates in ten rounds. Each round can be further sub-divided into four basic steps, namely, *shift rows*, *byte substitution*, *mix columns* and *add round key*. The last round is different from the others in the sense that in this round the mix columns operation is not performed. In addition to the rounds AES requires a key expansion algorithm which expands the input 128 bit key to more key bits to be used independently in each round.

We implemented two AES, one sequential architecture that yields a valid output after 11 clock cycles and another based on a 10-stage pipeline structure able to encrypt one block per cycle after eleven cycles of latency. Both implementations utilized double port memories for computing the AES Byte Substitution (BS) transformation.

For the sequential design, we implemented only one round that contains the four AES steps along with a multiplexer block that eliminates the MixColumn step in the tenth round as is shown in Figure 5.1. The control circuit consists of one 4-bit ascending/descending order counter for encryption/decryption, respectively. The counter output points to the correct address where the keys are stored and it controls the multiplexer **M1** that feedbacks the round or it allows that a new input data comes in, and the multiplexor **M2** that controls the omission of the MC transformation in the tenth AES round. For decryption we followed the procedure described in [38]. It is worth mentioning that the key schedule process is accomplished after 10 clock cycles. Each round key so generated is stored in a 128x32 RAM memory.

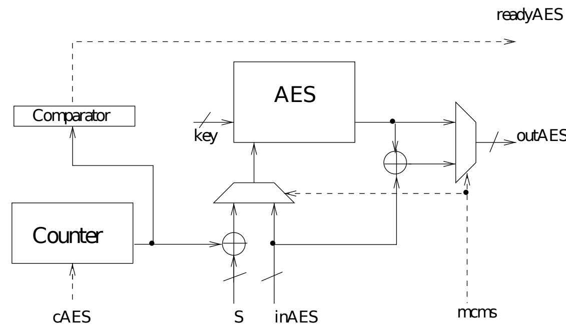


Figure 5.2: AES Architecture in sequential and Counter Modes

In the AES pipeline implementation the 10 AES rounds are unrolled, while the key generation is computed sequentially and each one of the round keys is stored in a register directly connected to the corresponding round as is shown in Figure 5.1. Because of synchronization purposes, each AES round is isolated from the preceding one by a latch circuit. This implementation does not use a control unit, since this is added with the help of outside circuitry whenever the AES core will be used in simple or counter mode. In this design, starting from cycle eleven, valid outputs will be produced every clock cycle.

In the designs of the modes we need to use the AES in different ways. For example, in case of HCTR and HCH we require to encrypt the bulk information using a counter mode and also at times only a single block of message is required to be encrypted. In case of TET and HEH we need an electronic code book (ECB) mode along with the capability of encrypting single blocks. This different functionalities can be easily obtained by two different encryption cores one suited for one block encryption (the single mode) and the other for encryption of multiple blocks. But, for most applications, the implementation of two separate AES cores is prohibitive in terms of cost. Therefore, we decided to use a single AES core that can be programmed for implementing both functionalities, namely, the counter mode (or ECB mode) and the single mode computation. This is realized by the circuit shown in Figure 5.2.

5.2.2 The Design of the Multiplier

Our strategy for multiplication is based on the Karatsuba multiplier as it was presented in [116]. The Karatsuba multiplier enjoys a superb sub-quadratic complexity of $O(n^{log_2 3})$ bit operations for multiplying two polynomials of degree less than n as we will briefly explain next. Let the field $GF(2^{128})$ be constructed using the irreducible polynomial $q(x) = x^{128} + x^7 + x^2 + x + 1$. Let A, B be two elements in $GF(2^{128})$. Both elements can be represented in the polynomial basis as,

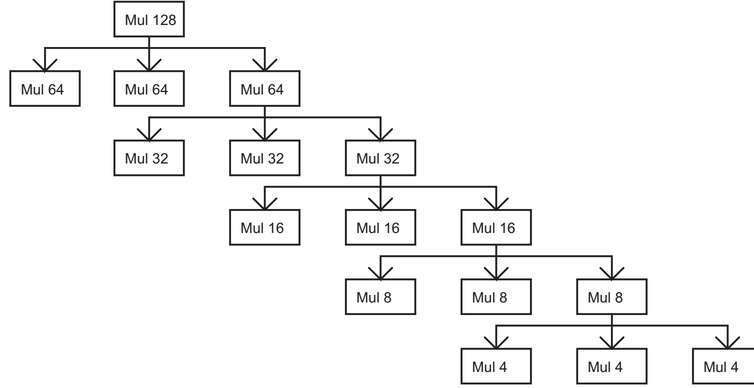


Figure 5.3: Structure of the Karatsuba Multiplier

$$\begin{aligned} A &= \sum_{i=0}^{127} a_i x^i = \sum_{i=64}^{127} a_i x^i + \sum_{i=0}^{63} a_i x^i \\ &= x^{64} \sum_{i=0}^{63} a_{i+64} x^i + \sum_{i=0}^{63} a_i x^i = x^{64} A^H + A^L \end{aligned}$$

and

$$\begin{aligned} B &= \sum_{i=0}^{127} b_i x^i = \sum_{i=64}^{127} b_i x^i + \sum_{i=0}^{63} b_i x^i \\ &= x^{64} \sum_{i=0}^{63} b_{i+64} x^i + \sum_{i=0}^{63} b_i x^i = x^{64} B^H + B^L \end{aligned}$$

Then, using last two equations, the polynomial product is given as

$$AB = x^{128} A^H B^H + (A^H B^L + A^L B^H) x^{64} + A^L B^L. \tag{5.1}$$

Karatsuba algorithm is based on the idea that the product of last equation can be equivalently written as

$$\begin{aligned} C &= x^{128} A^H B^H + A^L B^L + \\ &\quad (A^H B^H + A^L B^L + (A^H + A^L)(B^L + B^H)) x^{64} \\ &= x^{128} C^H + C^L. \end{aligned} \tag{5.2}$$

It is easy to see that Eq. (5.2) can be used to compute the product at a cost of four

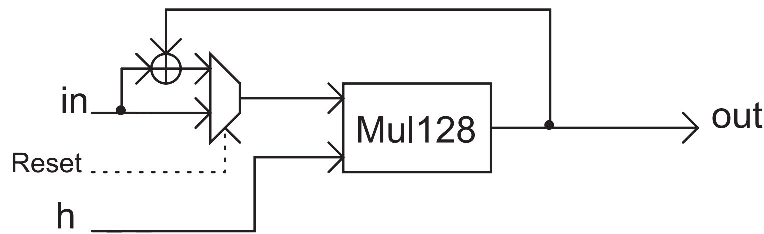


Figure 5.4: Horner's rule main Architecture

polynomial additions and three polynomial multiplications. In contrast, when using Eq. (5.1), one needs to compute four polynomial multiplications and three polynomial additions. Karatsuba algorithm can be applied recursively to the three polynomial multiplications in (5.2). Hence, we can postpone the computations of the polynomial products $A^H B^H, A^L B^L$ and $(A^H + A^L)(B^L + B^H)$, and instead we can split again each one of these three factors into three polynomial products. By applying this strategy recursively, in each iteration each degree polynomial multiplication is transformed into three polynomial multiplications with their degrees reduced to about half of its previous value.

Figure 5.3 shows the typical tree structure of a $GF(2^{128})$ Karatsuba multiplier. The polynomial multiplier shown in Figure 5.3 returns a 256-bit polynomial which we need to reduce back to 128 bit using the irreducible polynomial, $q(x) = x^{128} + x^7 + x^2 + x + 1$.

For implementing the hash functions required in the modes we need to evaluate polinomials in $GF(2^n)$ we use the Horner's rule to evaluate polynomials. The field multiplier is the main building block for implementing the Horner's rule Algorithm described in Fig 5.5. The corresponding hardware architecture is shown in Figure 5.4. The implementation of the hash functions required in the modes are based on this module.

5.3 The Design Overviews

In this Section we give a careful analysis of the modes' data dependencies and we explain how the parallelism present in the algorithms can be exploited. In the analysis which follows we assume the message to be of 512 bytes (32 AES blocks). Furthermore, we assume a single AES core designed with a 10 stage pipeline and a fully parallel single clock cycle multiplier. We also calculate the key schedules for AES on the fly, this computation can be parallelized with the AES rounds. The polynomial universal hash functions are computed using the Horner's rule shown in the algorithm of Figure 5.5:

Algorithm $\mathbf{HORN}_h(X_1, \ldots, X_m)$
$Y \leftarrow 0^n$;
for $i = 1$ to m,
    $Y \leftarrow (Y \oplus X_i)h$;
end for
return Y

Figure 5.5: The Horner's Rule

HCH mode of Operation

Referring to the Algorithm of Figure 5.6 (a) the algorithm starts with the computation of the parameter R in Step 1. For computing R the AES pipeline cannot be utilized and must be accomplished in simple mode, implying that 11 clock cycles will be required for computing R. At the same time, the AES round keys can be computed by executing concurrently the AES key schedule algorithm. The hash function of Step 3 can be written as

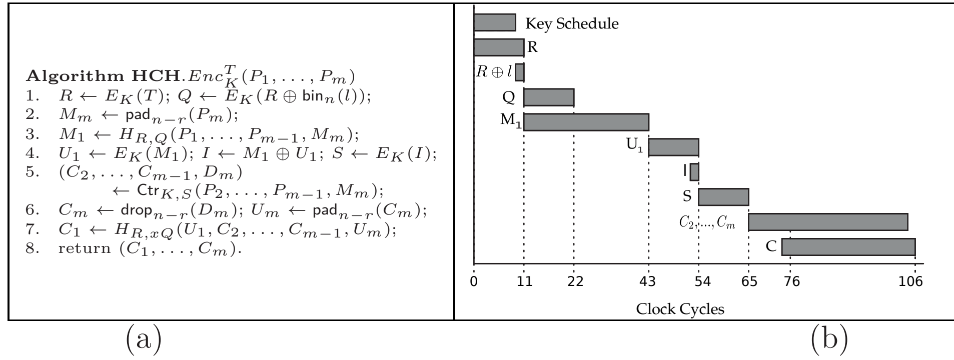
$$H_{R,Q}(P_1, P_2, \ldots P_{32}) = P_1 \oplus Q \oplus Z$$

where $Z = \mathbf{HORN}_R(P_2, \ldots P_{32})$. So, Z and Q can be computed in parallel. For computing Z, 31 multiplications are required and computation of Q takes 11 clock cycles. So the computation of the hash in step 2 takes 31 clock cycles. Then, the computation of Step 4 requires two simple mode encryption which implies 22 more clock cycles. So we need to wait 64 clock cycles before the counter mode starts. The counter mode in step 5 requires 31 blockcipher calls which can be pipelined. So computation of step 5 requires a total of $30 + 11 = 41$ clock cycles. The first cipher block C_2 is produced 11 clock cycles after the counter starts. The second hash function computation of Step 7 can start as soon as C_2 is available in the clock cycle 75. Hence the computation of the hash function can be completed at the same time that the last cipher block (C_m) of Step 5 is produced. Figure 5.6 (b) depicts above analysis. It can be seen that a valid output will be ready after the cycle 75 and a whole disk sector will be ready in the cycle 106.

In case of HCHfp the computation of Q is not required, and it uses a hash key which is different from R. Thus R and the hash function can be computed in parallel, which gives rise to a savings of 11 clock cycles. So HCHfp will produce a valid output in 64 clocks and it will take 95 clock-cycles to encrypt the 32 block message (see 5.7(b)).

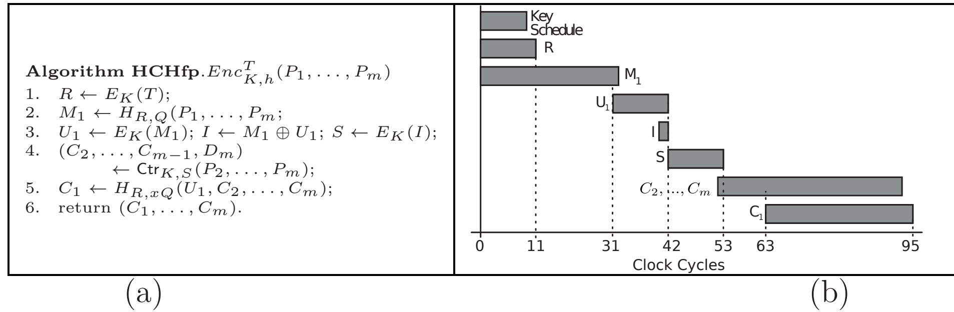
HCTR Mode of Operation

Referring to the Algorithm of Figure 5.8 (a), the computation of the hash function of Step 1 requires 33 clock cycles. At the same time, the design can derive the AES round keys



(a) (b)

Figure 5.6: (a) Encryption using HCH. The tweak is T and the key is K. For $1 \leq i \leq m-1$, $|P_i| = n$ and $|P_m| = r$ where $r \leq n$. (b) Timing diagram for HCH.



(a) (b)

Figure 5.7: (a) Encryption using HCHfp. The tweak is T and the key is (K, h). For $1 \leq i \leq m$, $|P_i| = n$. (b) Timing diagram for HCHfp.

by executing concurrently the AES key schedule algorithm. Then, the computation of the parameter CC in Step 2, must be accomplished in simple AES mode, implying that 11 clock cycles will be required for completing that calculation. As in HCH mode, the $m-1$ block cipher calls included in Step 4 are performed in counter mode, which once again can be computed in parallel via the pipeline architecture. Hence, the computation of all the C_i for $i = 2, \ldots, m = 32$, can be computed in $(32-1)+11 = 42$ clock cycles. At the same time, the second hash function computation of Step 7 can start as soon as C_2 is available in the clock cycle 56. Hence the computation of the hash function can be completed at the same time that the last block cipher (C_m) of Step 5 is produced. Figure 5.8(b) depicts the timing analysis just given. It can be seen that a valid output will be ready after the cycle 55 and a whole disk sector will be ready in the cycle 88.

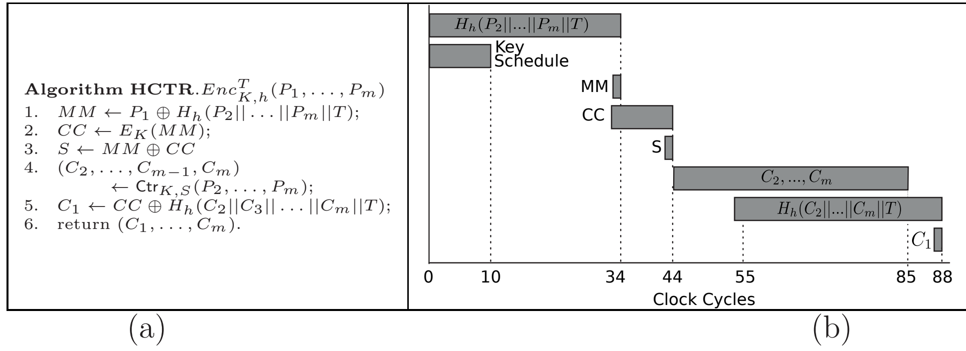


Figure 5.8: (a) Encryption using HCTR. The tweak is T and K is the block-cipher key and h the hash key. (b) Timing diagram for HCTR.

XCB Mode of operation

The computation of XCB starts with the derivation of the five keys (see Figure 5.9 (a)). Derivation of each key requires a block-cipher call. These five block-cipher calls can be parallelized thus requiring 15 clock cycles. The computation of the first hash requires 33 clock cycles and in the mean time the computation of the two key schedules and A can be completed. The computation of the first hash which require 33 clock cycles can thus be completed within clock-cycle 49. Then the counter mode starts, which requires 41 clock cycles to complete. The second hash can start at clock cycle 59 and thus it is completed in clock-cycle 93. For computing the last block, an AES decryption call is required with a different key. Hence, completing the key schedule and the decryption requires another 21 clock cycles, and a reset operation is necessary before computing the new key schedule. This thus gives rise to a requirement of 115 clock cycles to encrypt the whole sector. But the first cipher block would be ready in case of XCB after clock cycle 59.

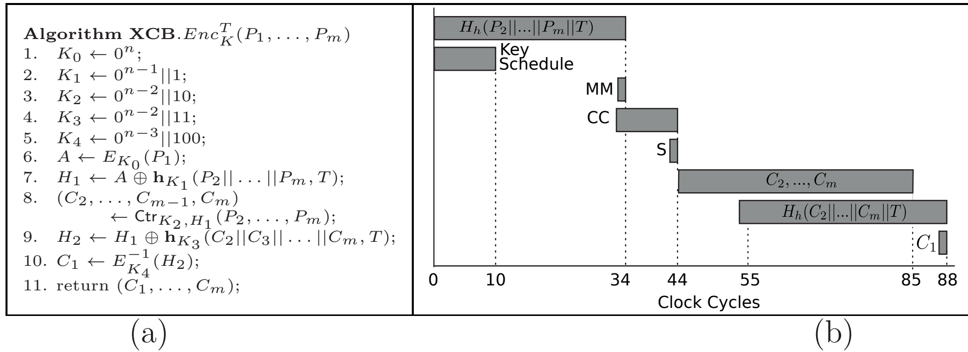
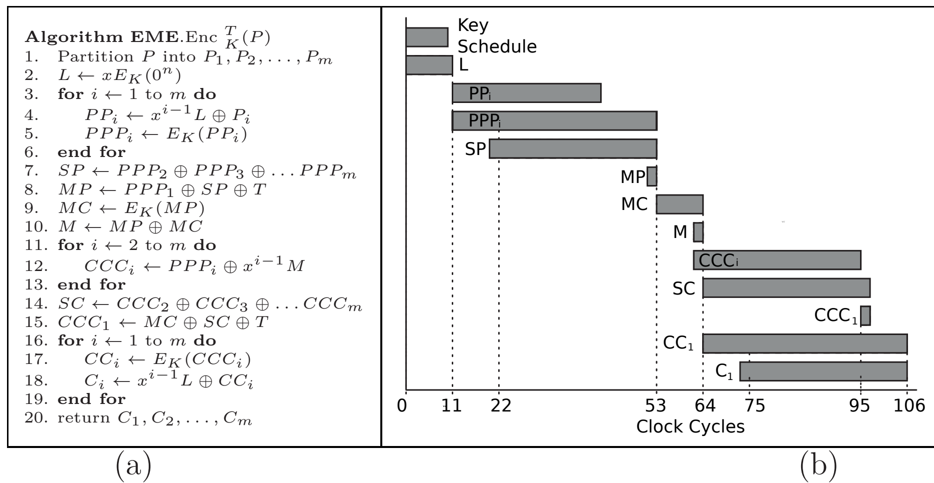


Figure 5.9: (a) Encryption using XCB. The tweak is T and K is the key from which two different hash keys and 3 different block-cipher keys are derived. (b) Timing diagram for XCB.

EME mode of Operation

Referring to the Algorithm of Figure 5.10 (a), the computation of the parameter L in Step 2, must be accomplished in a sequential fashion, implying that 11 clock cycles will be required for completing that calculation. Thereafter, the 32 block cipher calls included in Steps 3-6 can be accomplished using the benefits of the parallelism associated to the pipeline approach. So, the computation of all the PPP_i for $i = 1, 2, \ldots, m = 32$, can be computed in $(32 - 1) + 11 = 42$ clock cycles. On the contrary, the cipher call in Step 9 for obtaining MC must be performed in a sequential fashion, which implies 11 extra clock cycles. The second layer of encryption can also be performed in 42 clock cycles and the operations $x^i M$ and $x^i L$ in steps 12 and 18 can be parallelized with the block-cipher calls. So to complete encryption of 32 blocks EME should require $11 + 42 + 11 + 42 = 106$ clock cycles. And the first block of valid output would be produced after 75 clock cycles. Some pre-computations may save some of the EME costs. For example, L in Step 2 is a quantity only dependent on the key K, thus, L can be pre-computed yielding the saving of 11 clock cycles. But this will require storage of key materials. Figure 5.10 (b) shows the EME operations that are suitable for being computed in parallel.



Algorithm EME.Enc $^T_K(P)$
1. Partition P into $P_1, P_2, \ldots, P_m$
2. $L \leftarrow xE_K(0^n)$
3. **for** $i \leftarrow 1$ to m **do**
4. $\quad PP_i \leftarrow x^{i-1}L \oplus P_i$
5. $\quad PPP_i \leftarrow E_K(PP_i)$
6. **end for**
7. $SP \leftarrow PPP_2 \oplus PPP_3 \oplus \ldots PPP_m$
8. $MP \leftarrow PPP_1 \oplus SP \oplus T$
9. $MC \leftarrow E_K(MP)$
10. $M \leftarrow MP \oplus MC$
11. **for** $i \leftarrow 2$ to m **do**
12. $\quad CCC_i \leftarrow PPP_i \oplus x^{i-1}M$
13. **end for**
14. $SC \leftarrow CCC_2 \oplus CCC_3 \oplus \ldots CCC_m$
15. $CCC_1 \leftarrow MC \oplus SC \oplus T$
16. **for** $i \leftarrow 1$ to m **do**
17. $\quad CC_i \leftarrow E_K(CCC_i)$
18. $\quad C_i \leftarrow x^{i-1}L \oplus CC_i$
19. **end for**
20. return $C_1, C_2, \ldots, C_m$

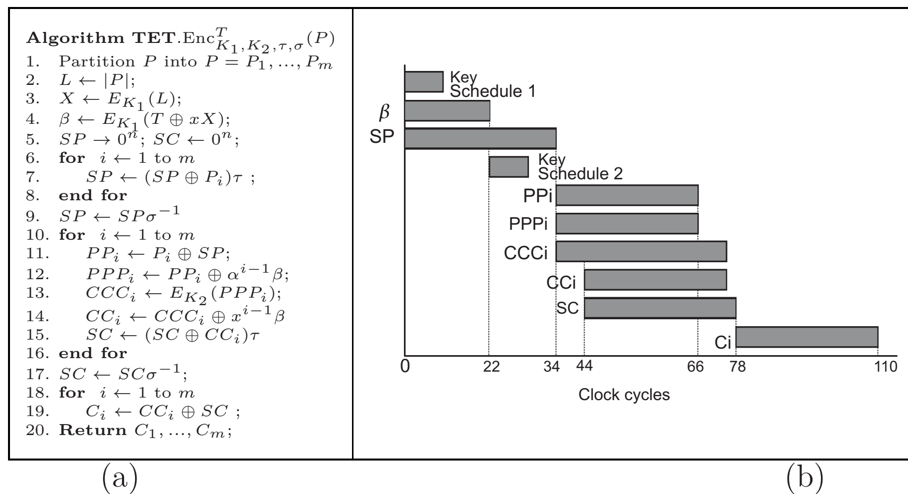
(a) (b)

Figure 5.10: (a) Encryption using EME. (b) Timing diagram for EME.

TET mode of Operation

Referring to the algorithm of Figure 5.11, the computation of the parameter β requires two AES calls in the simple mode, which can be accomplished in 22 clock cycles. The computation of SP can be done in parallel. Computation of SP will require 32 multiplications which can be completed in 33 clock cycles. In encryption it requires an extra multiplication with σ^{-1}, Thus the computation of SP would be complete in 34 clock cycles. In the mean-time

the key schedule for the second block-cipher key can also be completed. The computation of PP_i and PPP_i can be parallelized and they can be computed in 33 clock cycles. As soon as PPP_1 is available (which would be available at clock cycle 35), computation of CCC_i can start. Computation of CCC_i, $i = 1, \ldots, 32$ will take a total of 32 block cipher calls which can be completed in 42 (32 + 10) clock cycles. Thus, after clock-cycle 78 all CCC_i s would be ready. The computation of the final cipher texts C_i-s would take another 32 clock cycles. Thus, the whole disk would be ready after 110 clock cycles. And the first cipher block would be ready after 79 clock cycles. Note that in this analysis we do not consider computation of the inverse, which may give rise to a significant increase in the number of the required clock cycles.



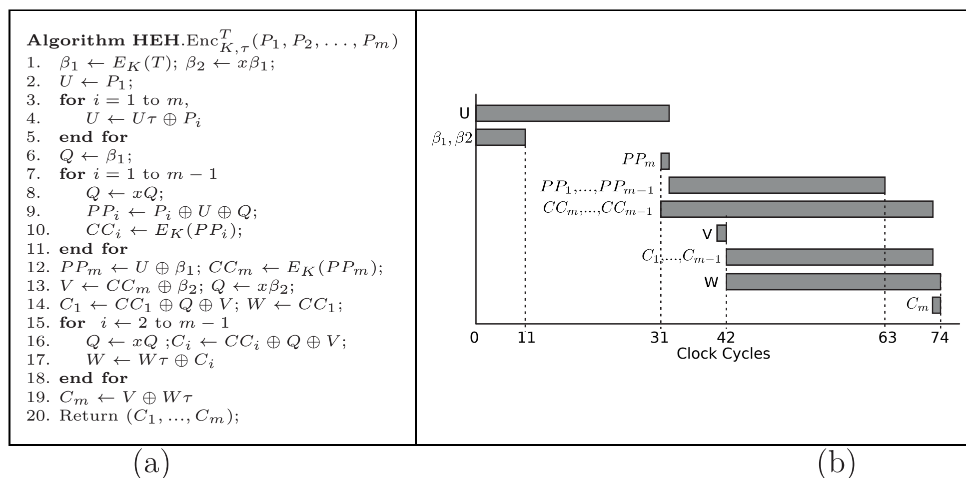
(a) (b)

Figure 5.11: (a) Encryption using TET. K_1 and K_2 both are block cipher key, τ is the hash key and σ is its multiplicative inverse. (b) Timing diagram for TET.

HEH mode of Operation

HEH is a significant improvement over TET, where the requirement of the inverse computation has been completely removed. The algorithm for HEH as described in Figure 5.12(a), is also meant for fixed length messages. The specific version of HEH that we present in Figure 5.12(a) has been named by the authors in [122] as HEHfp. HEHfp receives a hash key τ as an input, the computation of polynomial hash U is performed in parallel with the computation of β_1 and β_2. After 11 clock cycles β_1 and β_2 are ready, U is finished in 31 clock cycles. Having the value of U PP_m is computed, then the computation of CC_m is performed and after 11 clock cycles we obtain the values V. After one clock cycle the computation of PP_i is started and each PP_i feed the AES to generate values CC_i. In clock cycle 42 we obtain V and we can compute the C_is in parallel with W that represents the second polynomial hash, after 31 clock cycles C_m can be computed. Finally after 74 clock cycles the total sector of

the disk will be encrypt. The improvement of TET achieved by HEH is also in the efficiency while TET takes 110 clock cycles to encrypt 512 bytes, HEH takes only 74. The reason is that the second hash in HEH can be parallelized with the computation of cipher text C_i, in TET that is impossible.



(a) (b)

Figure 5.12: (a) Encryption using HEH. (b) Timing diagram for HEH.

5.4 Implementation Aspects

In practice, the timing performance of a generic hardware design will be determined by its associated critical path and the number of clocks required to complete the computation. In addition to the time, another important aspect which is required to be considered is the area. As for the area utilization of a given design, some of the factors that have impact in hardware resource utilization include: the number of temporary variables involved in the design (which implies extra registers) and the number of possible inputs that the main building blocks may have (which translates in extra multiplexer blocks).

Table 5.1: Performance of an Encryption/Decryption AES round and a 128-bit Fully-parallel Karatsuba multiplier

Design	Slices	B-RAM	Critical Path(nS)
Full AES round	1215	8	10.998
Encryption-only AES round	298	8	6.71
multiplier	3223	-	9.85

In order to give a rough estimation of the critical path associated to each one of the modes, we show in Table 5.1 the performance of the architectures' main building blocks, namely,

a key generation/encryption/decryption AES round, an encryption-only AES round, and a 128-bit fully-parallel Karatsuba multiplier. [3]

Considering the utilization of B-RAMs and slices, the size of one full AES round in our design is about 30% smaller than that of the Karatsuba multiplier. However, the critical path delay associated to an encryption/decryption AES round, is longer than that of the multiplier block by about 10%.

Table 5.2: Hardware Resources Utilized by the Six TES

Modo	Mux inAES	Extra B-RAM	Mul *xtime*	Registers 128 bits	Mux 2×128	Mux 3×128
HCTR	3×128	-	-	3	2	1
HCHfp	4×128	-	1	5	5	-
HCH	5×128	-	1	6	5	-
EME	4×128	2	3	5	5	-
TET	3×128	2	3	2	4	-
XCB	4×128	-	-	8	6	2
HEH	3×128	-	4	4	1	-

Table 5.2 shows the hardware resource usage by each of the six TES modes of operation. Besides an obvious impact in the area complexity of the modes, the resources occupied by each TES tend to increase its critical path. In the rest of this Section, we briefly analyze the resource utilization and timing potential performance of the five TES modes under analysis.

HCH and HCHfp

For HCH and HCHfp the critical path will be also lower bounded by the minimum critical path between the AES core and the hash function. Considering the values given in Table 5.1 the maximum throughputs that we can expect for these two modes when using the full and the encryption-only AES cores is 3.8 Gbps and 4.24 Gbps, respectively. However, we should expect that HCH and HCHfp timing performances will be appreciably lower than those bounds, because these modes requires six and five extra registers for temporary variables, respectively. Moreover as shown in Table 5.2, the possible inputs for the AES core and the hash function is more than that of HCTR, which will force us to use multiplexer blocks with more inputs.

HCTR

As shown in Table 5.2, HCTR requires three extra 128-bit registers in order to allocate temporary computation values and also the possible inputs for the AES core and the hash

[3]The experimental results shown in Table 5.1 correspond to a place &route simulations using Modelsim XE III 6.0d and Xilinx ISE 8.3 and a Xilinx Virtex4 XC4VLX100-12FF1148 FPGA as a target device.

function is three. This feature makes HCTR both, a fast an economical TES mode. The critical path of HCTR will be lower bounded by the one associated to either the hash function or the AES core, whichever is larger. Therefore, and according to the critical paths reported in Table 5.1, we can expect that an implementation of HCTR will have a critical path of at least 10.998ηS when using the full AES core and 9.85 ηS when using the encryption-only AES round. In terms of throughput, this translates to a maximum of 4.185 Gbps and 4.672 Gbps, respectively.

XCB

Out of the six modes analyzed, XCB is both, the most expensive in terms of hardware resource utilization, and the slowest. XCB's latency however, is relatively low but the total time required is quite high. Among other factors, XCB's total time is high because in its final step the calculation of E_K^{-1} cannot start till a key schedule computation of C_1 has finished. In total, 21 clock cycles are consumed in that final step. As shown in Table 5.2, storage of XCB temporary variables requires eight extra 128-bit registers and one 128-bit four-to-one multiplexer. Thus, the maximum throughput that one can expect for XCB when using the full and the encryption-only AES cores should be significantly lower than 3.21 Gbps and 3.59 Gbps, respectively.

EME

In the case of EME, its most notorious building block is the AES core. However, other smaller components that have some impact in the EME performance are the *xtime* multiplication algorithm along with the chain additions characteristic of this mode. Since the *xtime* operation can be performed efficiently in hardware, the critical path of EME is mainly given by that of the AES core utilized. Hence, we can expect that an implementation of EME will have a critical path of at least 10.998ηS when using the full AES core and 6.71 ηS when using the encryption-only AES round. In terms of throughput, this translates to a maximum of 3.48 Gbps and 5.71 Gbps, respectively. Regarding area utilization, EME is consistently the most economical TES mode. However, the computation of EME requires to store all the PPP_i values for $i = 2, \ldots, m$. This issue was solved by utilizing two extra FPGA block RAMs as reported in Table 5.2.

TET

Once again, the critical path of TET will be lower bounded by the minimum critical path between the AES core and the hash function. According to Table 5.1, the maximum throughput that we can expect for TET when using the full and the encryption-only AES cores is

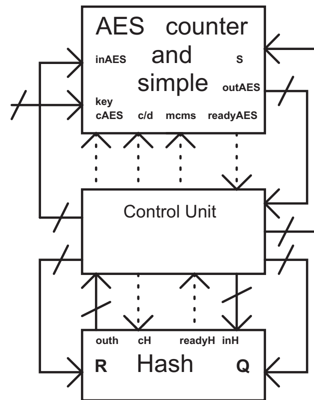


Figure 5.13: HCH Main Architecture

3.36 Gbps and 3.75 Gbps, respectively. As reported in Table 5.2, the computation of TET requires the allocation of two extra FPGA block RAMs. Moreover, two 128-bit registers and four 128-bit two-to-one multiplexers blocks are required.

HEH

In HEH the same resources like TET would be required. But HEH does not require computation of the inverse, which is a good advantage over TET. HEH do not require extra storage elements (BRAMs) like TET.

5.5 Implementation of HCH

As a representative design example we shall discuss the architecture of HCH in details. The architectures for the other modes are quite similar and we do not discuss them here, but we shall discuss the experimental results of all the modes in Section 5.6.

Figure 5.13 shows the general architecture of the HCH mode of operation. It can be seen that AES must be implemented both, in counter and in simple mode. Moreover a hash function is also required as one of the main building blocks. The architecture operation is synchronized through a control unit that performs the adequate sequence of operations in order to obtain a valid output.

The HCH control unit architecture is shown in Figure 5.14. It controls the AES block by means of four 1-bit signals, namely: **cAES** that initializes the round counter, the **c/d** signal that selects between encryption or decryption mode, the **msms** signal that indicates whether one single block must be processed or rather, multiple blocks by means of the counter mode. Finally, **readyAES** indicates whenever the architecture has just computed a valid output.

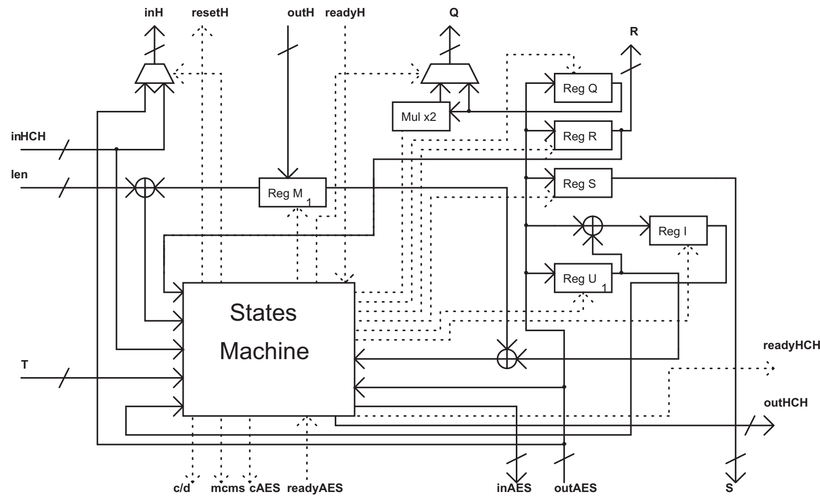


Figure 5.14: HCH Control Unit Architecture

The AES dataflow is carried out through the usage of three 128-bit busses, namely, **inAES** that receives the blocks to be encrypted, **outAES** that sends the encrypted blocks and **S** that receives the initialization parameter for the counter mode. The communication with the hash function block is done using two signals: **cH** for initializing the accumulator register and the counter of blocks already processed and **readyH** that indicates that the hash function computation is ready. The data input/output is carried by the **inH** and **outH** busses, respectively. The parameters **R** and **Q** are calculated in the control unit and send through the busses to the hash function.

The HCH control unit implements a finite state automaton that executes the HCH sequence of operations. It defines the following eight states: *RESET*, *AES1*, *AES2*, *HASH1*, *AES3*, *AES4*, *ECOUNTER* and *HASH2*. In each state, an appropriate control word is generated in order to perform the required operations. The correct algorithm execution requires storing the **R**, **Q**, **S**, **I**, $\mathbf{U}_1$ and $\mathbf{M}_1$ values. Thus, six extra 128-bit registers are needed. In particular the hash function input **inH** can come from the system input or from the output of the AES counter mode. Therefore, a multiplexer is needed for addressing the correct input, where the multiplexer signals are handled by the state machine's control word. We compute the **xQ** signal by means of an *xtimes* operation in the finite field GF(2^{128}), which was implemented as described in for example [38, 118].

The sequence of operations described in algorithm in the Figure 5.6(a) is performed through the execution of the state machine diagram shown in Figure 5.15. The state transition among states is controlled by two signals, namely, **readyAES**, which indicates that the current output in the bus **outAES** is valid and; **readyH** that indicates that the computation of the hash function has just been completed.

In *RESET* the AES key schedule process starts and the value in T is assigned to **inAES**.

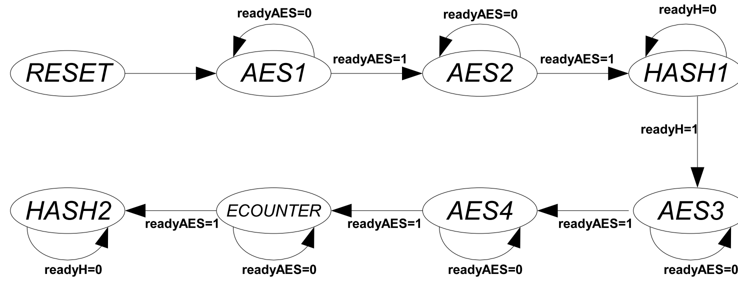


Figure 5.15: HCH State Machine Diagram

Then, the control is transferred to *AES1*. In *AES1*, the value $R = E_k(T)$ is computed concurrently with the key generation process and when the signal **readyAES** becomes active the value in **outAES** is transferred to the register **regR**, the round counter is re-initialized, the value $R \oplus len$ is computed and the control is transferred to the state *AES2*. In *AES2* the value $Q = E_k(R \oplus len)$ is computed and at the same time the computation of the hash value $\mathbf{M}_1$ starts. When the signal **readyAES** becomes active the round counter is re-initialized, the value in **Q** is transferred to the register **regQ**, **xQ** is computed and the automaton switches to state *HASH1*. In state *HASH1* the computation of the hash value $\mathbf{M}_1$ is completed, and when the signal *readyH* is active the hash result is stored in the register $\mathbf{M}_1$ and the automaton transitions to *AES3*. In the state *AES3* the value $U_1 = E_k(M_1)$ is computed, and when the signal **readyAES** becomes ready, the value $I = M_1 \oplus U_1$ is computed and stored in **RegI** and the control is transferred to the state *AES4*. In *AES4*, the value $S = E_k(I)$ is computed and when the signal **readyAES** is ready that value is stored in **RegS** and then we arrive to the state *ECOUNTER*. In the state *ECOUNTER* the AES multiple block encryption in counter mode starts, when the signal **readyAES** is activated the hash function initiates and the automaton switches to the state *HASH2*. When in state *HASH2*, the encryption of $C_2, \ldots, C_m$ in counter mode is performed in parallel with the computation of the hash function. Finally, when the signal *readyH* is activated we have the hash result in C_1.

En each one of the states mentioned above, a 14-bit control word *cword* that orchestrates all the architecture modules is continuously updated. The organization of the control word is summarized in Table 5.3.

The dataflow for encryption and decryption is essentially the same. The only modification is to determine whether **Q** or **xQ** should be used in the two hash function calls. This decision is taken in our architecture with the help of a multiplexer block whose output is the input hash signal **Q** (se Figure 5.15). In the first call to the hash function block, and when the state is *HASH1* and the mode signal is off that multiplexer selects **Q**, otherwise, it selects **xQ**. In the second hash function call, **xQ** is selected if the automaton is in the state *HASH2* and the mode signal is off, otherwise, it selects **Q**.

Table 5.3: Control word Specification permutation.

Control Word bits	**Functionality**
$cword_0$	Synchronizes the input dataflow
$cword_1$	Indicates whether the hash function input comes from the system input or rather, from the output of the AES in counter mode
$cword_2$	hash function reset
$cword_4$	round counter reset
$cword_5$	AES in simple or counter mode
$cword_6$	AES counter mode ready forcomputing a new plaintext
$cword_7$	Load signal for register $\mathbf{R}$
$cword_8$	Load signal for register $\mathbf{Q}$
$cword_9$	Load signal for register $\mathbf{U}_1$
$cword_{10}$	Load signal for register $\mathbf{M}_1$
$cword_{11}$	Load signal for register $\mathbf{S}$
$cword_{12}$	Determines if the AES will be working according to the stipulated in the input signal or in encryption mode
$cword_{13}$	Load signal for register $\mathbf{I}$

Fixed Length HCH

In the specific case of the disk encryption application, it is known in advance that the plaintext messages will be of exactly 512 bytes. Taking advantage of this fact, we can optimize the implementation of the HCH mode of operation even further as shown in the algorithm of Figure 5.6(a). The modification implies a total saving of 11 clock cycles as is shown in Figure 5.6(b). The initial encryption of R is omitted since this parameter is substituted by a second key, while Q is substituted by R α. This modification also implies a saving in area resources as it will be seen in the next Section.

We do not present the details of the other designs as they follow the same strategy, but in Table 5.2 we show the extra hardware resources required by each architecture of the TES modes of operation. Besides an obvious impact in the area complexity of the modes, the resources occupied by each TES tend to increase its critical path, a fact that will become more explicit from the experimental results presented in the next Section.

5.6 Results

In this Section we present the experimental results obtained from our implementations. We measure the performances of our designs based on the following criteria: *total time* required for encrypting 32 blocks of data; *latency*, *i.e.* the time needed to produce the first output block; the size of the circuit in slices; the number of B-RAMs used and the *throughput*. The rest of this Section is organized as follows. In Subsection 5.6.1 we report the area and timing performance obtained in the implementation of the main building blocks required for the TES modes. Then, Subsection 5.6.2 reports the performance achieved by the TES modes analyzed in this Chapter. The comparison was carried out using a fully pipelined AES core, a sequential AES core and a pipelined encryption-only AES core.

5.6.1 Main Building Blocks

Table 5.4: Performance of the AES and Hash implementations

Method	Slices	B-RAM	Frequency (MHz)	Clock Cycles	Throughput (Gbps)
Full-core AES-Sequential	1301	18	81.97	10	1.05
Full-core AES-Pipeline	6368	85	83.88	1	10.74
Encryption-Only AES-Pipeline	2468	85	149.00	1	19.07
Hash function	4014	-	101.45	1	12.99

The two building blocks shown in Table 5.1, were used for implementing a full AES core (*i.e.* a key generation/encryption/decryption core) in sequential and pipelined architecture;

an encryption-only pipeline AES core and a hash function for the HCH, HCTR, TET, XCB and HEH modes. Table 5.4 summarizes the performance obtained in the implementation of those blocks. Note that the sequential AES core gives significantly poor throughput, while the hash function has better throughput than the full AES-pipeline but smaller throughput than the encryption-only AES core.

5.6.2 Performance Comparison of the Six TES Modes

In Table 5.5 the experimental results for the six modes of operation implemented with an underlying full pipelined AES core are shown. Note that the number of clock-cycles reported in Table 5.5 are one more than those estimated in Section 5.3. This is because in our actual implementation, one clock cycle is wasted on the initial reset operation. The critical path of the designs shown in Table 5.5 is mainly determined by the AES core, which as it was shown in Table 5.4, has a longer path than the hash function utilized in all six HCTR, HCH, HEH, TET,HEH and XCB modes.

From Table 5.5 it is evident that EME is the most economical mode in terms of area resources, mainly due to the fact that this mode does not utilize a hash function. Hence, the critical path of EME is given by the AES full core plus a chain of three layers of additions. Out of the six modes analyzed, HEH is both the fastest and the mode that shows lowest latency to produce the first block. XCB is the second most expensive mode in terms of hardware resource utilization, and the slowest. HCH requires more hardware resources than HCTR. TET is slower than HCH even assuming that its parameter σ^{-1} has been previously precomputed.

In terms of speed, the fastest mode is HEH since it only utilizes one AES block cipher call in sequential mode that can be computed concurrently with the hash function computation, whereas HCTR requires one AES call in the single mode which cannot be parallelized and HCH requires a total of four such calls (although only three have consequences in terms of clock cycles since the fourth one is masked with the computation of the hash function). Under this scenario, ordered from the fastest to the slowest we have, HEH, HCTR, HCHfp, EME, HCH, TET and XCB.

In Table 5.6 we show the six TES modes of operation when using a sequential implementation of the AES core. In a sequential architecture, EME is the slowest mode in terms of latency due to the two costly block cipher passes that requires eleven clock cycles per block. Hence, a significant increment in the total number of clock cycles is observed for the EME mode. This situation does not occur in the other modes since they only need one encryption pass. The hash function computation is not affected in this scenario due to the fact that we use a fully-parallel multiplier block able to produce a result in one clock cycle. Using a sequential AES core the best throughput is obtained by HCTR and HEH in that order.

While performing a sector encryption, all modes considered here, except for XCB, require

Table 5.5: Hardware costs of the modes with an underlying full 10-stage pipelined 128-bit AES core when processing one sector of 32 AES blocks: Virtex 4 Implementation

Mode	**Slices**	**B-RAM**	**Frequency** (MHz)	**Clock Cycles**	**Time** (μS)	**Latency** (μS)	**Throughput** GBits/Sec
HCTR	12068	85	79.65	89	1.117	0.703	**3.665**
HCH	13622	85	65.94	107	1.623	0.801	2.524
HCHfp	12970	85	66.50	96	1.443	0.990	2.837
XCB	13418	85	54.02	116	2.147	1.114	1.907
EME	10120	87	67.84	107	1.577	1.123	2.597
TET	12072	87	60.51	111	1.834	1.301	2.232
HEH	11545	85	72.44	75	1.035	0.591	**3.956**

Table 5.6: Hardware costs of the modes with an underlying sequential 128-bit AES core when processing one sector of 32 AES blocks: Virtex 4 Implementation

Mode	**Slices**	**B-RAM**	**Frequency** (MHz)	**Clock Cycles**	**Time** (μS)	**Throughput** (Gbits/sec)
HCTR	6000	18	76.46	398	5.205	**0.786**
HCH	6805	18	63.36	416	6.565	0.624
HCHfp	6481	18	63.84	405	6.339	0.645
XCB	6706	18	51.86	455	8.773	0.466
EME	2958	20	80.00	716	8.950	0.457
TET	6518	20	58.08	421	7.248	0.565
HEH	6128	18	69.36	384	5.536	**0.739**

Table 5.7: Hardware costs of the HCH, HCTR, EME, TET and HEH modes with an underlying encryption-only 10-stage pipelined 128-bit AES core when processing one sector of 32 AES blocks: Virtex 4 Implementation

Mode	**Slices**	**B-RAM**	**Frequency** (MHz)	**Clock Cycles**	**Time** (μS)	**Latency** (μS)	**Throughput** GBits/Sec
HCTR	6996	85	98.58	89	0.902	0.557	4.540
HCHfp	7513	85	95.80	98	1.022	0.678	4.000
EME	4200	87	149.09	107	0.717	0.496	**5.710**
TET	7165	87	93.04	111	1.193	0.849	3.430
HEH	7494	85	93.70	75	0.800	0.458	**5.117**

AES calls in encryption mode only. Hence, we just need an encryption-only AES core for performing a sector encryption in those modes. It was this observation that motivated us to investigate the performance of all modes except XCB using an encryption-only AES underlying block cipher. The results of this experiment are summarized in Table 5.7. The fastest throughput in this scenario is achieved by EME (the only mode that does not use a Karatsuba multiplier). In fact, in this case EME essentially achieves the same maximum clock frequency as that of the encryption-only AES core (see Table 5.4). EME is closely followed by HEH, and TET turns out to be the slowest.

Although our designs were optimized for their implementation on the Xilinx Virtex 4 device xc4vlx100-12ff1148, in order to provide the reader with a better comparison spectrum we performed additional simulations on some other FPGA families such as the state-of-the-art Xilinx Virtex 5 and cheaper FPGA families like Virtex 2 pro and Spartan 3E. The obtained place-and-route simulation results are summarized in Table 5.8. [4]

Let us recall that Tables 5.5-5.7 present the Virtex 4 implementation results achieved by the TES modes when using a full pipelined, sequential and encryption-only AES core designs, respectively. The first three portions of Table 5.8 show the performances obtained by those three designs when implemented on theds among the modes already observed in the Virtex 4 and Virtex 5 implementations. On the other hand, in the case of the encryption-only AES core design, HCTR outperformed EME in the Virtex 5 implementation, whereas EME was the clear winner in the Virtex 4 device simulation (see Table 5.7). Finally we implemented the two fastest TES under the sequential-core scenario in the Spartan 3E device xc3s1600e-5fg484.

5.7 Discussions

As we stated in Section 5.1, the design objective was to match the data rates of modern day disk controllers which are of the order of 3Gbits/sec. Table 5.6 shows that using a sequential design it is not possible to achieve such data rates, although this strategy provides more compact designs. If we are interested in encrypting hard disks of desktop or laptop computers the area constraint is not that high, but speed would be the main concern. So, a pipelined AES will probably be the best choice for designing disk encryption schemes.

From Table 5.5 we see that while using an encryption/decryption pipeline AES core the most efficient mode in terms of speed is HEH followed by HCTR, HCHfp, EME, HCH, TET and XCB. Thus we can conclude based on the experiments in this Chapter that HEH and HCTR are the best modes to use for this application.

[4]We note that the architecture of Xilinx Virtex 5 is substantially different than the ones of previous generations. Specifically, each Virtex-5 slice contains four 6-input 2-output LUTs and four flip-flops, whereas slices in previous families have two 4-input 1-output LUTs and two flip-flops.

Table 5.8: Performance of the modes in other FPGA families of devices

	Mode	Slices	B-RAM	Frequency (MHz)	Clock Cycles	Time (μS)	Latency (μS)	Throughput GBits/Sec
Full 10-Stage pipelined AES-128 core in Virtex 5	HCTR	5865	85	107.93	89	0.824	0.519	**4.965**
	HCH	5667	85	91.24	107	1.172	0.832	3.452
	HCHfp	5640	85	92.38	96	1.039	0.692	3.941
	XCB	5854	85	92.69	116	1.251	0.647	3.272
	EME	4518	87	90.19	107	1.186	0.831	3.450
	TET	5554	87	89.91	111	1.234	0.878	3.317
	HEH	5139	85	89.65	75	0.836	0.479	**4.896**
Sequential AES in Virtex 5	HCTR	2735	18	103.75	398	3.836		**1.06**
	HCH	2986	18	87.59	416	4.749		0.862
	HCHfp	2976	18	88.68	405	4.566		0.896
	XCB	3057	18	88.97	455	5.114		0.800
	EME	1274	20	107.00	716	6.691		0.612
	TET	2968	20	86.50	421	4.867		0.841
	HEH	2940	18	86.07	384	4.461		**0.918**
Encryption only AES-128 in Virtex 5	HCTR	3192	85	147.44	89	0.603	0.379	**6.785**
	HCHfp	3324	85	118.30	96	0.811	0.642	5.047
	EME	2005	87	173.80	107	0.615	0.431	**6.653**
	TET	2979	87	125.30	111	8.885	0.630	4.623
	HEH	3695	85	120.54	75	0.622	0.356	6.583
Full 10-Stage pipelined AES-128 core in Virtex 2 pro	HCTR	10946	85	73.06	89	1.218	0.752	**3.360**
	HCH	11718	85	50.30	107	2.127	1.490	1.920
	HCHfp	11227	85	63.52	96	1.511	1.180	2.710
	XCB	11685	85	50.11	116	2.315	1.177	1.769
	EME	9423	87	62.57	107	1.710	1.182	2.390
	TET	11163	87	50.58	111	2.194	1.540	1.860
	HEH	11539	85	72.44	85	1.173	0.579	**3.490**
Sequential AES-128 core in Spartan 3E	HCTR	5783	18	32.05	398	12.410		0.330
	HEH	6254	18	30.72	384	12.500		0.327

In the case of the Virtex 5 family of devices, authors in [17] reported a fast and efficient AES design with an associated critical path smaller than the one corresponding to the Karatsuba multiplier used in this work. Since EME is the only TES mode that does not require a field multiplier block, one can conclude that if that AES design and technology is adopted, then EME will probably emerge as the fastest of the TES modes of operation studied here.

From Table 5.7 we see that the encryption operation of all the modes considered here except XCB can be significantly improved if an encryption only AES core is implemented. So, in certain scenarios, it may be possible to have two different circuits for encryption and decryption where the encryption operation would be considerably faster. For the disk encryption scenario, it is probable that a sector would be written once and would be read many times. So it is better to have a faster decryption circuit, as the decryption operation is likely to be performed more frequently.

Since the TES are length preserving encryption schemes, *i.e.* they are permutations, one can process data by encrypting-then-decrypting or by decrypting-then-encrypting without affecting the security guarantees provided by the modes. However, from the practical perspective, this subtle change can improve the total throughput of a disk-encryption considerably. If an encryption only AES core is used then EME gives the best throughput and other modes are far behind it.

In this Chapter we presented the implementation on reconfigurable hardware on some TES, the efficiency of each of them were carefully analyzed and an specific architecture for HCH was explained in detail. These implementations would serve as the starting point and provide us with the baseline data for comparisons with other interesting designs that we provide in the following three Chapters.

Chapter 6

Efficient Implementations of BRW Polynomials

The walls are the publishers of the poor.

Eduardo Galeano

Polynomial hashes formed an important part of most of the modes that we implemented in the previous Chapter. In most modes discussed before we require the computation of an univariate polynomial of degree $m - 1$ defined over a finite field $\mathbb{F}_q$ as,

$$\mathsf{Poly}_h(X) = x_1 h^{m-1} + x_2 h^{m-2} + \cdots + x_{m-1} h + x_m, \tag{6.1}$$

where $X = (x_1, \ldots, x_m) \in \mathbb{F}_q^m$ and $h \in \mathbb{F}_q$. Traditionally, the evaluation of $\mathsf{Poly}_h(X)$ has been done using Horner's rule, which requires $(m - 1)$ multiplications and $m - 1$ additions in $\mathbb{F}_q$. In the rest of this Chapter, we will refer to $\mathsf{Poly}_h()$ as a normal polynomial.

Bernstein [10] introduced a new class of polynomials which were later named in [125] as Bernstein-Rabin-Winograd (BRW) polynomials. BRW polynomials on m message blocks defined over $\mathbb{F}_q$ have the interesting property that they can be used to provide authentication, but, unlike the normal polynomial they can be evaluated using only $\lfloor \frac{m}{2} \rfloor$ multiplications in $\mathbb{F}_q$ and $\lceil \log_2 m \rceil$ squarings. Thus, these polynomials potentially offer a computational advantage over the normal ones. Further, in [125] BRW polynomials were used to construct new TESs named HEH[BRW] and HMCH[BRW]. Our focus would be to develop hardware architectures for these modes.

The use of BRW polynomials in hardware has not been addressed till date. As will be clear from discussions later, the structure of a BRW polynomial is fundamentally different from the normal ones, and there are some subtleties associated to their efficient implementation that are worthy of further analysis. In particular, the recursive definition of a BRW polynomial gives it a certain structure which is amenable to parallelization. It turns out that to take advantage of this parallel structure one needs to carefully schedule the order of multiplications involved in the polynomial evaluation. The scheduling is determined by the dependencies in the multiplications and also by the desired level of parallelization and

hardware resources available.

In this Chapter we present a hardware architecture for efficient evaluation of BRW polynomials. The hardware design heavily depends on the careful analysis of the inherent parallelism in the structure of a BRW polynomial. This leads to a method to determine the order in which the different multiplications are to be performed. We present an algorithm that schedules in an efficient fashion, all the $\lfloor \frac{m}{2} \rfloor$ multiplications required for the evaluation of a BRW polynomial keeping in mind the amount of parallelism desired. This algorithm leads to a hardware architecture that can perform an optimal computation of BRW polynomials in the sense that the evaluation is achieved using a minimum number of clock cycles. This analysis and architecture would be used to construct architectures for the modes HEH[BRW] and HMCH[BRW], we discuss these architectures separately in the next Chapter.

From the point of view of hardware realizations, the most crucial building block of a polynomial hash function is a field multiplier. Digit-serial multipliers yield compact designs in terms of area and enjoy short critical paths but they require several clock cycles in order to compute a single field multiplication. In contrast, fully-parallel multipliers as used in the previous Chapter are able to compute one field multiplication every clock cycle. However, due to their large critical path, these multipliers seriously compromise the design's maximum achievable clock frequency.

Since polynomial hash blocks require the batch computation of a relatively large number of products, it makes sense to utilize pipelined multiplier architectures. In this Chapter, we decided to utilize a k-stage pipeline multiplier with $k = 2, 3$. After a latency period required to fill up the pipe, these architectures are able to perform one field multiplication every clock cycle. The advantage is a much shorter critical path than the one associated with fully parallel multiplier schemes [11]. This change in the multiplier is a major deviation from the design philosophy adopted in the previous Chapter. Also, the choice of the pipelined multiplier opens up the interesting problem of efficient scheduling to reduce pipeline delays. We adequately address this scheduling problem for BRW polynomials in this Chapter.

The organization of the rest of the Chapter is as follows. In Section 6.1, we define the BRW polynomials and present a tree based analysis of such polynomials. Using the tree structure of the BRW polynomials we develop a scheduling algorithm and provide analysis of the scheduling algorithm. Finally, based on the scheduling algorithm we present the hardware architecture for computing BRW polynomials. In Section 6.6 we provide implementation details of the hardware architecture used for evaluating a BRW polynomial. We do not provide experimental results in this Chapter. The results using BRW polynomials are presented in Chapter 7. A large part of the discussions in this Chapter appear in [27].

6.1 BRW Polynomials

A special class of polynomials was introduced in [10] for fast polynomial hashing and subsequent use in message authentication codes. In [10] the origin of these polynomials were traced back to Rabin and Winograd [113], but the construction presented in [10] has subtle differences compared to the construction in [113]. The modifications were made keeping an eye to the issue of computational efficiency. Later in [125] these polynomials were used in the construction of tweakable enciphering schemes and the class of polynomials were named as Bernstein-Rabin-Winograd (BRW) polynomials.

Let $X_1, X_2, \ldots, X_m, h \in \mathbb{F}_q$, then the BRW polynomial $H_h(X_1, \ldots, X_m)$ is defined recursively as follows.

- $H_h() = 0$
- $H_h(X_1) = X_1$
- $H_h(X_1, X_2) = X_2 h + X_1$
- $H_h(X_1, X_2, X_3) = (h + X_1)(h^2 + X_2) + X_3$
- $H_h(X_1, X_2, \ldots, X_m) = H_h(X_1, \ldots, X_{t-1})(h^t + X_t) + H_h(X_{t+1}, \ldots, X_m)$, if $t \in \{4, 8, 16, 32, \ldots\}$ and $t \leq m < 2t$.

Note, the additions and multiplications are all in $\mathbb{F}_q$

Computationally the most important property is that for $m \geq 2$, $H_h(X_1, \ldots, X_m)$ can be computed using $\lfloor m/2 \rfloor$ multiplications and $\lceil \lg m \rceil$ squarings. In the rest of the Chapter, we will use either $H_h()$ or $\mathsf{BRW}_h()$ to denote a BRW polynomial.

Our objective here is to device a way to compute $H_h(X_1, \ldots, X_m)$ efficiently in hardware. An important issue we address here is how one can exploit the parallelism inherent in the definition of the polynomial to enable construction of a circuit for computing $H_h(X_1, \ldots, X_m)$ using a pipelined field multiplier for any (but fixed) values of m. For constructing such a circuit we need to identify the multiplications that can be performed in parallel and thus devising a method to schedule the multiplications in such a manner that the delay in the pipeline is minimized. To design such a scheduling method we view a given BRW polynomial as a tree which enables us to identify the dependencies in the various multiplications involved and thus schedule the multiplications appropriately with the aim of keeping the pipeline always full. The procedure is described in the next section.

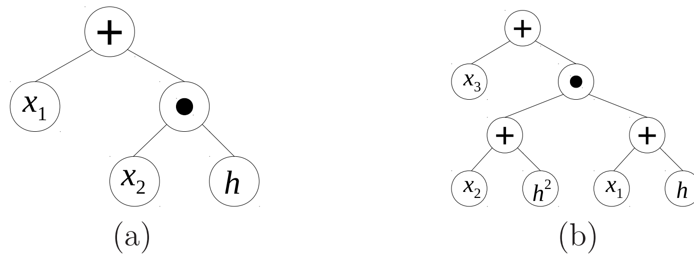


Figure 6.1: Trees corresponding to $m = 2, 3$. The nodes labeled with $\odot$ and $\oplus$ represent a multiplication node and an addition node respectively. (a) Tree corresponding to $H_h(X_1, X_2)$. (b) Tree corresponding to $H_h(X_1, X_2, X_3)$.

6.2 A Tree Based Analysis

A BRW polynomial $H_h(X_1, \ldots, X_m)$ can be represented as a tree T_m which contains three types of nodes, namely, *multiplication nodes*, *addition nodes* and *leaf nodes*. The tree T_m will be called a BRW tree and can be recursively constructed using the following rules:

1. For $m = 2, 3$ it is easy to construct T_m directly as shown in Figure 6.1.
2. If $m = 2^s$, for some $s \geq 2$, the root of T_m is a multiplication node. The left subtree of the root consists of a single addition node which in turn has the leaf nodes h^m and X_m as its left and right child, respectively. The right subtree of the root is the tree T_{m-1}.
3. If $2^s < m < 2^{s+1}$ for some $s \geq 2$, the root is an addition node with its left subtree as T_{2^s} and the right subtree as T_{m-2^s}.

A construction of the BRW tree T_{16} corresponding to the polynomial $H_h(X_1, \ldots, X_{16})$ is shown in Figure 6.2. According to this construction, the following two properties hold.

- Any leaf node is either a message block X_j or it is h^k, for some j, k.
- For a multiplication node, either, its left child is labeled by a message block X_j and the right child is labeled by h; or, its left child is an addition node which in turn has a message block X_j and h^k as its children for some j and k. As a consequence, for a multiplication node, there is exactly one leaf node in its left subtree which is labeled by a message block.

As we are only interested in multiplications, we can ignore the addition nodes and thus simplify the BRW tree by deleting the addition nodes from it. We shall address the issue of addition later when we describe our specific design in Section 6.6, and we would then see that ignoring the additions as we do now will not have any significant consequences

from the efficient implementation perspective. We reduce the tree T_m corresponding to the polynomial $H_h(X_1, \ldots, X_m)$ to a new tree by applying the following steps in sequence.

1. Label each multiplication node v by j where X_j is the leaf node of the left subtree rooted at v.
2. Remove all nodes and edges in the tree T_m other than the multiplication nodes.
3. If u and v are two multiplication nodes, then add an edge between u and v if u is the most recent ancestor of v in T_m.

The procedure above will delete all the addition nodes from the tree T_m. We shall call the resulting structure a *collapsed forest* (as the new structure may not be always connected, but its connected components would be trees) and denote it by F_m. Note that for every m, there is a unique BRW tree T_m and hence a unique collapsed forest F_m. The collapsed forests corresponding to polynomials $H_h(X_1, \ldots, X_{16})$ and $H_h(X_1, \ldots, X_{30})$ are shown in Figure 6.3.

By construction, the number of nodes in a collapsed forest F_m is equal to the number of multiplication nodes in T_m. The nodes of F_m are labeled with integers. Label j of a node in F_m signifies that either the multiplicands are X_j and h; or, one of the multiplicands is $(X_j + h^k)$ for some k. As a result, there is a unique multiplication associated with each node of a collapsed forest.

For example, the multiplication $(X_2 + h^2) * (X_1 + h)$ is associated to the node labeled *2* in Figure 6.3. Refer to Figure 6.1 to see this. Similarly, if the outputs of nodes labeled *4* and *6* are A and B respectively, then the multiplication associated with the node labeled *8* is $(X_8 + h^8) * (A + B + X_7)$.

This procedure easily generalizes and it is possible to explicitly write down the unique multiplication associated with any node of a collapsed forest. So, the problem of scheduling the multiplication in T_m reduces to obtaining an appropriate sequencing (linear ordering) of the nodes of F_m.

The structure of the collapsed forest corresponding to a polynomial $H_h(\cdot)$ helps us to visualize the dependencies of the various multiplications involved in the computation of $H_h(\cdot)$. The following definitions would help us to characterize dependencies among those operations.

Definition 6.1. *Let v be a node in a collapsed forest F, the level of v in F denoted by* $\mathsf{level}_F(v)$ *is the number of nodes present in the longest path from v to a leaf node. A node v in F such that* $\mathsf{level}_F(v) = 0$ *is said to be* independent. *Any node v with* $\mathsf{level}_F(v) > 0$ *is said to be* dependent.

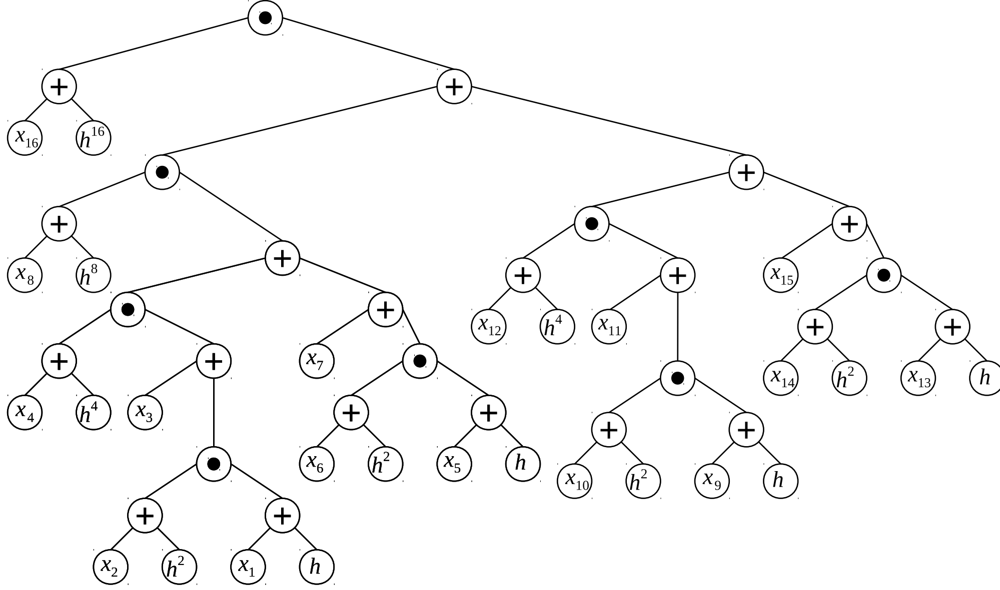


Figure 6.2: The BRW tree representing $H_h(X_1, \ldots, X_{16})$.

Definition 6.2. *Suppose u, v are nodes in a collapsed forest F such that $\mathsf{level}_F(u) > \mathsf{level}_F(v)$ and u is an ancestor of v in F, then we say that u is dependent on v.*

In the following proposition, we state some important properties of collapsed forests.

Proposition 6.1. *Let F_m be a collapsed forest corresponding to the BRW polynomial $H_h(X_1, \ldots, X_m)$.*

1. *The number of nodes in F_m is $\lfloor \frac{m}{2} \rfloor$.*
2. *The nodes in F_m are labeled by integers $2i$, $1 \leq i \leq \lfloor \frac{m}{2} \rfloor$.*
3. *If m is even then F_m and F_{m+1} are same.*
4. *The number of connected components in F_m is equal to the Hamming weight of $\lfloor \frac{m}{2} \rfloor$.*
5. *Let $p = \lfloor m/2 \rfloor$ and $\mathsf{bit}_i(p)$ denote the ith bit of p where $0 \leq i \leq \mathsf{len}(p)$. If $\mathsf{bit}_i(p) = 1$ then F_m contains a tree of size 2^i.*
6. *If x is a label of a node and $x \equiv 2 \pmod 4$ then the node is an independent node.*
7. *If x is a label of a node and $x \equiv 0 \pmod 8$ then x has at least $x - 2$ and $x - 4$ as its children.*
8. *If x is the label of a node and $x \equiv 4 \pmod 8$, then $x - 2$ is the only child of x.*

Proof. (1)-(3) directly follows from the definition of the BRW polynomial and the construction of the collapsed forest.

Proof of 4: Observe that for $m > 3$,

$$H_h(X_1, \ldots, X_m) = H_h(X_1, \ldots, X_{t-1})(h^t + X_t) + H_h(X_{t+1}, \ldots, X_m),$$

where t is a power of 2 and $t \leq m < 2t$. Let F_m be the collapsed forest corresponding to $H_h(.)$, then by construction F_m would have one connected component corresponding to the polynomial $H_h(X_1, \ldots, X_{t-1})(h^t + X_t)$, in which the node labeled with t would be the root, and would have other components corresponding to $H_h(X_{t+1}, \ldots, X_m)$. Thus, if C_m denotes the number of connected components of F_m, we have the following recurrence for C_m

$$C_m = 1 + C_{m-t} \text{ for } m > 3$$

and by inspection we have $C_0 = C_1 = 0$, $C_2 = C_3 = 1$. Now, let m be an $(\ell + 1)$-bit number and $m = m_0 + m_1 \cdot 2 + \ldots + m_\ell \cdot 2^\ell$, where $m_i \in \{0, 1\}$ for $0 \leq i \leq \ell - 1$ and $m_\ell = 1$. Then the above recurrence becomes

$$\begin{aligned} C_m &= 1 + C_{m-2^\ell} \\ &= 1 + 1 + C_{m-2^l-2^k} \text{ where } k \text{ is the largest integer smaller than } \ell \text{ such that } m_k = 1 \\ &= \underbrace{1 + 1 + \ldots + 1}_{\mu \text{ times}} + C_j, \text{ where } j \in \{1, 2, 3\},\ \mu = \sum_{i=2}^{\ell} m_i \end{aligned}$$

This along with the initial conditions proves (4).

Proof of 5: Following the same arguments as above we see that number of connected components in F_m is equal to $\sum_{i=1}^{\ell} m_i$, which is the Hamming weight of $\lfloor m/2 \rfloor$. For each $m_i = 1$ $(i > 0)$ we have a connected component which is associated with the polynomial

$$p_i = H_h(X_{J+1}, \ldots, X_{J+2^i-1})(X_{J+2^i} + h^{2^i}),$$

where $J = m - \sum_{j=0}^{i} 2^j m_j$. By (1), the tree corresponding to p_i contains 2^{i-1} nodes. This proves (5).

Proofs of (6), (7) and (8) are by induction on m and is based on the structure of F_m. We provide the proof of (6). The proofs of (7) and (8) are similar.

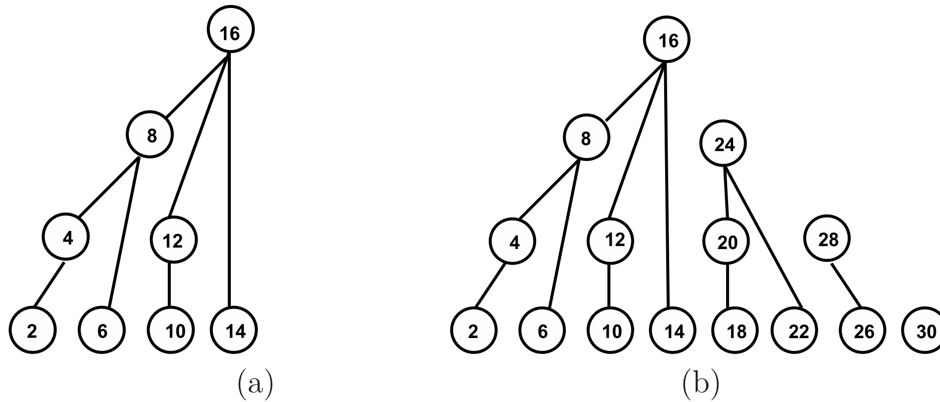


Figure 6.3: (a) Collapsed forest corresponding to $H_h(X_1, \ldots X_{16})$. (b) Collapsed forest corresponding to $H_h(X_1, \ldots X_{30})$.

Proof of 6: First we prove by induction that (6) is true for any F_m where $m = 2^i$. The cases for $i = 1, \ldots, 4$ can be easily verified from Figure 6.3(a). Assuming (6) to be true for $m = 2^i$ ($i \geq 4$) we will show it to be true for $m = 2^{i+1}$. Following the construction of the collapsed forest, $F_{2^{i+1}}$ can be constructed from F_{2^i} as follows. Construct a new tree F' from F_{2^i} by adding 2^i to the label of each node of F_{2^i}. Join F_{2^i} and F' by making the root of F_{2^i} the rightmost child of the root of F'. The new tree thus constructed is $F_{2^{i+1}}$. The nodes which were independent in F_{2^i} and F' remain so in $F_{2^{i+1}}$ also the only labels which were congruent to 2 modulo 4 in F_{2^i} remains so in F'. Hence (6) is true for $m = 2^{i+1}$.

To prove for any m we use the following observations which follow from (4) and the construction of the collapsed forest.

- F_m, for any m, consists of trees, where the number of nodes in each tree is a distinct power of 2.

- The component trees occur in descending order of sizes from left to right.

- If the size of the jth tree is 2^i and S_j be the set of the sizes of trees preceding the jth tree then the jth tree is constructed by adding $J = 2\sum_{x \in S_j} x$ to each label of F_{2^i}. So, for $j \geq 0$, $J \equiv 0 \bmod 4$.

The proofs of (7) and (8) are similar to the proof of (6).

□

6.3 Scheduling of Multiplications

Our goal, as stated earlier, is to design a circuit for computing BRW polynomials using a pipelined multiplier. If we use a pipelined multiplier with N stages, then N clock cycles would be required to complete one multiplication, but in each clock cycle N different multiplications can be processed, as long as these N multiplications happen to be independent of each other, i.e., none of these N multiplications should depend on the results of the others. Thus, if it can be guaranteed that N independent multiplications are available in each clock then the circuit will require $m + N$ clock cycles to complete m multiplications (there would be an initial latency of N clocks for filling the pipe and thereafter the result of one multiplication would be produced in each subsequent clock cycle).

A collapsed forest is a convenient way to view the dependencies among the various multiplications which are required to compute a BRW polynomial. In this section, we propose an algorithm Schedule which uses a collapsed forest to output a multiplication schedule. The aim of the algorithm is to minimize the number of clock cycles.

For designing the scheduling algorithm we require two lists L_1 and L_2. For a list L and an element x of L, we shall require the following operations.

1. Pop(L): returns the first element in L; or, returns NULL if L is empty.
2. Delete(L): deletes the first element in L.
3. Insert(x, L): inserts x in L and x becomes the last element in L.

Note that Pop(L) does not delete the first element from L. Two successive pop operations from L without any intermediate delete operation will result in the same element.

Each node in the collapsed forest is given two fields NC and ST associated with it. If x is a node in the collapsed forest then x.NC represents the number of children of node x, and x.ST denotes the time at which the node x was inserted into the list L_2 (the requirement of ST will become evident soon). Let Parent(x) denote the parent of node x in the collapsed forest.

The algorithm for scheduling is described in Figure 6.4. The algorithm uses a function Process which is also depicted in Figure 6.4. The inputs to the algorithm are m and a variable NS which represents the number of pipeline stages. The outputs from Step 103 of

Process form a sequence of integers. This provides the desired sequence of multiplications.

Before the main while loop begins (in line 11) the list L_1 contains all the independent nodes in the collapsed forest corresponding to the given polynomial and L_2 is empty. Within the while loop no nodes are inserted in L_1, but new nodes are inserted into and get deleted from L_2. L_2 is a queue, i.e., the nodes get deleted from L_2 in the same order as they enter it. The way we define the operations Pop(),Delete() and Insert() guarantee this.

At any given clock-cycle, the nodes in the forest can be in four possible states: *unready*, *ready*, *scheduled* and *completed*. A node x is unready if there exist a node y on which x is dependent but y has not been completed yet. A node becomes ready if all nodes on which it depends are completed. A node can only be scheduled after it is ready. Once a node is scheduled it takes NS clock cycles to get completed.

In the beginning, the nodes with level zero, i.e., the independent nodes are the only nodes in the ready state all others being in the unready state. These independent nodes are listed in L_1 at the beginning, no more nodes are further added to L_1. Thus, the nodes in L_1 can be scheduled at any time. As the algorithm proceeds, nodes get scheduled in line 103 of the function Process.

After a node is scheduled the algorithm updates the field NC (number of children) of its parent. When the last child of a given node x is scheduled then x is inserted into the list L_2, and in the field ST of x a record of the time when its last child was scheduled is kept.

If a node is in L_2 then it is sure that all its children have been scheduled but not necessarily completed. The condition in line 12 checks if the last child of a given node in L_2 has already been completed and if a node x passes this check then it is ready to be scheduled.

For each execution of the while loop (lines 10 to 20) at most one node gets scheduled and once a node is scheduled it is deleted from the corresponding list. The condition on the while loop (line 10) checks whether both the lists are empty and the condition on line 12 checks whether the first element of L_2 is ready, in the next two propositions we state why these checks would be sufficient.

Proposition 6.2. *If L_1 and L_2 are both empty then there are no nodes left to be scheduled. Further, the algorithm terminates, i.e., the condition that L_1 and L_2 are both empty is eventually attained.*

Proof. Suppose both L_1 and L_2 are empty but there is a node v which is left to be scheduled.

```
Algorithm Schedule(m,NS)
1.     Construct the collapsed forest F_m;
2.     for each node x in F_m
3.         x.NC ← number of children of x;
4.         x.ST ← undefined;
5.         if level_{F_m}(x) = 0,
6.             Insert(x, L_1);
7.     end for
8.     L_2 ← Empty;
9.     clock ← 1;
10.    while (L_1 and L_2 are both not empty)
11.        x ← Pop(L_2);
12.        if (x ≠ NULL and clock − x.ST > NS)
13.            Process(x, L_2, clock);
14.        else
15.            x ← Pop(L_1);
16.            if (x ≠ NULL))
17.                Process(x, L_1, clock);
18.        end if;
19.        clock ← clock + 1;
20.    end while

Function Process(x, L,clock)
101.   Delete(L);
102.   y ← Parent(x);
103.   Output x;
104.   if y ≠ NULL
105.       y.NC ← y.NC − 1;
106.       if (y.NC = 0)
107.           y.ST = clock;
108.           Insert(y, L_2);
109.       end if;
110.   end if;
111.   return
```

Figure 6.4: The algorithm Schedule

As L_1 contains all independent nodes in the beginning and it is empty thus v is not an independent node. As v has not been scheduled and it is not in L_2 thus there must be a child of v which has not been scheduled. As there must exist a path from v to some independent node x, applying the same argument repeatedly we would conclude that there exist some independent node x which has not been scheduled. This give rise to a contradiction as L_1 is empty.

For the second statement, note that as long as L_1 is non-empty, each iteration of the while loop results in exactly one node of F_m been added to the schedule. This node is either a node in L_2 (if there is one such node), or, it is a node of L_1.

Once L_1 becomes empty, if L_2 is also empty, then by the first part, the scheduling is complete. If L_2 is non-empty, then let v be the first element of L_2. It may be possible that an iteration of the while loop does not add a node to the existing schedule. This happens if $\text{clock} - v.\mathsf{ST} \leq \mathsf{NS}$. But, the value of $v.\mathsf{ST}$ does not change while the value of clock increases. So, at some iteration, the condition $\text{clock} - v.\mathsf{ST} > \mathsf{NS}$ will be reached and the node v will be output as part of the call $\mathsf{Process}(v, L, \text{clock})$.

□

Proposition 6.3. *If the first element of L_2 is not ready to be scheduled then no other elements in L_2 would be ready.*

Proof. Let v be the first element in L_2, as v is not ready to be scheduled, hence $clock - v.\mathsf{ST} \leq \mathsf{NS}$. Let u be any other node in L_2, as u was added to L_2 later than v thus $u.\mathsf{ST} > v.\mathsf{ST}$ and so $\text{clock} - u.\mathsf{ST} < \text{clock} - v.\mathsf{ST} < \mathsf{NS}$. Thus, u is also not ready to be scheduled. □

In the next Subsection some examples about the running of the algorithm Schedule are provided.

6.3.1 Some examples on algorithm Schedule

We give an example of the running the algorithm for $m = 16$ and $\mathsf{NS} = 2$. The collapsed tree corresponding to the BRW polynomial $H_h(X_1, X_2, \ldots, X_{16})$ is shown in Figure 6.3(a). The independent nodes in the tree are *2*, *6*, *10*, *14* and according to line 6 of the algorithm Schedule these nodes are inserted in the list L_1, and initially L_2 is empty. The contents of the two lists along with the output in each clock is shown in Figure 6.5. The entries in the list L_2 are listed as $x(y)$, where x is the label of the node and $x.\mathsf{ST} = y$. Figure 6.5 shows that after clock 9 both the lists L_1 and L_2 become empty and thus the algorithm stops. There is no output produced in clock 8 as in clock 8 L_1 is empty and the only node in L_2 is not ready as its start time is 7, which means that its ultimate child got scheduled in clock 7

and thus is yet to be completed. The following sequence of nodes is produced as output of Schedule.

2, 6, 4, 10, 8, 12, 14, 16.

We describe the scheduling of multiplications corresponding to this sequence. Again refer to Figure 6.2.

M_1: $R_1 = (X_2 + h^2)(X_1 + h)$;
M_2: $R_2 = (X_6 + h^2)(X_5 + h)$;
M_3: $R_3 = (X_4 + h^4)(X_3 + R_1)$;
M_4: $R_4 = (X_{10} + h^2)(X_9 + h)$;
M_5: $R_5 = (X_8 + h^8)(R_3 + R_2 + X_7)$;
M_6: $R_6 = (X_{12} + h^4)(X_{11} + R_4)$;
M_7: $R_7 = (X_{14} + h^2)(X_{13} + h)$;
M_8: $R_8 = (X_{16} + h^{16})(R_5 + R_6 + R_7 + X_{15})$.

The 8 multiplications are $M_1, \ldots, M_8$. In this example, we have not tried to minimize the number of intermediate storage registers that are required. A method for doing this will be discussed later. Note the following points.

1. In each of the multiplications, the subscript of X in the first multiplicand is the label of the corresponding node in F_{16}.

2. The scheduling is compatible with $\mathsf{NS} = 2$, i.e., a 2-stage pipeline: M_3 and M_4 depend on the output of M_1 and so start 2 clocks after M_1 starts; M_5 depends on the output of M_2 and M_3 and starts 2 clocks after M_3; and so on.

The output of the algorithm Schedule for various number of blocks for $\mathsf{NS} = 2$ and 3 are shown in Tables 6.1 and 6.2. The entries $-$ in those Tables means that no multiplication was scheduled in the corresponding clock. The last column (total clocks) is the clock when the last multiplication was scheduled.

6.4 Optimal Scheduling

Given a BRW polynomial on m message blocks, the number of nodes in the corresponding collapsed tree is $p = \lfloor m/2 \rfloor$. The scheduling of these nodes is said to be *optimal* if one node can be scheduled in each clock-cycle thus requiring p clock-cycles to schedule all the nodes. If such a scheduling is possible for a given value of the number of stages (NS) we say that the scheduling admits a *full pipeline*, as such a scheduling will not give rise to any pipeline delays.

Blocks	Clock															Total
(m)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	clocks
2	2															1
4	2	–	4													3
6	2	6	4													3
8	2	6	4	–	8											5
10	2	6	4	10	8											5
12	2	6	4	10	8	12										6
14	2	6	4	10	8	12	14									7
16	2	6	4	10	8	12	14	–	16							9
18	2	6	4	10	8	12	14	18	16							9
20	2	6	4	10	8	12	14	18	16	20						10
22	2	6	4	10	8	12	14	18	16	20	22					11
24	2	6	4	10	8	12	14	18	16	20	22	–	24			13
26	2	6	4	10	8	12	14	18	16	20	22	26	24			13
28	2	6	4	10	8	12	14	18	16	20	22	26	24	28		14
30	2	6	4	10	8	12	14	18	16	20	22	26	24	28	30	15

Table 6.1: The output of Schedule for NS $= 2$ for small number of blocks

Blocks	Clock															Total
(m)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	clocks
2	2															1
4	2	–	–	4												4
6	2	6	–	4												4
8	2	6	–	4	–	–	8									7
10	2	6	10	4	–	–	8									7
12	2	6	10	4	–	12	8									7
14	2	6	10	4	14	12	8									7
16	2	6	10	4	14	12	8	–	–	16						10
18	2	6	10	4	14	12	8	18	–	16						10
20	2	6	10	4	14	12	8	18	–	16	20					11
22	2	6	10	4	14	12	8	18	22	16	20					11
24	2	6	10	4	14	12	8	18	22	16	20	–	–	24		14
26	2	6	10	4	14	12	8	18	22	16	20	26	–	24		14
28	2	6	10	4	14	12	8	18	22	16	20	26	–	24	28	15
30	2	6	10	4	14	12	8	18	22	16	20	26	30	24	28	15

Table 6.2: The output of Schedule for NS $= 3$ for small number of blocks

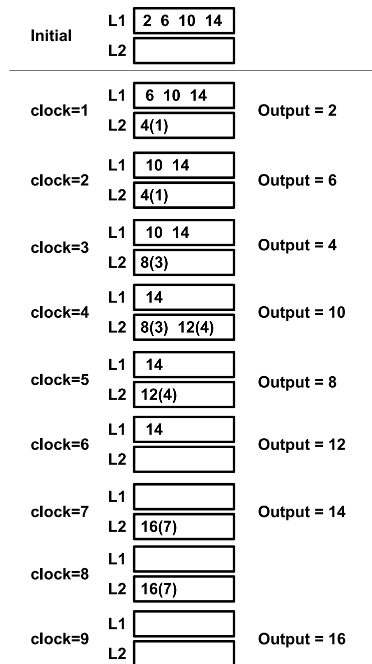


Figure 6.5: The states of the lists L_1 and L_2 when $\mathsf{Schedule}(16, 2)$ is run. The entries in the list are denoted as $x(y)$ where x is the label of a node and $y = x.\mathsf{ST}$

The above notion of optimality is a strong one and an optimal scheduling will not exist for all values of m and NS. Existence of an optimal scheduling for NS stages means that in each clock cycle NS independent nodes are available.

If m is a power of two then it is easy to see that the collapsed forest would contain a single tree and the root would be dependent on all other nodes (as is the case in Figure 6.3(a)), thus no scheduling procedure can yield an optimal scheduling for such an m for any $\mathsf{NS} > 1$.

Also, as the number of pipeline stages increases, for an optimal scheduling to be possible, more independent multiplications are required. For small values of NS, however, the following theorem gives the conditions for which $\mathsf{Schedule}$ gives an optimal scheduling for $\mathsf{NS} = 2$ and 3.

Theorem 6.1. *Let $H_h(X_1, X_2, \ldots, X_m)$ be a BRW polynomial and let $p = \lfloor m/2 \rfloor$ be the number of nodes in the corresponding collapsed forest. Let clks be the number of clock cycles taken by* $\mathsf{Schedule}$ *to schedule all nodes, then,*

1. *If $\mathsf{NS} = 2$, and $p \geq 3$, then*

$$clks = \begin{cases} p + 1 & \text{if } p \equiv 0 \pmod 4; \\ p & \text{otherwise.} \end{cases}$$

2. If $\mathsf{NS} = 3$ *and* $p \geq 7$*, then*

$$clks = \begin{cases} p+2 & \text{if } p \equiv 0 \pmod 4; \\ p+1 & \text{if } p \equiv 1 \pmod 4; \\ p+1 & \text{if } p \equiv 2 \pmod 4; \\ p & \text{if } p \equiv 3 \pmod 4. \end{cases}$$

Proof. Both the proofs are by induction. We present the proof only for $\mathsf{NS} = 2$ as the other case is similar. For $p = 3$ (i.e. $m = 6$) the explicit output of the algorithm is *2, 6, 4*, and it takes 3 clock cycles to schedule the three nodes, this proves that the base case is true. Suppose the results hold for some $p \geq 3$ and we wish to show the results for $p+1$. There are the following cases to consider:

1. $p+1 \equiv 1 \pmod 4$. Then $p \equiv 0 \pmod 4$, hence by induction hypothesis the p nodes were scheduled in $p+1$ cycles, signifying that there was one cycle when no node was scheduled. The last node in this case has label $2(p+1)$ and as $2(p+1) \equiv 2 \pmod 4$, hence the last node is an independent node (from Proposition 6.1), hence the last node can be scheduled in the missed cycle, thus the total clocks required for $p+1$ nodes would be $p+1$.

2. $p+1 \equiv 2 \pmod 4$. Then, $p \equiv 1 \pmod 4$, hence by induction hypothesis p nodes were scheduled in p cycles, the last node to be scheduled has label $2(p+1)$ and $2(p+1) \equiv 4 \pmod 8$ and hence by Proposition 6.1, has only one child and the label of the child is $2p$. Considering the previous case, $2p$ was not the last node to be scheduled; hence, the node $2(p+1)$ can be scheduled in the $p+1$-th cycle.

3. $p+1 \equiv 3 \pmod 4$. Then, $p \equiv 2 \pmod 4$, hence p nodes were scheduled in p cycles, the last node to be scheduled has label $2(p+1)$ and $2(p+1) \equiv 2 \pmod 4$ and hence by following the same arguments as in case 1 the nodes can be scheduled in $p+1$ cycles.

4. $p+1 \equiv 0 \pmod 4$. Then, $p \equiv 3 \pmod 4$, hence by induction hypothesis p nodes were scheduled in p cycles. The last node to be scheduled has label $2(p+1)$, and by Proposition 1 it would have nodes with labels $2p$ and $2(p-1)$ as its children. Considering cases 2 and 3 if p nodes are scheduled then the last node to be scheduled has label $2(p-1)$ which is a child of the node $2(p+1)$, hence the node $2(p+1)$ cannot be scheduled in the $p+1$-th cycle. Thus the number of cycles required would be $p+2$.

This completes the proof. □

From the proof above one can obtain a recursive description of the output of the scheduling algorithm for $\mathsf{NS} = 2$. Let $p \geq 4$, and $x_1, \ldots, x_p$ be the sequence for p, where $x_1, \ldots, x_p \in \{2, 4, \ldots, 2p\}$. Then, the following is the construction of the sequence for $p+1$:

If $p+1 \equiv 0 \pmod 2$ then output the sequence $x_1, \ldots, x_p, 2(p+1)$;

If $p+1 \equiv 3 \pmod 4$, then output the sequence $x_1, \ldots, x_p, 2(p+1)$;

If $p+1 \equiv 1 \pmod 4$, then output the sequence $x_1, \ldots, x_{p-1}, 2(p+1), x_p$.

Similarly if $\mathsf{NS} = 3$, and if $x_1, \ldots, x_p$ be the sequence for $p \geq 6$, then the following is the construction of the sequence for $p+1$:

If $p+1 \equiv 0 \pmod 2$, then output the sequence $x_1, \ldots, x_p, 2(p+1)$;

if $p+1 \equiv 1 \pmod 4$, then output the sequence $x_1, \ldots, x_{p-2}, x_{p-1}, 2(p+1), x_p$;

if $p+1 \equiv 3 \pmod 4$, then output the sequence $x_1, \ldots, x_{p-2}, 2(p+1), x_{p-1}, x_p$.

As stated, our definition of optimality is a strong one. It is possible to define optimality in a weaker sense as follows: Given a BRW polynomial and number of stages NS a scheduling of the multiplication nodes is called weakly optimal if it takes the minimum number of clock cycles among all possible schedules for the given polynomial and the given value of NS. Using this weaker definition of optimality it would be guaranteed that for any polynomial and any value of NS a optimal schedule will always exist. Moreover, if a schedule is optimal in the stronger sense that we formulated it would also be optimal in the weaker sense. Characterizing weak optimality seems to be combinatorially a difficult task and is not required for our case, as for our work, we mostly can show strong optimality or small deviations from it.

6.5 The Issue of Extra Storage

Optimizing the number of clock cycles should not be the only goal for a scheduling algorithm. An important resource associated with a pipelined architecture is the requirement of extra storages for storing the intermediate results. The issue of storage in the case of computing BRW polynomials is simple, we illustrate the issue with an example. Refer to the diagram of the collapsed tree in Figure 6.3(b), suppose for a two-stage pipeline we schedule the multiplications in the following order:

$$2,\ 6,\ 10,\ 14,\ 18,\ 22,\ 26,\ 30,\ 4,\ 12,\ 20,\ 28,\ 8,\ 24,\ 16 \tag{6.2}$$

This schedule requires 15 clock cycles and is thus optimal, but this is very different from the order of the multiplications given by the algorithm Schedule. This ordering, though it is optimal in the terms of number of clock cycles required, requires more extra storage for storing the intermediate results. Recall that the dependence of the nodes in the BRW tree shows that multiplication operation represented by a node x may be started when all its

children have been completed. In each clock cycle at most one multiplication gets completed, thus the intermediate results computed for the children of x have to be stored, as they will be required for the computing of x. If the scheduling is done as in Eq. (6.2) then the starting times and finishing times (in clocks) of the nodes would be as below.

Nodes	*2*	*6*	*10*	*14*	*18*	*22*	*26*	*30*	*4*	*12*	*20*	*28*	*8*	*24*	*16*
Starting Time:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Finishing Time	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Note that the results of the multiplications in nodes *2*, *10*, *18*, *26* which are completed in the clocks 3, 5, 7 and 9, are further used to compute the multiplications in the nodes *4*, *12*, *20* and *28* which are started in the clocks 9, 10, 11 and 12 respectively. Hence, the results obtained in the clocks 3, 5, 7 and 9 are all needed to be stored. If we continue in this manner we shall see that the scheduling in Eq. (6.2) would require a significant amount of extra storage for storing the intermediate results.

In contrast to the scheduling in Eq. 6.2, if we follow the algorithm Schedule, then the starting and the finishing time of the nodes would be as:

Nodes	*2*	*6*	*4*	*10*	*8*	*12*	*14*	*18*	*16*	*20*	*22*	*26*	*24*	*28*	*30*
Starting Time:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Finishing Time	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Number of intermediate storages for this schedule is just one and can be seen from the following considerations.

- Node *2* is completed in clock 3 and in the same clock node *4* gets started which requires the result of the multiplication in clock 3 thus the result of node *2* is not required to be stored.
- In clock 4 node *6* is completed and *10* is started, as *10* does not depend on *6*, hence the result of node *6* needs to be stored.
- Continuing in this way we see that only the results of nodes *6*, *8*, *12* and *20* are needed to be stored (they are underlined in the table above).
- But, this does not mean that four distinct storage locations are required, as the storage locations can be reused.
- Note that node *8* is ready in clock 7 and it is required to be stored. Node *6* was stored previously, and the result was already utilized when node *8* started in clock 5. Thus the location used for storing *6* can be used to store *8*.

- Arguing in this manner the total number of storage locations required in this case is just 1.

6.5.1 Determining the number of intermediate storage locations required by Schedule

The design of the algorithm Schedule tries to minimize the requirement of extra storage by trying to use the intermediate results as quickly as possible. For any given input, the extra storage requirements of Schedule can be easily determined from the following two simple principles.

1. A result x is required to be stored if it is completed in a certain clock t and the node y which starts at t is not a parent of x.
2. If there exists a storage location which stores results that have been already used, then the location can be reused, otherwise a new storage location must be defined.

The extra storage requirement for Schedule grows very slowly with the increase in the number of message blocks. Figure 6.6 shows the number of storage for various number of message blocks for $\mathsf{NS} = 3$.

The values reported in Figure 6.6 are obtained by the procedure described above. It may be possible to come up with a closed form formula which shows the amount of extra storage required for each configuration. This combinatorial problem is not straightforward and remains open. For all practical purposes the procedure depicted above can give an exact count of the extra amount of storage required.

6.6 A Hardware Architecture for the Efficient Evaluation of BRW Polynomials

Utilizing the nice properties of the BRW polynomials as discussed in the previous sections we propose a hardware architecture for computing such polynomials. We "show-case" our architecture for 31 blocks of messages using a three-stage pipelined multiplier. The number of message blocks of the polynomial and the pipeline stages of the multiplier can be varied without hampering the design philosophy. This issue of scalability is discussed later.

Each block is 128 bits long, and so the multiplication, addition and squaring operations take place in the field $\mathbb{F}_{2^{128}}$ generated by the irreducible polynomial $\tau(x) = x^{128} + x^7 + x^2 + x + 1$. This specific design would be also useful for the designing of tweakable enciphering schemes

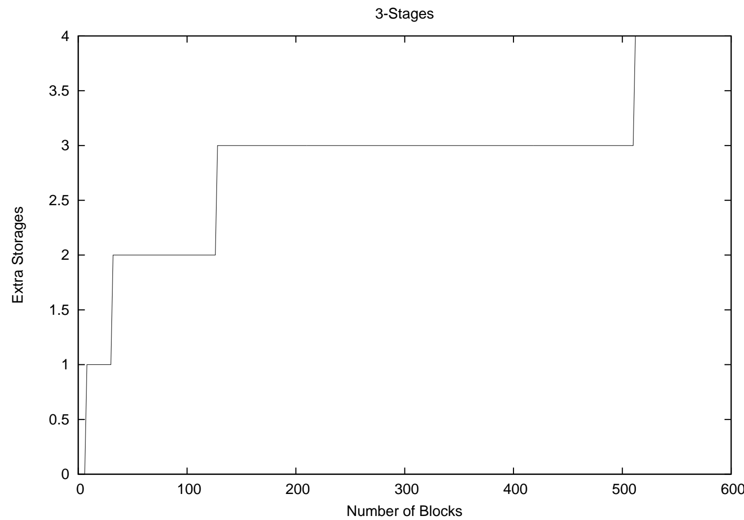


Figure 6.6: The growth of number of extra storage locations required with the number of blocks for $\mathsf{NS} = 3$.

which are discussed in Section 4.2. We describe the design of the multiplier followed by the hardware architecture for the polynomial hash using BRW polynomials.

6.6.1 The Multiplier

Karatsuba multipliers as discussed in Section 5.2.2 computes the polynomial product $c = A \cdot B$, for $A = A^L + A^H x^{64}$, and $B = B^H + B^H x^{64} \in \mathbb{F}_{2^{128}}$ as,

$$C \;=\; A^L B^L + \Big[(A^H + A^L)(B^L + B^H) - (A^H B^H + A^L B^L)\Big]\, x^{64} + A^H B^H x^{128}.$$

With a computational cost of three 64-bit polynomial multiplications and 4 additions/subtractions. By applying this strategy recursively, in each iteration each degree polynomial multiplication is transformed into three polynomial multiplications with their degrees reduced to half of its previous value. After 7 iterations of applying this recursive strategy, all the polynomial operands collapse into single coefficients. However, it is common practice to stop the Karatsuba recursion earlier, performing multiplications with small operands using alternative techniques that are more compact and/or faster.

The Karatsuba multiplier block implemented here is different from the one described in Section 5.2.2 by the fact that we apply a pipelining strategy here. The three-stage pipelined design adopted in this work is shown in Figure 6.7.

The multiplier shown in Figure 6.7 uses three 64-bit Karatsuba multipliers, and in turn,

each one of them are composed by three 32-bit multipliers and successively we implemented 16-bit and 8-bit multiplier blocks. We decided to stop the Karatsuba recursion at 4-bit level, where we used a school-book multiplier. After a careful timing analysis we decided to place registers at the output of the three 64-bit multipliers, at the output of the 8-bit multiplier and finally, after the 128-bit reduction block. This gives us a three-stage pipelined multiplier with each one of its three stages balanced in terms of their critical path. In fact, the critical path of the first stage is shorter than the other two stages because we wanted to include the critical path associated to the input multiplexer block (see Figure 6.8) as a part of the critical path associated with the other two stages of our Karatsuba multiplier architecture.

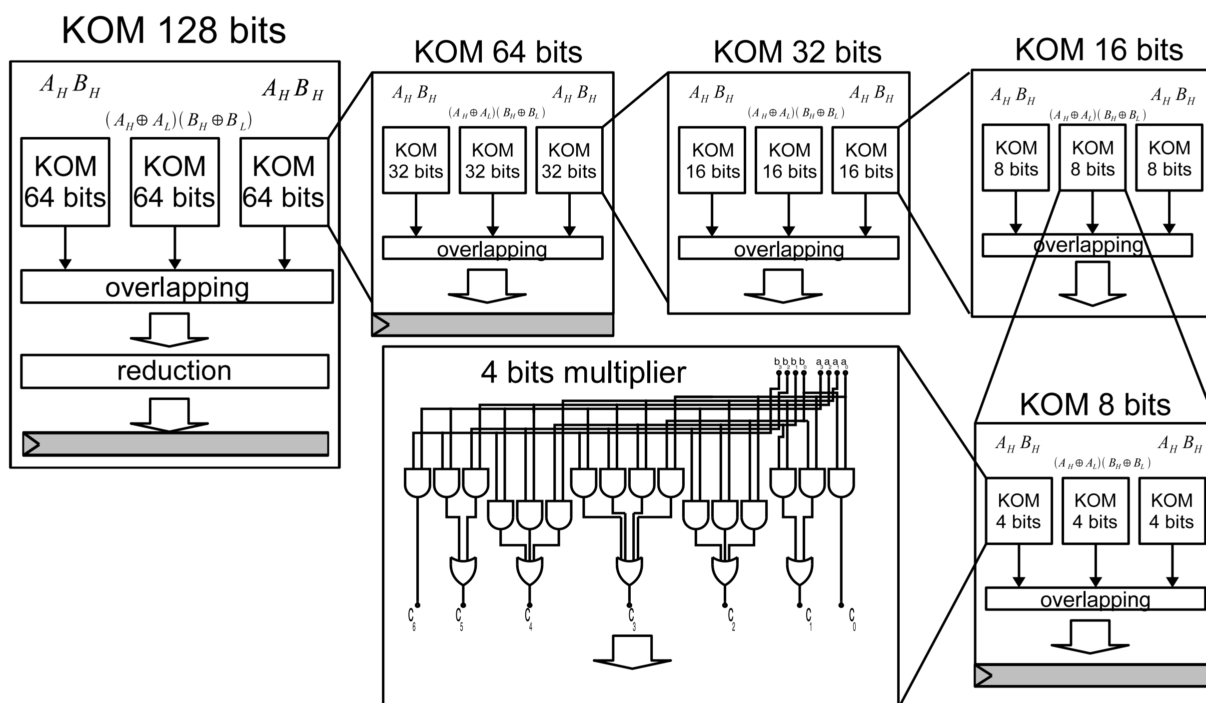


Figure 6.7: Architecture of the three-stage pipelined Karatsuba multiplier

6.6.2 Hardware Architecture to Evaluate BRW Polynomials

The schematic diagram of the proposed architecture is shown in Figure 6.8, where the principal component is a three-stage pipelined Karatsuba multiplier denoted as KOM. At the output of the multiplier, we placed two accumulators, ACC1 and ACC2, which are used to accumulate intermediate results.

Figure 6.8 also includes two blocks for computing squares in the field $\mathbb{F}_{2^{128}}$. These circuits are depicted in the diagram as Sqr1 and Sqr2. Computing squares in binary extension fields

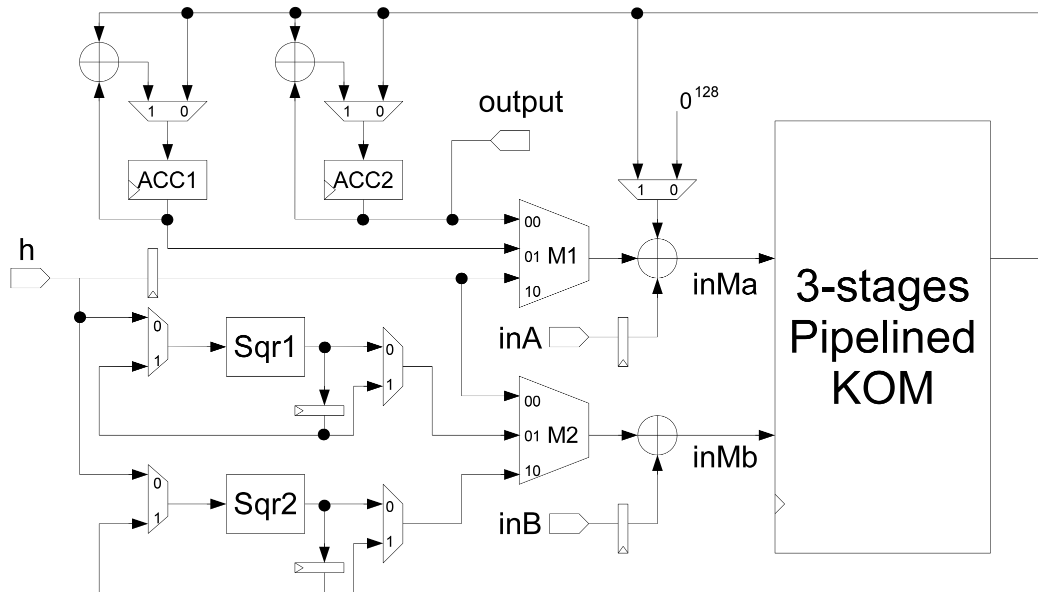


Figure 6.8: Architecture for computing the BRW polynomial for $m = 31$.

are much easier than multiplications. The strategy used for computing squares is as follows.

Let $\alpha \in \mathbb{F}_{2^{128}}$. Then, α can be seen as a polynomial $\alpha = \sum_{i=0}^{127} b_i x^i$, where each $b_i \in \{0, 1\}$. Then

$$\alpha^2 = \left(\sum_{i=0}^{127} b_i x^i \right)^2 \bmod \tau(x) = \sum_{i=0}^{127} b_i x^{2i} \bmod \tau(x).$$

Both squaring blocks in Figure 6.8 are equipped with output registers that allow to save the last field squaring computation. The multiplier block KOM has two inputs designated as inMa and inMb.

The first multiplier input (inMa) is the field addition of three values. Explanations of these values are as follows.

1. The first of these values is the output of a multiplexer block M1 that selects between the key h or any one of the two accumulators.
2. The second value is the output of another multiplexer that selects between the last output produced by the multiplier or zero.
3. Finally, the third value is the input signal inA.

The second multiplier input (inMb) consists of the field addition of two values. Explanations

of these values are as follows.

1. The first one is taken from the output of a multiplexer M2 that selects either the output of Sqr1, or Sqr2 or the key h.
2. The second value is the input inB.

As was discussed in Section 6.1, the computation of a 31-block BRW polynomial denoted as, $H_h(P_1, \ldots, P_{31})$, requires the calculation of $\lfloor \frac{31}{2} \rfloor = 15$ multiplications. We give in Figure 6.9 the time diagram that specifies the way that these fifteen multiplications were scheduled. The final value of the polynomial $H_h(P_1, \ldots, P_{31})$ is obtained in just eighteen clock cycles.

The dataflow specifics of the architecture in Figure 6.8 is shown in the time diagram of Figure 6.9. This figure shows the different data stored/produced in the various blocks at each clock cycle along with the order in which the multiplications were performed. $M_1, \ldots, M_{15}$ denote the fifteen multiplications to be computed and the multiplicands are depicted in the rows designated inMa and inMb, which are the two inputs of the KOM block. The row designated C denotes the output of the multiplier. As a three-stage pipelined multiplier is being used, a multiplication scheduled at clock i can be obtained at C in clock $i + 3$.

The rows ACC1 and ACC2 denote the values which are accumulated in the accumulators in the various clock cycles. Note that an entry M_i in any of the rows representing the state of the two accumulators signify that the value M_i gets xor-ed to the current value in the accumulator, and an entry $*M_i$ denotes that the accumulator gets initialized by M_i.

The rows squaring1 and squaring2 show the state of the squaring circuits output register. Each of the circuits for squaring can compute the square of the current content of the output register in one clock cycle, maintain its current state, or initialize its value with h^2 taking h as a fresh input.

As depicted in Figure 6.9, the computation of the polynomial $H_h(X_1, \ldots, X_{31})$ can be completed in 18 clock cycles and the final value can be obtained from the accumulator ACC2.

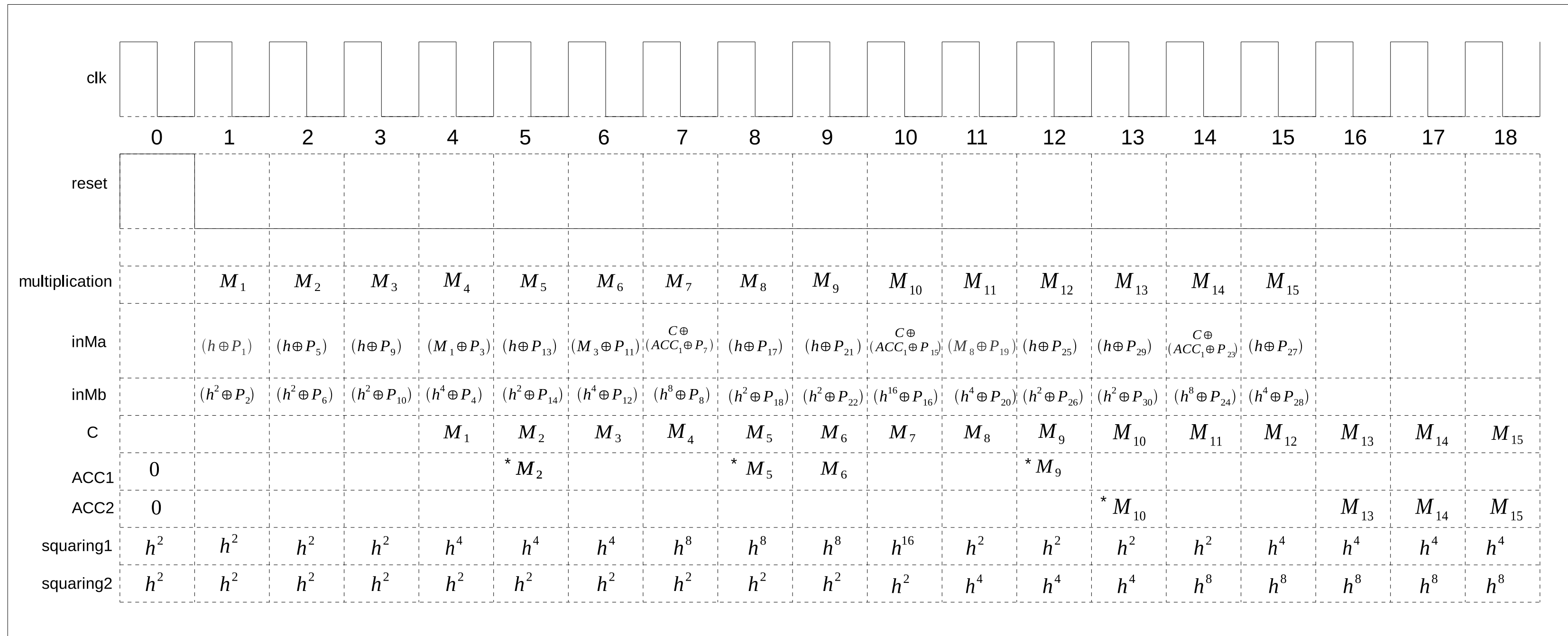


Figure 6.9: Time diagram of the circuit in Figure 6.8.

The circuit shown in Figure 6.8 uses the strategy of computing the squares as required on the fly. An alternative strategy would be to pre-compute the required powers of h and store them in registers. By using this strategy we can get rid of the squaring circuits at the cost of some extra storage, and come up with a circuit which would be very similar to the circuit described in Figure 6.8.

If the pre-computing strategy is adopted, then for computing $H_h(P_1, \ldots, P_{31})$ we need to store h^2, h^4, h^8, h^{16} in registers. The multiplexer which feeds inMb in this case would be a five-input multiplexed, where four of the inputs come from the registers where the squares were stored and the fifth input is the input line h. As squaring in binary extension fields is easy, these two strategies do not provide significantly different performances. This becomes evident from the experimental results.

Irrespective of the way in which squarings are performed, the construction of the circuit follows the scheduling strategy as dictated by the algorithm Schedule. According to Theorem 6.1, if a three-stage pipelined multiplier is used, then for computing $H_h(P_1, \ldots, P_{31})$ the 15 multiplications can be scheduled in 15 clock cycles without any pipeline bubbles.

Figure 6.9 shows that this is indeed the case as starting from clock 1 to 15, in each clock, a multiplication gets scheduled without any pipeline delays. The extra storage required to store the intermediate products is provided by the accumulator ACC1, which stores the products M_2, M_5, M_6 and M_9.

ACC2 is used to accumulate the final result, note that the products M_{10}, M_{13}, M_{14} and M_{15} are accumulated in order in the accumulator ACC2. These multiplications corresponds to the nodes *16*, *30*, *24*, *28* of the collapsed forest (see Figure 6.3(b)).

6.6.3 Scalability

The architecture presented previously is meant for 31-block messages. But the same design philosophy can be used for k-block messages for any fixed k.

Here we give a short description of how the circuit for computing $H_h(P_1, \ldots, P_m)$ grows with the growth of m. A 3-stage pipelined multiplier is assumed. For ease of exposition, we shall only consider the case where the powers of h are pre-computed.

The main components of the circuit will be the two multiplexers which are connected to the inputs of the multiplier, the accumulators and the registers to store the powers of h. If $H_h(P_1, \ldots, P_m)$ is to be computed, then we will require to store $h^2, h^4, \ldots, h^{2^s}$ where $2^s \leq m < 2^{s+1}$. This will require s registers.

$M2$ would thus be a $(s+1)$-input multiplexer. The number of accumulators required would be at most one more than the number of extra storages required. For a given polynomial $H_h(P_1, \ldots, P_m)$, the number of extra storages required by Schedule can be determined using

the procedure described in Section 6.5.

If the number of accumulators required is α then M1 would be substituted by an $(\alpha + 1)$-input multiplexer, where α inputs come from the accumulators and the last one is the input line h. The dataflow specifics can be automatically obtained from the algorithm Schedule.

6.7 Summary and Discussions

In this Chapter we explored the possibility of applying pipelined multipliers for computing BRW polynomials. To achieve this we analyzed the structure of the BRW polynomial. Our analysis viewed the polynomial as a tree where addition and multiplication nodes are interconnected with each other. Viewing the BRW polynomial as a tree immediately gives us information about the dependence of the various operations required for its computation. We discovered some interesting properties of the tree, and used these properties to design a scheduling algorithm. The scheduling algorithm takes as input a BRW polynomial and the desired number of pipeline stages and outputs the schedule (or order) in which the different multiplications are to be performed. This schedule has several attractive features.

1. For pipeline structures with two or three stages, we give a full characterization of the number of clock cycles that is required for computing the polynomial.
2. The schedule ensures that the pipeline delays would be minimal.
3. The scheduling algorithm greedily attempts to minimize the storage. We show that the requirement of extra storage grows very slowly with the increase in the number of blocks.

Utilizing the schedule produced by the scheduling algorithm we came out with a hardware architecture that is meant for computing BRW polynomials with a fixed number of message blocks. We show-cased a specific architecture which uses 31 blocks of messages and a 3-stage pipelined Karatsuba multiplier. Two variants of the architecture are discussed. In the first one, the field squaring operations are computed on the fly, whereas in the second variant the field squarings are pre-computed. Advantages and disadvantages of the two approaches are compared. Finally, we show that the design philosophy is scalable and can be utilized for different pipeline stages and different number of message blocks.

In spite of the comprehensive study that we present in this Chapter, we think that the following interesting problems which are left open are worth studying in the near future:

1. We provided a full characterization of the a behavior of the algorithm Schedule for small values of m. Though for our and all other practical purposes this would be enough but a full characterization for arbitrary values of m may be an interesting combinatorial

exercise. Such a characterization may also tell us which configurations of the collapsed forest would admit a full pipeline given a number of pipeline stages. This study can help is defining a weaker form of optimality (as mentioned in Section 7.3), which would be achievable in all cases.

2. We provide a method for counting the number of extra storage locations for each configuration of the collapsed forest and a given number of pipeline stages. A combinatorial analysis may yield a closed form formula for counting the extra storage locations.

In the following Chapter we use the BRW architecture developed here to implement two TES namely HEH[BRW] and HMCH[BRW].

Chapter

TES constructions based on BRW Polynomials

You may say I'm a dreamer, but I'm not the only one. I hope someday you'll join us. And the world will live as one.

John Lennon

We shall devote this Chapter to two constructions of TES which uses BRW polynomials. In [125] it was first suggested that BRW polynomials can be used instead of normal polynomials to design tweakable enciphering schemes of the hash-ECB-hash and hash-counter-hash family. In [125] it was also claimed that TES constructions using BRW polynomials would be far more efficient than their counter parts which use normal polynomials. The claim was justified using operation counts, as a BRW polynomial requires about half the amount of multiplications than the normal polynomials. But, in [125] real design issues were not considered and thus there exist no hard experimental data to demonstrate the amount of speedups which can be achieved by the use of such polynomials. Here we concentrate on the real design issues for hardware implementation of some of the schemes described in [125], and ultimately provide experimental results which justifies that TES with BRW polynomials would have higher throughput than the ones using the normal ones.

The material of this Chapter have have been previously published in [27].

7.1 The Schemes

There are two basic schemes described in [125], which are named as HEH and HMCH. The schemes can be instantiated in different ways for different applications. The encryption and decryption algorithms for HEH and HMCH are described in Figures 7.1 and 7.2 respectively [1]. The descriptions are for a specific instantiation which is suitable for the purpose of disk

[1]Note that we already gave a description of HEH in Chapter 4, Figure 4.6. The description in Figure 4.6 assumes a normal polynomial for the hash. Figure 7.1 provides a more general description.

Figure 7.1: Encryption and decryption using HEH.

Algorithm HEH.Encrypt$^T_{h,K}(P_1,\ldots,P_m)$	**Algorithm** HEH.Decrypt$^T_{h,K}(C_1,\ldots,C_m)$
1. $\beta_1 \leftarrow E_K(T)$; $\beta_2 \leftarrow x\beta_1$;	1. $\beta_1 \leftarrow E_K(T)$; $\beta_2 \leftarrow x\beta_1$;
2. $U \leftarrow P_m \oplus \psi_h(P_1,\ldots,P_{m-1})$;	2. $U \leftarrow C_m \oplus \psi_h(C_1,\ldots,C_{m-1})$;
3. $PP_m \leftarrow U \oplus \beta_1$;	3. $CC_m \leftarrow U \oplus \beta_2$;
4. $CC_m \leftarrow E_K(PP_m)$; $V \leftarrow CC_m \oplus \beta_2$;	4. $PP_m \leftarrow E_K^{-1}(CC_m)$; $V \leftarrow PP_m \oplus \beta_1$;
5. **for** $i \leftarrow 1$ to $m-1$,	5. **for** $i \leftarrow 1$ to $m-1$,
6. $PP_i = P_i \oplus U \oplus x^i\beta_1$;	6. $CC_i = C_i \oplus U \oplus x^i\beta_2$;
7. $CC_i \leftarrow E_K(PP_i)$;	7. $PP_i \leftarrow E_K^{-1}(CC_i)$;
8. $C_i \leftarrow CC_i \oplus x^i\beta_2 \oplus V$;	8. $P_i \leftarrow PP_i \oplus x^i\beta_1 \oplus V$;
9. **end for**	9. **end for**
10. $C_m \leftarrow V \oplus \psi_h(C_1,\ldots,C_{m-1})$;	10. $P_m \leftarrow V \oplus \psi(P_1,\ldots,P_{m-1})$;
11. **return** $(C_1,\ldots,C_m)$;	11. **return** $(P_1,\ldots,P_m)$;

encryption.

In the description of the algorithms we assume that $E_K : \{0,1\}^n \rightarrow \{0,1\}^n$ is a block cipher, whose inverse is $E_K^{-1} : \{0,1\}^n \rightarrow \{0,1\}^n$. The additions and multiplications are all in the field $\mathbb{F}_{2^n}$ represented by a irreducible polynomial $\tau(x)$ of degree n which is primitive. For our implementations we use the field $\mathbb{F}_{2^{128}}$ and $\tau(x) = x^{128}+x^7+x^2+x+1$. An $A \in \{0,1\}^n$ can be seen as a polynomial $a_0 + a_1x + \cdots + a_nx^{n-1}$ where each $a_i \in \{0,1\}$, thus every n bit string A can be treated as an element in $\mathbb{F}_{2^n}$. By xA we mean the n bit binary string corresponding to the polynomial $x(a_0+a_1x+\cdots+a_nx^{n-1}) \bmod \tau(x)$. This operation can be performed easily by a shift and a conditional xor. In the description $\psi_h(.)$ can be instantiated in two different ways, it can either be $h\mathsf{Poly}_h(.)$ or $hH_h(.)$, where $H_h(.)$ is a BRW polynomial. From now onwards to avoid confusion we shall represent a BRW polynomial by $\mathsf{BRW}_h(.)$, and for the two different instantiations we shall call the schemes as HEH[BRW], HEH[Poly] and HMCH[BRW], HMCH[Poly].

7.2 Analysis of the Schemes and Design Decisions

We analyze here the schemes presented in Section 7.1 from the perspective of efficient hardware implementations and thus come up with some basic strategies for designing them. As always, the implementation is targeted towards the disk encryption application, thus in the following discussions we shall only consider messages of fixed lengths which are 512 byte long, i.e. 32 blocks of 128 bits. Our primary design goal is speed, but we shall try to keep the area metric reasonable. The basic components of both schemes are a block cipher (which we chose to instantiate using AES-128) and the polynomial hash (either Poly or BRW). Thus,

Figure 7.2: Encryption and decryption using HMCH.

Algorithm HMCH.Encrypt$^T_{h,K}(P_1,\ldots,P_m)$	**Algorithm** HMCH.Decrypt$^T_{h,K}(C_1,\ldots,C_m)$
1. $\beta_1 \leftarrow E_K(T)$; $\beta_2 \leftarrow x\beta_1$;	1. $\beta_1 = E_K(T)$; $\beta_2 = x\beta_1$;
2. $M_1 \leftarrow P_1 \oplus \psi_h(P_2,\ldots,P_m)$;	2. $U_1 \leftarrow C_1 \oplus \psi_h(C_2,\ldots,C_m)$;
3. $U_1 \leftarrow E_K(M_1)$; $S \leftarrow M_1 \oplus U_1 \oplus \beta_1 \oplus \beta_2$;	3. $M_1 \leftarrow E_K^{-1}(U_1)$; $S \leftarrow M_1 \oplus U_1 \oplus \beta_1 \oplus \beta_2$;
4. **for** $i = 2$ to m,	4. **for** $i = 2$ to m,
5. $C_i \leftarrow P_i \oplus E_K(x^{i-2}\beta_1 \oplus S)$;	5. $P_i \leftarrow C_i \oplus E_K(x^{i-2}\beta_1 \oplus S)$;
6. **end for**	6. **end for**
7. $C_1 \leftarrow U_1 \oplus \psi_h(C_2,\ldots,C_m)$;	7. $P_1 \leftarrow M_1 \oplus \psi_h(P_2,\ldots,P_m)$;
8. **return** $(C_1,\ldots,C_m)$;	8. **return** $(P_1,P_2,\ldots,P_m)$;

in terms of hardware the basic components required would be an AES (both encryption and decryption cores) and an efficient finite-field multiplier. As the focus of this work is in BRW polynomials, in the rest of this Section we shall discuss about the instantiation with only BRW polynomials here, the instantiation with $\mathsf{Poly}_h()$ is briefly discussed in Section 7.4.3.

Referring to the algorithm HEH.Encrypt$^T_{h,K}$ of Figure 7.1, we see that irrespective of the choice of $\psi_h(.)$, $(m+1)$ encryption calls to the block-cipher are required, whereas HEH.Decrypt$^T_{h,K}$ requires one encryption call and m decryption calls to the block cipher. The encryption/decryption calls in lines 4 and 7 of both HEH.Encrypt and HEH.Decrypt procedures are independent of each other and thus can be suitably parallelized. Algorithm HMCH.Encrypt$^T_{h,K}$ of Figure 7.2, requires $(m+1)$ encryption calls to the block-cipher, and for HMCH.Decrypt$^T_{h,K}$, m encryption calls and one decryption call to the block-cipher are required. The $(m-1)$ block-cipher calls required by both encryption and decryption procedures of HMCH can be parallelized. Thus, for both modes the bulk amount of block-cipher calls can be parallelized. This suggests that a pipelined implementation of AES would be useful for implementing the ECB mode in HEH and the counter type mode in HMCH. Computation of the $\mathsf{BRW}_h(.)$ can also be suitably parallelized (as discussed in Section 6.6). Thus we also decided to use a pipelined multiplier to compute the BRW hash.

As a target device for the implementation we choose FPGAs of the Virtex 5 family. These are one of the most efficient devices available in market. In [17] a highly optimized AES design suitable for Virtex 5 FPGAs was reported. One important design decision taken in [17] was to implement the byte substitution table using the LUT fabric, this is in contrast to previous AES designs (including the one presented in Chapter 5) where extra block RAMs were used for the storage of the look up tables. This change has a positive impact both in area and the length of the critical path, given rise to better performance. The design described in [17] is sequential. The AES design implemented in this work closely follows the techniques used in [17], but we suitably adapt and extend the techniques in [17] to a pipelined

Table 7.1: The permutation $\pi(x)$

x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$\pi(x)$	1	2	7	8	3	4	13	14	5	6	11	12	9	10	19	20	15	16	21	22	17	18	27	28	23	24	29	30	25	26	31

design of ten stages. Moreover, another important characteristic of our AES design is that we do not attempt to design a single core for the encryption and decryption functionalities as was done in Chapter 5 but instead, we chose to design separate cores for encryption and decryption. This gives us better throughput and also provides some extra flexibility in terms of optimization. For one of the schemes we required a sequential AES decryption core. In our experiments we were unable to obtain good performance for the decryption core using strategies used in [17]. The design of the sequential decryption core uses ideas from [56],where operations inverse byte substitution (IBS) and inverse mixcolumn (IMC) are combined together in a single module which are called inverse T-Boxes. We implemented the T-boxes using large multiplexors and avoided the use of memories, for this reason our designs occupies more slices. Details about our AES implementation is provided in Section 7.4.1.

As it has been mentioned, in the case of the field multiplier we decided to use a three stage pipelined Karatsuba multiplier. The number of stages was fixed keeping an eye to the critical path of the circuit. Once we fixed our design for AES we selected the pipeline stages for the multiplier in such a manner that it matches the critical path of the AES. As both components would be used in the circuit, hence if a very high number of pipeline stages for the multiplier is selected then, the critical path would be given by the AES but the latency for multiplication would increase. Several exploratory experiments suggested that a three stage pipeline would be optimal as the critical path of such a circuit would just match that of the AES circuit.

Both HEH[ψ] and HMCH[ψ] were proved to be secure as tweakable enciphering schemes in [125]. The security proof requires $\psi_h()$ to be a almost xor universal (AXU) hash function. Both $h\mathsf{BRW}(X_1, \ldots, X_{m-1})$ and $h\mathsf{Poly}(X_1, \ldots, X_{m-1})$ are AXU. If $\pi : \{1, \ldots, m-1\} \rightarrow \{1, \ldots, m-1\}$ be a fixed permutation then it is easy to see that $h\mathsf{BRW}(X_{\pi(1)}, X_{\pi(2)}, \ldots, X_{\pi(m-1)})$ would also be AXU. Thus, using any fixed ordering of the messages for evaluating each of the BRW polynomials in the modes will not hamper their security properties. This observation is important in the context of hardware implementations of HEH[BRW] and HMCH[BRW]. As, for optimal computation of BRW polynomials we require a different order of the messages than the normal order. In our case, the permutation $\pi()$ is dictated by the algorithm Schedule. If $m = 31$ and the number of pipeline stages of the multiplier is 3 the permutation π as dictated by Schedule is shown in Table 7.1.

Thus, for implementing HEH[BRW].Encrypt we replace $\psi_h(P_1, \ldots, P_{31})$ in line 2 of Figure 7.1 by $h\mathsf{BRW}(P_{\pi(1)}, \ldots, P_{\pi(31)})$. Similar change is done in line 10 of the encryption algorithm and lines 2 and 10 of the decryption algorithm. For implementing HMCH[BRW] we replace $\psi_h(P_2, \ldots, P_{32})$ in line 2 of Figure 7.2 by $h\mathsf{BRW}(P_{\pi(1)+1}, P_{\pi(2)+1}, \ldots, P_{\pi(31)+1})$. Similar change is done in line 10 of the encryption algorithm and lines 2 and 10 of the decryption algorithm.

7.3 Analysis of the schemes

With these basic design decisions as described above, we shall analyze HEH and HMCH to exploit the maximum parallelization possible. The following discussion assumes the use of $h\mathsf{BRW}(.)$ in place of $\psi_h(.)$ and the number of blocks to be 32 for both the schemes. First we analyze HEH which is described in Fig. 7.1. In Line 2 of the encryption algorithm the computation of the BRW polynomial on 31 blocks takes place. Using a 3 stage pipelined multiplier and the design described in Section 6.6, $\mathsf{BRW}(P_1, \ldots, P_{31})$ can be completed in 18 clock cycles and computation of $h\mathsf{BRW}(P_1, \ldots, P_{31})$ would thus require 21 clock cycles for the extra multiplication with h. Thus the computation of U (as in line 2) can be completed in 21 clock cycles. The computation of β_1 and β_2 (in line 1) can be done in parallel with the computation of U.

Then in lines 4 to 9 the main operations required are 32 calls to AES. Following our design these 32 calls can be completed in 43 cycles, and after an initial delay of 11 cycles we shall obtain one value of C_i ($i < m$) in each cycle. For computing C_m we again need to compute the BRW polynomial which would take 21 cycles. The computation of the BRW polynomial can be parallelized with the block cipher calls, as soon as we start getting outputs of the AES calls we can start computing the BRW polynomial necessary in line 10. The specific architecture that we have designed for the BRW polynomials requires the availability of two input blocks per each clock cycle. For this reason we decided to have two AES cores running in parallel which can feed the circuit for computing the BRW polynomials and thus can reduce the total latency of the circuit. Using this strategy, all values of CC_i would be produced in 27 cycles instead of 43. After 11 of these 27 cycles we can start computing the BRW polynomial and would require a total of 21 cycles to complete. The total computation can be completed in 55 cycles if two AES cores are used. This description is summarized in the time diagram in Figure 7.3 (a). Decryption would be similar, but we need to implement two AES decryption cores for obtaining the same latency as encryption.

If we use a single AES core, then we would not be able to do the BRW computation in line 11 in 21 cycles as in each cycle we shall not be able to obtain two blocks of data as required, thus for each multiplication we need to wait two cycles, and thus the total computation for the second hash (in line 11) would require 35 cycles, and the total computation would require 69 cycles.

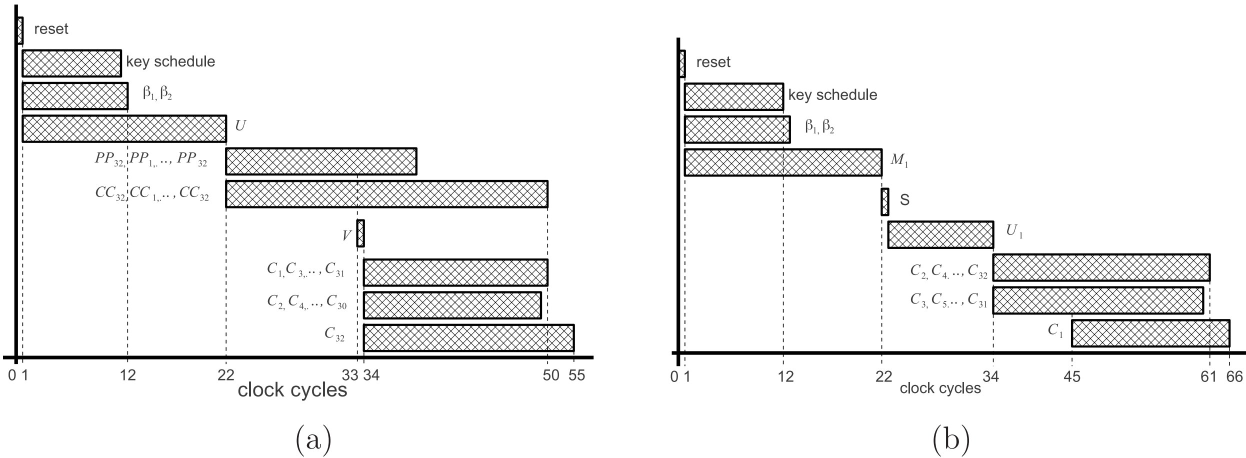


Figure 7.3: Time diagram of HEH and HMCH: (a) HEH (b) HMCH.

The time diagram for HMCH is shown in Figure 7.3(b). In case of HMCH also line 2 can be completed in 21 cycles and line 1 can be performed in parallel with line 2. For computing line 3 which involves a single AES call we would need to wait 11 cycles. Again, using two ten staged pipelined AES encryption cores the computation in line 5, which involves 31 calls to the AES in counter mode can be completed in 27 cycles. After 11 of these 27 cycles, the BRW hash can be started and it would require 21 cycles to compute. Thus, the total computation could be done in 66 cycles. In case of decryption there is only one inverse call to the AES as in line 3. Thus, for decryption there is no need to implement two AES decryption cores as is required in case of HEH. Only one decryption core is sufficient in this case and also as there is only one call, a pipelined design for this core is also un-necessary. Hence, we designed a sequential decryption core which saved us some area. If a single AES core is used, as in the case of HEH in HMCH also an additional 14 cycles would be required for computing the second hash.

7.4 Architecture of HMCH[BRW]

We implemented the modes HEH[BRW] and HMCH[BRW]. For both the modes both encryption and decryption functionality were implemented in a single chip. In this section we shall only describe the architecture for HMCH[BRW] which uses two pipelined encryption cores and a single sequential decryption core, first at all we will describe the AES designs.

7.4.1 The AES

We designed the AES encryption and decryption cores separately. For HEH[BRW] we used two pipelined AES encryption cores and two pipelined AES decryption cores and for

HMCH[BRW] we used two pipelined AES encryption core and one sequential decryption core. The AES design closely follows the techniques used in [17]. In [17] the S-boxes were implemented as 256×8 multiplexers. This was possible due to special six input lookup tables (LUT) available in Virtex 5 devices. One S-box fits in 32 six inputs LUTs available in Virtex 5 FPGA devices. In [17] the authors presented a sequential core, we extended their idea to a $10-stages$ pipelined core. Initially, both the encryption and decryption cores take 10 clock cycles to produce a valid output, and produces one block as output in subsequent cycles. AES encryption/decryption consist of 10 rounds, the rounds 1 to 9 has four transformations: SubBytes (BS), ShiftRows (SR), MixColumns (MC) and AddRoundKey (ADDRK). The last round has only three transformations BS, SR and ADDRK. The decryption core looks similar to figure 7.4, where each transformation is replaced by its inverse and the order of rounds are inverted (i.e., the computation starts at round 10 and the final output is given by the xor of initial key and output of round 1).

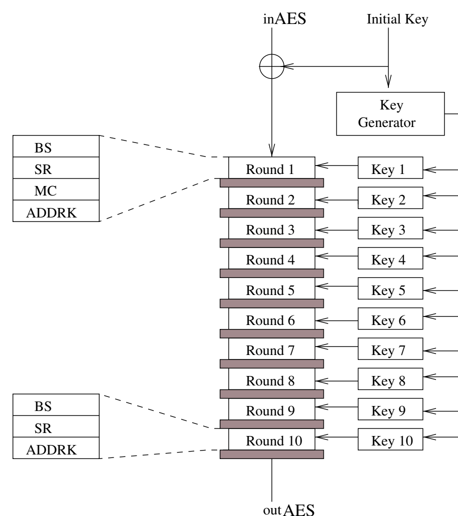


Figure 7.4: Architecture for 10-stages pipelined encryption AES core.

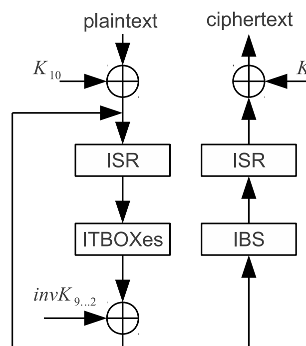


Figure 7.5: Architecture for sequential decryption AES core.

The sequential decryption core in HMCH[BRW] was implemented using the techniques in [56]. In [56] the operations BS and MC are combined in a substitution operation which the authors call as T-boxes. In our implementation for the decryption core we used inverse T-boxes (ITbox) which combines the operations inverse byte substitution (IBS) and inverse mixcolumn (IMC). We implemented the T-boxes using large multiplexers and avoided the use of memories. As a sequential core was only required hence we needed to implement only two rounds (see Fig. 7.5). Irrespective of the number of cores used we used a single key generator and the S-boxes for the key generator were also implemented using multiplexors.

7.4.2 Design of HMCH

The simplified architecture for HMCH[BRW] is depicted in Fig. 7.6. For ease of exposition in Fig. 7.6 we only show the encryption part of the circuit, an additional component of the circuit is the sequential decryption core which we omit for the sake of simplicity. The main components of the general architecture depicted in Fig. 7.6 are the following: A BRW polynomial hash block (which corresponds to the circuit shown in Fig. 6.8), two AES cores (equipped with both electronic code book and counter mode functionalities), and two $x^2 Times$ blocks. The $x^2 Times$ blocks compute $x^2 A$, where $A \in \mathbb{F}_{2^{128}}$. The architecture also includes five registers to store the values M_1, β_1, β_2, U_1 and S, and makes use of six multiplexer blocks labeled 1 to 6 in the figure and we shall refer to them as mux1 to mux6. When the $x^2 Times$ block is first activated, it simply outputs the value placed at its input (for the circuit of Fig. 7.6, this input value will correspond to either β_1 or β_2). Thereafter, at each clock cycle the field element $x^2 A$ will be produced as an output, where $A \in \mathbb{F}_q$ is the last value computed by this block. The control unit of this architecture consists of a ROM memory where a microprogram with sixty seven micro-instructions has been stored, each microinstruction consisting of 28-bit control words. Additionally, the control unit uses a counter that gives the address of the next instruction to be executed.

The general dataflow of Fig. 7.6 can be described as follows. First the parameter β_1 is computed as $\beta_1 = E_K(T)$. This is done by properly selecting mux1 and mux2 so that the tweak T gets encrypted in single mode by the AES_{even} core. The value so obtained is stored in the register $reg\beta_1$ and also $\beta_2 = x\beta_2$ is computed and stored in $reg\beta_2$. Then, the plaintext blocks $P_2, \ldots, P_m$ are fed into the BRW hash block through the inputs inA and inB and the proper selection of mux4 and mux5. After 21 clock cycles, the hash of the plaintext blocks is available at $outHash$, allowing the computation of the parameter M_1 as, $M_1 = outHash \oplus P_1$, where P_1 is taken from the input signal inB. The parameter U_1 is computed as $E_K(M_1)$ by selecting the third input of mux1 as the input value for the AES_{even} core. The value so computed is stored in $regU_1$. At this point the circuit of Fig. 7.6 is ready to compute the encryption in counter mode of $m-1$ plaintext blocks (corresponding to line

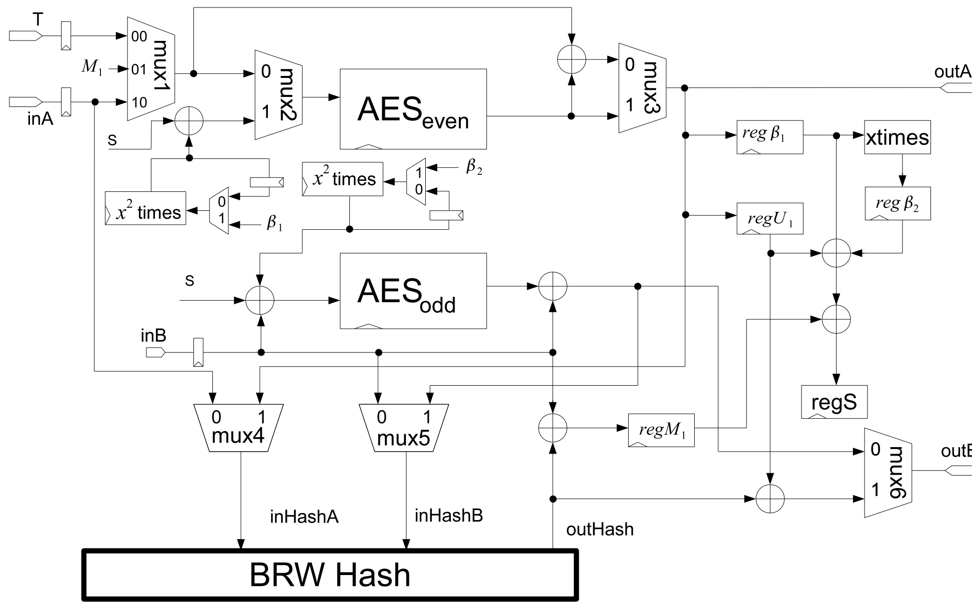


Figure 7.6: Architecture for performing the HMCH[BRW] Encryption Scheme in hardware.

5 of the HMCH[BRW] encryption algorithm shown in Fig. 7.2) as,

$$AES_{even}: \quad C_i \leftarrow P_i \oplus E_K(x^{i-2}\beta \oplus S), \quad \text{for } i = 2, 4, \ldots, 31.$$
$$AES_{odd}: \quad C_i \leftarrow P_i \oplus E_K(x^{i-2}\beta \oplus S), \quad \text{for } i = 3, 5, \ldots, 32.$$

It is noticed that this last computation is achieved in 28 clock cycles using the two AES cores in parallel. The encryption blocks C_i for $i = 2, \ldots, m$ are simultaneously sent to circuit's outputs *outA* and *outB*, and to the BRW hash block through a proper selection of mux4 and mux5. After 21 clock cycles, the cipher blocks' hash is available at *outHash*, allowing the computation of the encryption block C_1 as, $C_1 = outHash \oplus U_1$, where U_1 was previously computed and stored as explained above.

Fig. 7.3(b) shows the time diagram of the sequence of operations required for computing the HMCH[BRW] encryption of a sector that has a size of thirty-two AES blocks. We first compute in parallel the BRW polynomial hash function M_1 along with the AES round key generation and the computation of β_1, β_2. After eleven clock cycles we thus obtain $\beta_1 = E_K(T)$ and $\beta_2 = x\beta_1$, whereas the computation of M_1 is completed after twenty one clock cycles. Hence, we can start at cycle 22 the computation of U_1. We are able to obtain the parameter S at clock cycle 34, which is required to start the counter mode encryption. Then, from clock cycle forty-four up to sixty, the counter mode encryption is able to deliver two valid input data that will be used for computing the second hash function M_2. The last value C_1 that corresponds to the final result of the hash function M_2 is delivered at clock cycle sixty-seven. This completes the encryption of the 32 AES block sector.

7.4.3 HEH[Poly] and HMCH[Poly] Using Pipelined Multipliers

For the sake of comparison we also implemented HEH[Poly] and HMCH[Poly]. As stated in Section 4.2 these schemes can be obtained by replacing $\psi_h()$ by $\mathsf{Poly}_h()$ in the algorithms of Figures 7.1 and 7.2. When a normal polynomial is used for the constructions then the usual Horner's rule is the most efficient way to compute it. At first glance, the advantages of a pipelined multiplier cannot be used due to the sequential nature of the Horner's rule. In [129] A three way parallelization strategy was proposed to evaluate a normal polynomial using three different multipliers and thus running three different instances of the Horner's rule in parallel. We adopt the strategy presented in [129] by utilizing a three staged pipelined multiplier as a tool to evaluate a normal polynomial using Horner's rule.

As we are interested in encrypting 32 blocks of messages hence in case of both HEH[Poly] and HMCH[Poly] the polynomial to be computed is

$$\begin{aligned} \psi_h(P_1,\ldots,P_{31}) &= h\mathsf{Poly}_h(P_1,P_2,\ldots,P_{31}) \\ &= h\sum_{i=1}^{31} P_i h^{31-i} \\ &= h(p_1+p_2+p_3) \end{aligned}$$

where

$$\begin{aligned} p_1 &= \sum_{i=1}^{11} P_{3i-2}(h^3)^{11-i} \\ p_2 &= h^2\sum_{i=1}^{10} P_{3i-1}(h^3)^{10-i} \\ p_3 &= h\sum_{i=1}^{10} P_{3i}(h^3)^{10-i}. \end{aligned}$$

Note that the multiplications in p_1 does not depend on the multiplications in p_2 and p_3, etc. Hence, a three staged pipelined multiplier can be used to compute $h\mathsf{Poly}_h(P_1,P_2,\ldots,P_{31})$. If h^2 and h^3 are pre-computed then the computation of the polynomial can be completed in 35 clock cycles.

For HEH[BRW] we used two pipelined AES encryption and decryption cores and for HMCH[BRW] we used two pipelined encryption core and a single sequential decryption core. The usage of two AES cores gave us considerable savings in the number of clock cycles as discussed

Table 7.2: Primitive operations on Virtex-5 device.

Core	Slices	Cycles	Frequency MHz	Throughput Gbits/Sec	Throughput/Slice
AES pipelined encryption (AES-PEC)	2859	1	300.56	38.47	0.0134
AES pipelined decryption (AES-PDC)	3110	1	239.34	30.72	0.0098
AES sequential decryption (AES-SDC)	1075	11	292.48	3.40	0.0031
$h\mathsf{Poly}_h(P_1,\ldots,P_{31})$	1886	35	251.38	28.50	0.0151
$h\mathsf{BRW}_h(P_1,\ldots,P_{31})$	2086	21	243.49	46.01	0.0220

in Section 7.3, as $h.\mathsf{BRW}(.)$ could be computed in only 21 cycles. But $h.\mathsf{Poly}(.)$ requires 35 clock cycles to complete, and hence dedicating two cores for this task does not give rise to any savings. Hence, while implementing HEH[Poly], we used one pipelined AES encryption core and one pipelined AES decryption core and for HMCH[Poly] we used one pipelined AES encryption core and one sequential AES decryption core.

7.5 Experimental Results

In this section we present the experimental results obtained from our implementations. All reported results were obtained from place and route simulations, where the target device is XILINX Virtex 5 xc5vlx330-2ff1760. Table 7.2 shows the performance of the basic primitives. Table 7.2 clearly shows that $\mathsf{BRW}_h(.)$ is far better in performance than $\mathsf{Poly}_h(.)$, but $\mathsf{BRW}_h(.)$ occupies more slices than $\mathsf{Poly}_h(.)$. We note that only the pipelined AES decryption core achieved lower frequency than the hash blocks. Thus in case of HMCH[BRW], which does not use the pipelined decryption core, the critical path is given by the hash block and in case of HEH[BRW] the critical path is given by the pipelined decryption core.

For both HEH[BRW] and HMCH[BRW] we implemented three variants, we name these variants as 1, 2 and 3. The naming conventions along with the performance of the variants are described in Table 7.3. Table 7.3 also shows the variants using Poly. From the results shown in Table 7.3 we can infer the following:

1. **BRW versus Poly:** The variants using BRW give better throughput but occupies more area than the variants using Poly.

2. **Single core versus double core:** When two AES cores are used for HEH[BRW] and HMCH[BRW] the throughput is much higher than the case when one AES core is used,

as using two AES cores we can accommodate more parallelization. In particular, the following observations can be made:

- For the two core implementations we gain 14 clock cycles against the one core implementations. The improvement in clock cycles (66 versus 80 in case of HMCH[BRW] ; or 55 versus 69 in case of HEH[BRW]) is not reflected to that extent in the throughput (13 versus 11 in case of HMCH[BRW]; or 15 versus 13 in case of HEH[BRW])). This is due to operation at lower frequencies for the double-core implementations.
- Increase in hardware for HMCH[BRW]-1 over HMCH[BRW]-3 is probably not significant, but, for HEH[BRW] the increase is marked. The reason behind this is for HEH[BRW]-1 two pipelined AES decryption cores are also necessary for achieving the desired parallelization.

3. **Pre-computing squares versus computing squares on the fly:** Pre-computing squares for BRW polynomials gives a negligible improvement on throughput and the circuits using pre-computation utilizes a few slices more than the circuits where squares are computed on the fly.

4. **HEH versus HMCH:**

   - HEH[BRW] gives better throughput than HMCH[BRW]. The reason being the increased latency in case of HMCH[BRW]. HMCH[BRW] has an AES call (the one in line 3 of Figure 7.2) which cannot be parallelized. This results in an additional latency of 11 cycles in HMCH[BRW] compared to HEH[BRW].
   - For the same reason HEH[Poly] gives better throughput than HMCH[Poly].
   - HEH[BRW] requires pipelined AES decryption cores for the required parallelization in decryption but for HMCH[BRW] decryption a sequential AES decryption core is sufficient. Thus, HMCH[BRW] occupy lesser area than HEH[BRW].
   - HEH[BRW]-3 is comparable to HMCH[BRW]-1 and HMCH[BRW]-2 both in terms of number of slices and throughput.

5. **Recommendation:**

   - For best speed performance, use double-core HEH[BRW]; in particular, HEH[BRW]-2.
   - For smallest area, use HMCH[Poly].
   - For best area-time measure, use single-core HMCH[BRW], i.e., HMCH[BRW]-3. The area time measure for HMCH[Poly] is very close to HMCH[BRW]-3.

Table 7.3: Modes of operation on Virtex-5 device. AES-PEC: AES pipelined encryption core, AES-PDC: AES pipelined decryption core, AES-SDC: AES sequential decryption core, SOF : squares computed on the fly, SPC: squares pre-computed

Mode	**Implementation Details**	**Slices**	**Frequency (MHz)**	**Clock Cycles**	**Time (nS)**	**Throughput (Gbits/Sec)**	$\frac{1}{(\text{Slice}*\text{Time})}$
HMCH[BRW]-1	2 AES-PEC, 1 AES-SDC, SOF	8040	211.79	66	311.64	13.14	399.11
HMCH[BRW]-2	2 AES-PEC, 1 AES-SDC, SPC	8140	212.59	66	310.46	13.19	395.71
HMCH[BRW]-3	1 AES-PEC, 1 AES-SDC, SOF	6112	223.36	80	358.16	11.44	**456.81**
HEH[BRW]-1	2 AES-PEC, 2 AES-PDC, SOF	11850	202.86	55	271.13	15.17	311.25
HEH[BRW]-2	2 AES-PEC, 2 AES-PDC, SPC	12002	203.89	55	269.75	**15.18**	308.88
HEH[BRW]-3	1 AES-PEC, 1 AES-PDC, SOF	8012	218.38	69	315.96	12.96	395.02
HMCH[Poly]	1 AES-PEC, 1 AES-SDC	**5345**	225.49	94	416.88	9.83	448.79
HEH[Poly]	1 AES-PEC, 1 AES-PDC	6962	218.19	83	380.39	10.77	377.61

7.5.1 Comparison with implementations in Chapter 5

There are no published data regarding the performance of HEH[BRW] and HMCH[BRW] available in the literature. The closest work with which our designs can be compared is the designs presented in Chapter 5. In Chapter 5, implementations of HEH and HCHfp were provided which are very similar to HEH[Poly] and HMCH[Poly]. The designs in Chapter 5 were optimized for Virtex 4 family of devices, but the performance of the same design for other devices like Virtex II pro and Virtex 5 were also reported. The throughput for HEH and HCHfp reported in Chapter 5 were 5.11 GBits/sec and 4.00 GBits/sec respectively with area overheads of 7494 and 7513 Virtex 4 slices respectively. Our designs of HEH[Poly] and HMCH[poly] achieves much better throughput using lesser area. But, one needs to be careful in comparing the area metric as the structure of the slices in Virtex 4 and Virtex 5 are very different. But our designs achieve much better throughput than the designs in Chapter 5 because of the following reasons:

- Due to better technology used in Virtex 5 higher frequencies are achievable.
- Our design of the AES is specially suited for Virtex 5 and uses the special slice structure of such devices and thus can achieve much better frequencies than the designs reported in Chapter 5.
- The multiplier used in Chapter 5 is a combinatorial circuit which produces one product in each clock cycle, this design gives a much longer critical path than our pipelined

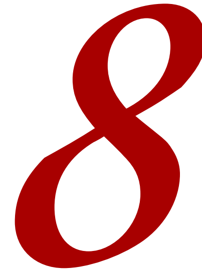
multiplier. Hence our circuits for HEH[Poly] and HMCH[Poly] operates at much higher frequencies and thus give better throughput.

7.6 Final Remarks

Using the architecture of BRW polynomials developed in Chapter 6, we implemented two TES HEH[BRW] and HMCH[BRW], the experiments suggests that BRW polynomials are a far better alternative than normal polynomials in terms of speed for design of TES.

Out designs for HEH and HMCH exploits the parallelism assuming the messages are 32 blocks long, which is the size of a disk sector. In a practical application multiple sectors may be written or read at the same time from a disk. This opens up the possibility of identifying ways of parallelizing across sectors. The structure of both HEH and HMCH would allow such parallelism and such parallelization may yield architectures which would be much more efficient than those reported here. We plan to consider such designs in future.

Chapter



STES: A New TES Amenable to Low Area/Power Implementation

How many years can some people exist
Before they're allowed to be free?

Bob Dylan

In the previous three Chapters we discussed in details some highly efficient architectures for tweakable enciphering schemes. The results obtained are very encouraging as prototypical studies and they demonstrate that TES can match the data rates of modern day disk controllers (which operates around 3Gbits/sec). The studies are targeted towards high end FPGAs hence it may not be cost efficient for large scale deployments in commercial hard disks etc, but the design philosophies adopted in these works can be easily adopted for design of ASICs where it is expected that the throughput rates would be higher as ASICs are capable of operating in much higher frequencies than FPGAs.

Storage is an integral part of numerous modern devices which are constrained in terms of area and power consumption. For example, non-trivial storage is provided in modern mobile phones, cameras etc. Most of these devices do not have hard disks but rely on flash memories. NAND type flash memory has a similar organization as hard disks and it seems that the algorithms that are applicable for hard disks can also be applied here. But one needs to keep in mind the constraints in these devices in terms of area and power utilization. Due to the availability of low cost yet performant FPGAs, now it may be possible to put an FPGA in a personal device like a mobile phone. Using FPGAs in such devices gives a better competitive edge to the manufacturer, as FPGAs have lower development time and also the reconfigurability options allows the manufacturers to add more features to a given device even after the devices have been released in the market or already owned by a customer. These considerations have led to the ideas of replacing a processor in such a device with an FPGA, or to use a hybrid design consisting of both a FPGA and a processor. Such possibilities hints the need for cost efficient FPGA circuits which can be directly deployed to small devices and perform some cryptographic functionalities. In the last few years there have been numerous proposals for light weight cryptographic primitives which when implemented would have a

very small hardware footprint and be very efficient in terms of power consumption [19,61,95]. In this Chapter we aim to address this issue of light weight circuits for storage encryption in constrained devices.

The application that we have in mind is to encrypt flash memories which have a block-wise organization (same as hard disks) and are used in small and/or mobile devices. Our target application includes memory cards like those specified in the SD standard [4]. The data rates for such storage components are much less than that of modern hard disks. For example the SD standard classifies memory cards into four categories based on their speeds. These categories are named as normal speed, high speed, ultra high speed-I (UHS I) and ultra high speed II. They required bus speeds for these categories are specified in the Table below.

Category	Speed range
Normal Speed	12.5 MB/sec
High Speed	25 MB/sec
UHS-I	50 - 104 MB/sec
UHS-II	156-312 MB/sec

It is evident from the above Table that the speed requirements for SD cards are much less than that of modern hard disks, where with the modern technologies like serial ATA and native command queuing, the data rates achieved are more than 3 Gigabits/sec. Also, the UHS-II category of devices are only recommended for special applications like storing high quality streaming video etc. Hence for encryption of SD cards the speed of encryption is not much demanding, and is much less than the speed achieved by the implementations reported in Chapters 5 and 7.

In this Chapter we explore the development of a TES which can be implemented with small area and power consumption. As a starting point we take an algorithm presented in [128] which aims to design more efficient TES. The focus of [128] was to design a TES which would not require block-cipher decryption. It was argued that such a construction would be more efficient than a construction which requires both encryption and decryption. The construction in [128] uses a Feistel network in a novel way to remove the inverse call of the block-cipher. In [128] it is also mentioned that the basic strategy of using a Feistel network can also be adopted to construct a mode which uses stream ciphers with initialization vector (IV) instead of the block cipher. We further develop the strategy in [128] and propose a new TES which uses stream ciphers with IV and some special universal hash functions. We call the new construction as STES. We formally prove that STES is a secure TES and analyze STES extensively from an implementational perspective. Finally we come up with various hardware designs which offer a very good time /area trade-off. We argue that these architectures can be suitable for encryption of flash memories of various classes and can be deployed in area/power constrained environments.

8.1 Some Technical Preliminaries

The new construction of TES that is reported in this Chapter is fundamentally different from the constructions introduced till now by the fact that they do not use block ciphers, but use stream ciphers. Additionally the hash function that this construction uses is structurally different from a normal or a BRW polynomial. In this Section we give some overview of these basic primitives used for the new construction

8.1.1 Stream Ciphers with IV

Like block ciphers stream ciphers are an important symmetric key primitive. A stream cipher takes as input a key and outputs a "long" stream of random bits. The theoretical object that a stream cipher models is a pseudorandom generator, which (informally) converts a "short" random string to a "long" string which looks random to any efficient adversary. Stream ciphers can be used as stand alone ciphers to do symmetric encryption.

Modern stream ciphers are designed to incorporate initialization vectors (IV), which helps in increasing variability of the ciphertexts. A stream cipher with IV is a function which takes as input two strings, namely, the key and the IV, and produces a stream of random (looking) bits as output.

Let $\mathsf{SC}_K : \{0,1\}^\ell \to \{0,1\}^L$ be a stream cipher with IV, i.e., for every choice of K from a certain pre-defined key space $\mathcal{K}$, SC_K maps a ℓ bit IV to a string of length L bits. The length L is assumed to be long enough for practical sized messages to be encrypted. Actual encryption of a plaintext P is done by XORing the first $|P|$ bits of $\mathsf{SC}_K(IV)$ with P. By $\mathsf{SC}_K^\ell(IV)$ we shall denote the first ℓ bits of the output of $\mathsf{SC}_K(IV)$.

For proving the security of our scheme we we will assume that $\mathsf{SC} : \mathcal{K} \times \{0,1\}^\ell \to \{0,1\}^L$ to be a pseudorandom function family with domain $\{0,1\}^\ell$ and range $\{0,1\}^L$. This is indeed a design goal for modern stream ciphers [127].

8.1.2 Multilinear Universal Hash

A MLUH (Multilinear Universal Hash) with data path d takes in a message $M = M_1||M_2||\cdots||M_m \in \{0,1\}^{dm}$, where each $|M_i| = d$. MLUH produces a bd bit output for some $b \geq 1$. To do this MLUH requires $(m+b-1)d$ bits of key material. Assuming that $K = K_1||K_2||\ldots||K_{m+b-1}$, where each $|K_i| = d$ we define

$$\mathsf{MLUH}_K^{d,b}(M) = h_1||h_2||\cdots||h_b,$$

where

$$\begin{aligned} h_1 &= M_1 \cdot K_1 \oplus M_2 \cdot K_2 \oplus ... \oplus M_m \cdot K_m \\ h_2 &= M_1 \cdot K_2 \oplus M_2 \cdot K_3 \oplus ... \oplus M_m \cdot K_{m+1} \\ &. \\ &. \\ &. \\ h_b &= M_1 \cdot K_b \oplus M_2 \cdot K_{b+1} \oplus .... \oplus M_m \cdot K_{b+m-1}, \end{aligned} \tag{8.1}$$

and the additions and multiplications are in the field $\mathrm{GF}(2^d)$. We will often call the parameter d as the data-path of the MLUH. We define the $\mathsf{MLUH}_K^{d,b}(.)$ in such a way that the valid key lengths and message lengths are always multiples of d. Cases where the message length and/or key length are not multiples of d can be handled with appropriate padding, but it makes the notation and description much complex. Moreover in the application that we have in mind this property regarding the length of the message and the key would always be satisfied.

A MLUH is a xor universal (XU) hash function, more specifically for a randomly chosen key K from the key space $\{0,1\}^{d(b+m-1)}$, and any pair of distinct messages $M_1, M_2 \in \{0,1\}^{md}$ and any $\delta \in \{0,1\}^{db}$,

$$\Pr[\mathsf{MLUH}_K^{d,b}(M_1) \oplus \mathsf{MLUH}_K^{d,b}(M_2) = \delta] \leq \frac{1}{2^{db}}, \tag{8.2}$$

where the probability is taken over the uniform random choice of the key K. See [124] for a proof of this.

A variant of MLUH is the pseudo-dot construction, which we will denote by PD. Assuming m to be even, similar to the MLUH construction, the PD takes in a input $M = M_1||M_2||\cdots||M_m \in \{0,1\}^{dm}$ and a key $K = K_1||K_2||\ldots||K_{m+2b-2}$, where $M_i, K_i \in \{0,1\}^d$, and we define $\mathsf{PD}_K^{db}(M) = h_1||h_2||...||h_b$, where

$$\begin{aligned} h_1 &= (M_1 \oplus K_1)(M_2 \oplus K_2) \oplus (M_3 \oplus K_3)(M_4 \oplus K_4) \oplus ... \oplus (M_{m-1} \oplus K_{m-1})(M_m \oplus K_m) \\ h_2 &= (M_1 \oplus K_3)(M_2 \oplus K_4) \oplus (M_3 \oplus K_5)(M_4 \oplus K_6) \oplus ... \oplus (M_{m-1} \oplus K_{m+1})(M_m \oplus K_{m+2}) \\ &. \\ &. \\ &. \\ h_b &= (M_1 \oplus K_{2b-1})(M_2 \oplus K_{2bb}) \oplus ... \oplus (M_{m-1} \oplus K_{m+2b-3})(M_m \oplus K_{m+2b-2}) \end{aligned} \tag{8.3}$$

The PD function is also a XU hash function [10].

8.2 Construction of STES

The description of the encryption algorithm using STES is given in Figure 8.2 and the block diagram in Figure 8.1. The construction is parameterised by a stream cipher SC supporting ℓ-bit IVs, a hash function MLUH with data path d and a fixed ℓ-bit string fStr. This is emphasized by writing $\mathsf{STES}[\mathsf{SC}, \mathsf{MLUH}, \mathsf{fStr}]$. When one or more of the parameters are clear from the context, we drop these for simplicity of notation; if all three parameters are clear, then we simply write STES. We assume that $d \mid \ell$. Plaintexts and tweaks are fixed length messages. If P is any plaintext and T is any tweak, then we also assume that $d \mid (|P| + |T| - 2\ell)$. For practical implementations, the restrictions on d are easy to ensure as we discuss later.

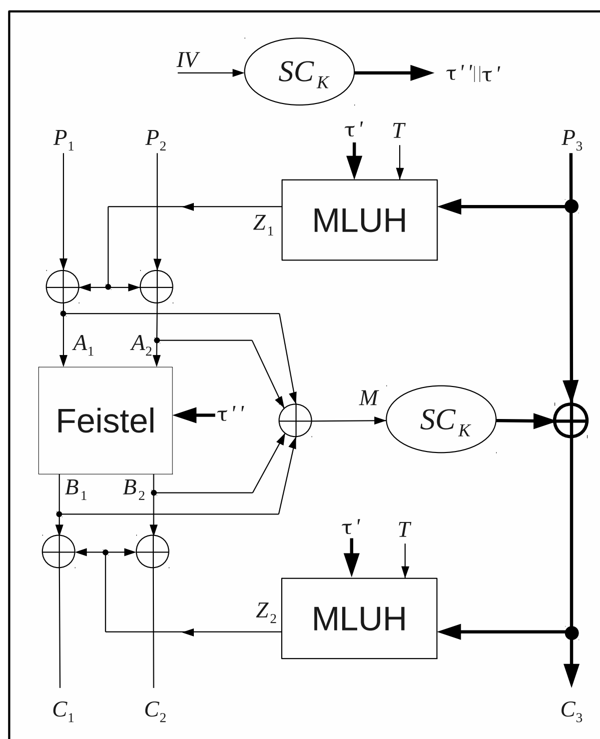


Figure 8.1: Block diagram of STES.

The algorithm takes an IV of length ℓ which is a random element in $\{0,1\}^{\ell}$, and a key K which is again a random element in the key space of the stream cipher. Both IV and K are kept secret. Other than the key and IV it takes in the plain text P and the tweak T.

The secret key for STES is the secret key K of the underlying stream cipher. In this context, we would like to mention the role that the parameter fStr can play. From the point of view of the formal security analysis, there is no restriction on fStr. Thus, this can be used as a secret customisation option. In other words, for actual deployment, one may choose a uniform random value for fStr and keep it secret. This provides an additional layer of obscurity over

Figure 8.2: STES: A TES using a stream cipher and a MLUH.

$\boldsymbol{STES.Encrypt}_K^T(P)$
1. $b \leftarrow \frac{\ell}{d}$;
2. $b_1 \leftarrow \frac{|P|+|T|-2\ell}{d}$;
3. $\ell_1 \leftarrow (b_1 + b - 1)d$;
4. $\ell_2 \leftarrow (2b - 1)d$;
5. $\ell_3 \leftarrow |P| - 2\ell$;

6. $P_1 \leftarrow \mathsf{bits}(P, 1, \ell)$; /* $|P_1| = \ell$) */
7. $P_2 \leftarrow \mathsf{bits}(P, \ell + 1, 2\ell)$; /* $|P_2| = \ell$ */
8. $P_3 \leftarrow \mathsf{bits}(P, 2\ell + 1, |P|)$; /* $|P_2| = \ell_3$ */

9. $\tau \leftarrow \mathsf{SC}_K^{\ell_1+\ell_2+\ell}(\mathsf{fStr})$;
10. $\tau' \leftarrow \mathsf{bits}(\tau, 1, \ell_1)$;
11. $\beta \leftarrow \mathsf{bits}(\tau, \ell_1 + 1, \ell_1 + \ell)$;
12. $\tau'' \leftarrow \mathsf{bits}(\tau, \ell_1 + \ell + 1, \ell_1 + \ell + \ell_2)$;

13. $Z_1 \leftarrow \mathsf{MLUH}_{\tau'}^{d,b}(P_3||T) \oplus \beta$;
14. $A_1 \leftarrow P_1 \oplus Z_1$;
15. $A_2 \leftarrow P_2 \oplus Z_1$;
16. $(B_1, B_2) \leftarrow \mathsf{Feistel}_{K,\tau''}^{\ell,d}(A_1, A_2)$;
17. $M_1 \leftarrow A_1 \oplus B_1$;
18. $M_2 \leftarrow A_2 \oplus B_2$;
19. $M \leftarrow M_1 \oplus M_2$;
20. $C_3 \leftarrow P_3 \oplus \mathsf{SC}_K^{\ell_3}(M)$;
21. $Z_2 \leftarrow \mathsf{MLUH}_{\tau'}^{d,b}(C_3||T) \oplus (\beta \lll 1)$;
22. $C_1 \leftarrow B_1 \oplus Z_2$;
23. $C_2 \leftarrow B_2 \oplus Z_2$;

$\boldsymbol{return}(C_1||C_2||C_3)$;

and above the provable security analysis that we perform. There is another advantage to using fStr as part of the secret key. The security bound that is obtained is in terms of the IV length ℓ and the number of queries for which security holds can be obtained as a function of 2^ℓ. The key length $|K|$ should be at least ℓ for the analysis to be meaningful. If the key length is equal to ℓ, then certain "out of model" attacks may apply as has been pointed out in [32]. Increasing the key length by keeping fStr as part of the secret key will help in preventing such attacks.

Apart from the secret key K, the input to the encryption algorithm of STES is the tweak T

Figure 8.3: Feistel network constructed using a stream cipher and a MLUH.

$\mathsf{Feistel}^{\ell,d}_{K,\tau''}(A_1, A_2)$	$\mathsf{InvFeistel}^{\ell,d}_{K,\tau''}(B_1, B_2)$
1. $b \leftarrow \lceil \frac{\ell}{d} \rceil$	1. $b \leftarrow \lceil \frac{\ell}{d} \rceil$;
2. $H_1 \leftarrow \mathsf{MLUH}^{d,b}_{\tau''}(A_1)$;	2. $H_2 = \mathsf{MLUH}^{d,b}_{\tau''}(B_2)$;
3. $F_1 \leftarrow H_1 \oplus A_2$;	3. $F_2 = H_2 \oplus B_1$;
4. $G_1 \leftarrow \mathsf{SC}^{\ell}_{K}(F_1)$;	4. $G_2 = \mathsf{SC}^{\ell}_{K}(F_2)$;
5. $F_2 \leftarrow A_1 \oplus G_1$;	5. $F_1 = B_1 \oplus G_2$;
6. $G_2 \leftarrow \mathsf{SC}^{\ell}_{K}(F_2)$;	6. $G_1 = \mathsf{SC}^{\ell}_{K}(F_1)$;
7. $B_2 \leftarrow F_1 \oplus G_2$;	7. $A_1 = F_2 \oplus G_1$;
8. $H_2 \leftarrow \mathsf{MLUH}^{d,b}_{\tau''}(B_2)$;	8. $H_1 = \mathsf{MLUH}^{d,b}_{\tau''}(A_1)$;
9. $B_1 \leftarrow H_2 \oplus F_2$;	9. $A_2 \leftarrow H_1 \oplus F_1$;
10. ***return***(B_1, B_2);	10. ***return***(A_1, A_2);

and a plaintext P. Similarly, the input to the decryption algorithm of STES consists of T and the ciphertext C.

The algorithm begins with some length calculations, and fixes values for the variables ℓ_1, ℓ_2 and ℓ_3. ℓ_1 and ℓ_2 contains the key lengths necessary for the MLUH which are called later in the algorithm. Next, it parses the input plaintext into three parts P_1, P_2 and P_3 where P_1 and P_2 are both ℓ bits long and P_3 is $|P| - 2\ell$ bits long. In line 9, $(\ell_1 + \ell_2)$ bits are generated from the stream cipher SC_K using the fStr as input. This output of the stream cipher is again parsed into three strings τ', β and τ'', τ' and τ'' are later used as keys for the MLUH and β is xored with the output of MLUH. The part P_3 of the message and the tweak T is hashed using the MLUH and mixed with the message parts P_1 and P_2 to generate two strings A_1 and A_2. These strings are used as an input to the function Feistel() which is described in Figure 8.3. The function Feistel() receives two keys K and τ'' and it mixes the input strings A_1 and A_2 by appropriate use of the hash MLUH and the stream cipher SC. Note that the function Feistel has a structure similar to the Feistel network, and it is invertible. The inverse function for Feistel is also shown in Figure 8.3. The output of the function Feistel is mixed with its input to create a ℓ bit string M. This M is used as an initialization vector for the stream cipher to generate $|P_3|$ many bits which are XORed with P_3 to get the ciphertext corresponding to P_3, the other parts of the ciphertext C_1 and C_2 are generated using the output of the Feistel and the hash of the cipher C_3 and the tweak T.

The description of the algorithm STES can be modified by using the pseudo dot-product hash PD which is described in Section 8.1.2. If PD is used instead of MLUH the key lengths required are to be suitably changed. For hashing m blocks (where each block is d bits long) of message the PD construction requires $m + 2b - 2$ blocks of keys, where bd is the length of the output. Hence the parameter ℓ_1 in line 4 must be fixed to $b_1 + 2b - 2$ and ℓ_2 to $(4b - 2)d$,

and all the calls to MLUH should be replaced by PD in the algorithm STES and the function Feistel with the same parameters as it appears in the descriptions.

8.2.1 Some Characteristics of the Construction

- **Key lengths:** The only secret values used in the construction is the key K for the stream cipher and the IV. The other keys τ' and τ'' used for the hash are generated using the stream cipher. In certain usages it may be possible to store τ' and τ'' as keys and give them as input to the algorithm. In that case the stream cipher call in line 9 would not be required. But as the key τ' is much larger in size compared to K and IV, hence for most applications it would be convenient to generate it on the fly.

- **Message lengths:** STES works only for fixed length messages. It is possible to extend the construction to accommodate variable length messages but for the application of block wise encryption of flash memories such generalization is not required.

- **Efficiency:** The costly operations that take place in the algorithm are the calls to the stream cipher and the hash functions. In the main body of the algorithm the stream cipher is called twice in lines 9 and 19, and in the function Feistel it is also called twice in lines 4 and 6. It is to be noted that in real life, stream ciphers are quite fast in generation of the outputs, but when a stream cipher is called on a different initialization vector then there is a significant time required for initialization. Note that the four calls to the stream cipher required in STES are all on different initialization vectors. Hence stream cipher initialization occupies a significant amount of the time required for STES. The MLUH and PD can be implemented very efficiently in hardware with a proper choice of the data path d. The choice of d dictates the amount of parallelism possible. Recall that the main goal of the construction is to enable a hardware realization which uses small amount of hardware resources. A proper choice of the stream cipher and the data path can help in realizing a circuit with adequate throughput but with a small hardware footprint. These issues would be discussed in details when we describe in Section 8.4 the hardware realization of a circuit for STES, and we would demonstrate that STES meets the expected efficiency requirements both in terms of time and circuit size.

- **Security:** The main objective of the construction is to achieve the property of a strong pseudorandom permutation (SPRP). This construction do achieves it and we prove it formally in Section 8.3, but here we informally argue that in this construction each cipher bit depends on each plaintext bit and vice versa, which is a necessary property of a SPRP. Note that the main enciphering operation takes place in line 19 of the algorithm with the help of the stream cipher. The initialization vector M used in the stream cipher depends on the whole message, also the parts C_1 and C_2 are mixed

with the hash of the whole message and the tweak making each bit of the ciphertext dependent on each bit of the plaintext.

8.3 Security of STES

The security of tweakable enciphering schemes have been described in Section 4.2. To prove that a TES is secure in the SPRP sense we show that the advantage of any adversary in distinguishing the TES from a uniform random tweak indexed permutation is small.

The following theorem specifies the security of STES.

Theorem 8.1. *Let* $\delta \xleftarrow{\$} \mathsf{Func}(\ell, L)$ *and* $\mathsf{STES}[\delta]$ *be STES instantiated with the function* δ *in place of the stream cipher. Then, for any arbitrary adversary* $\mathcal{A}$ *which asks at most* q *queries we have*

$$\mathbf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathsf{STES}[\mathsf{Func}(\ell,L)]}(\mathcal{A}) \leq \frac{10q^2+3q}{2^\ell}. \tag{8.4}$$

The theorem guarantees that if the stream cipher acts like a random function then, for any arbitrary adversary who asks a reasonable number of queries the advantage of distinguishing $\mathsf{STES}[\delta]$ and a uniform random tweak indexed permutation would be small.

The proof of the theorem consists of a standard game transition argument and a combinatorial analysis of some collision probabilities, we provide the full proof in the next subsection.

The bound in Eq. (8.4) decreases with increase in ℓ. Thus the bound would be better when stream ciphers with lager IVs are used.

8.3.1 Proof of Theorem 8.1

In the proof, the string fStr will be fixed and so we will not explicitly mention this as a parameter to STES. The analysis of the proof will be done assuming the hash function H to be MLUH. Essentially the same analysis also holds when H is instantiated as MLUH.

Let δ be a uniform random function from $\{0,1\}^\ell$ to $\{0,1\}^L$, i.e., δ is chosen uniformly at random from the set of all functions from $\{0,1\}^\ell$ to $\{0,1\}^L$. This means that for distinct inputs $X_1, \ldots, X_q$, the values $\delta(X_1), \ldots, \delta(X_q)$ are independent and uniform random. This property will be used in the argument below. In the first step of the proof, the stream cipher SC is replaced by δ. Note that this is only a conceptual step and there is no need to obtain the acutal construction. In fact, there is no need to even chose the whole of δ and its behaviour can be simulated in an incremental fashion as follows. Keep a list of hitherto generated pairs of inputs and outputs; for any input, one needs to check whether it has already occurred and if so, return the corresponding output, if not, return an independent and uniform random string.

Denote by $\mathsf{STES}[\delta, \mathsf{H}]$ the corresponding construction. It is quite standard to argue that the following inequality holds.

$$\mathsf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathsf{STES}[\mathsf{SC},\mathsf{H}]}(t,q,\sigma) \;\leq\; \mathsf{Adv}^{\mathrm{prf}}_{\mathsf{SC}}(t',q,\sigma) + \mathsf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathsf{STES}[\delta,\mathsf{H}]}(t,q,\sigma). \tag{8.5}$$

The idea is that if there is an adversary which can distinguish between $\mathsf{STES}[\mathsf{SC}]$ and $\mathsf{STES}[\delta]$, then that adversary can be used to distinguish between SC and a uniform random function and so breaks the PRF-property of SC. The parameters q and σ carry over directly whereas the parameter t increases to t' since during the simulation, each query has to be processed using either the encryption or the decryption algorithm of $\mathsf{STES}[\mathsf{SC}, \mathsf{H}]$.

The construction $\mathsf{STES}[\delta]$ is parameterized by the uniform random function δ. Unlike SC, there is no computational assumption on δ. Basically, the computational aspect is taken care of by the bound on the PRF-advantage of SC. So, the rest of the proof poceeds in an information theoretic manner. The time parameter in $\mathsf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathsf{STES}[\delta]}(t,q,\sigma)$ is redundant, since allowing unlimited time does not help the adversary. So, we drop t and use the notation $\mathsf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathsf{STES}[\delta]}(q,\sigma)$. The main part of the proof is to show the following.

$$\mathsf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathsf{STES}[\delta]}(t,q,\sigma) \;\leq\; \frac{10q^2+3q}{2^{\ell}}. \tag{8.6}$$

We have (see [65])

$$\mathsf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathsf{TES}[\delta]}(q,\sigma) \leq \mathsf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{TES}[\delta]}(\sigma) + \binom{q}{2}\frac{1}{2^{\ell}}. \tag{8.7}$$

So, the task reduces to upper bounding $\mathsf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{TES}[\delta]}$. In other words, this is to show that the advantage of an adversary in distinguishing $\mathsf{STES}[\mathsf{SC}]$ from oracles which simply return independent and uniform random strings is small. The proof now proceeds via a sequence of games as described below.

In each game, the adversary $\mathcal{A}$ makes a total of q queries. For convenience of description, we introduce a variable ty^s for each $s = 1, \ldots, q$. The value of $ty^s = enc$ (resp. $ty^s = dec$) denotes the corresponding query to be an encryption (resp. decryption) query. At the end of each game, the adversary outputs a bit. By $\Pr[\mathcal{A}^{\mathbf{G}} \Rightarrow 1]$ we denote the event that the adversary outputs 1 in Game G where G is one of **G0**, **G1** or **G2**.

The first game **G0** is depicted in Figure 8.4. Game **G0** is just the rewrite of the algorithm of STES in Fig. 8.2, but we replace the stream cipher SC by a uniform random function δ. The random function is constructed on the fly using the subroutine $\delta()$, which is also shown in Figure 8.4. The subroutine $\delta()$ maintains a table T, indexed on the strings in $\{0,1\}^{\ell}$ and is initially undefined everywhere. In the table $T[\,]$ the subroutine keeps information regarding the values returned by it corresponding to the inputs. When called on an input $X \in \{0,1\}^{\ell}$, $\delta()$ checks if $T[X]$ is undefined; if so, then it returns a random string in $\{0,1\}^{L}$ and stores

the returned value in $T[X]$; otherwise, it returns $T[X]$ and sets a flag labeled bad to true. Note that game **G0** is a perfect simulation of STES instantiated with the random function δ. Hence if $\mathcal{A}$ is the adversary interacting with **G0**, and if we denote the encryption and decryption procedures of STES$[\delta]$ as Π_δ and Π_δ^{-1} respectively, then we have

$$\Pr[\mathcal{A}^{\mathbf{G0}} \Rightarrow 1] = \Pr[\mathcal{A}^{\Pi_\delta(.,.),\Pi_\delta^{-1}(.,.)} \Rightarrow 1]. \tag{8.8}$$

```
Subroutine δ(X)
01.   Y ←$ {0,1}^L;
02.   if X ∈ D then bad ← true; [Y ← T[X]] ; end if;
03.   T[X] ← Y; D ← D ∪ {X};
04.   return Y;
Initialization:
   for all X ∈ {0,1}^ℓ, T[X] ← undef; endfor
   bad ← false; D ← {fStr};
      τ ← δ(fStr);
      τ' ← bits(τ, 1, ℓ1);
      β ← bits(τ, ℓ1 + 1, ℓ1 + ℓ);
      τ'' ← bits(τ, ℓ1 + ℓ + 1, ℓ1 + ℓ + ℓ2);

Feistel_{K,τ''}(A1^s, A2^s)
201.  b^s ← ⌈ℓ/d⌉
202.  H1^s ← MLUH^{d,b}_{τ''}(A1^s);
203.  F1^s ← H1^s ⊕ A2^s;
204.  G1^s ← bits(δ(F1^s), ℓ);
205.  F2^s ← A1^s ⊕ G1^s;
206.  G2^s ← bits(δ(F2^s), ℓ);
207.  B2^s ← F1^s ⊕ G2^s;
208.  H2^s ← MLUH^{d,b}_{τ''}(B2^s);
209.  B1^s ← H2^s ⊕ F2^s;
210.  return(B1^s, B2^s);

InvFeistel_{K,τ''}(B1^s, B2^s)
201.  b^s ← ⌈ℓ/d⌉;
202.  H2^s = MLUH^{d,b}_{τ''}(B2^s);
203.  F2^s = H2^s ⊕ B1^s;
204.  G2^s = bits(δ(F2^s), ℓ);
205.  F1^s = B1^s ⊕ G2^s;
206.  G1^s = bits(δ(F1^s), ℓ);
207.  A1^s = F2^s ⊕ G1^s;
208.  H1^s = MLUH^{d,b}_{τ''}(A1^s);
209.  A2^s ← H1^s ⊕ F1^s;
210.  return(A1^s, A2^s);

Response to the s^th query:

Case ty^s = enc:
100.  P1^s ← bits(P^s, 1, ℓ); /* |P1| = ℓ) */
101.  P2^s ← bits(P^s, ℓ + 1, 2ℓ); /* |P2| = ℓ */
102.  P3^s ← bits(P^s, 2ℓ + 1, |P|); /* |P2| = ℓ3 */

103.  Z1^s ← MLUH^{d,b}_{τ'}(P3^s||T^s) ⊕ β;
104.  A1^s ← P1^s ⊕ Z1^s;
105.  A2^s ← P2^s ⊕ Z1^s;
106.  (B1^s, B2^s) ← Feistel^{ℓ,d}_{K,τ''}(A1^s, A2^s);
107.  M1^s ← A1^s ⊕ B1^s;
108.  M2^s ← A2^s ⊕ B2^s;
109.  M^s ← M1^s ⊕ M2^s;
110.  C3^s ← P3^s ⊕ bits(δ(M^s), ℓ3);
111.  Z2^s ← MLUH^{d,b}_{τ'}(C3^s||T^s) ⊕ (β ⋘ 1);
112.  C1^s ← B1^s ⊕ Z2^s;
113.  C2^s ← B2^s ⊕ Z2^s;
return(C1^s||C2^s||C3^s);

Case ty^s = dec:
100.  C1^s ← bits(C^s, 1, ℓ); /* |P1| = ℓ) */
101.  C2^s ← bits(C^s, ℓ + 1, 2ℓ); /* |P2| = ℓ */
102.  C3^s ← bits(C^s, 2ℓ + 1, |P|); /* |P2| = ℓ3 */

103.  Z2^s ← MLUH^{d,b}_{τ'}(C3^s||T^s) ⊕ (β ⋘ 1);
104.  B1^s ← C1^s ⊕ Z1^s;
105.  B2^s ← C2^s ⊕ Z1^s;
106.  (A1^s, A2^s) ← InvFeistel^{ℓ,d}_{K,τ''}(A1^s, A2^s);
107.  M1^s ← B1^s ⊕ A1^s;
108.  M2^s ← B2^s ⊕ A2^s;
109.  M^s ← M1^s ⊕ M2^s;
110.  P3^s ← C3^s ⊕ bits(δ(M^s), ℓ3);
111.  Z1^s ← MLUH^{d,b}_{τ'}(P3^s||T^s) ⊕ β;
112.  P1^s ← A1^s ⊕ Z1^s;
113.  P2^s ← A2^s ⊕ Z1^s;
return(P1^s||P2^s||P3^s);
```

Figure 8.4: Games **G0** and **G1**. The full description is of Game **G0**; Game **G1** is obtained by removing the boxed entry in Line **02**.

We change game **G0** to game **G1** by eliminating the boxed entry in game **G0**. **G1** is shown in Figure 8.4. With this change, the games **G0** and **G1** executes in the same manner unless the bad flag is set to true. Hence, using the difference Lemma (see [8, 133])

$$|\Pr[\mathcal{A}^{\mathbf{G0}} \Rightarrow 1] - \Pr[\mathcal{A}^{\mathbf{G1}} \Rightarrow 1]| \leq \Pr[\mathcal{A}^{\mathbf{G1}} \text{ sets } \mathsf{bad}]. \tag{8.9}$$

In Game **G1**, the responses received by $\mathcal{A}$ are random strings as C_1, C_2 and C_3 are all outputs of $\delta()$ xor-ed with other independent strings. Also, in Game **G1**, $\delta()$ responds with random strings irrespective of the inputs it receives.

Now, we do a purely syntactic change to **G1** to obtain **G2** which is shown in Figure 8.5. In **G2**, when an encryption or decryption query from $\mathcal{A}$ is received, a random string of the length equal to that of the message/cipher length is returned immediately. After all the q queries of the adversary have been answered, the game enters the finalization phase. The finalization of **G2** runs in two phases. In the first phase, based on the query and the response, the internal random variables in the algorithm are adjusted and these values are inserted in the $\mathcal{D}$. In Phase 2, if there is a collision within $\mathcal{D}$, i.e., two random variables in $\mathcal{D}$ take the same value, then the bad flag is set to true.

As **G1** and **G2** provide the same view to the adversary and differ only in the way they are written, we have

$$\Pr[\mathcal{A}^{\mathbf{G1}} \Rightarrow 1] = \Pr[\mathcal{A}^{\mathbf{G2}} \Rightarrow 1]. \tag{8.10}$$

and

$$\Pr[\mathcal{A}^{\mathbf{G1}} \text{ sets } \mathsf{bad}] = \Pr[\mathcal{A}^{\mathbf{G2}} \text{ sets } \mathsf{bad}]. \tag{8.11}$$

Moreover, when $\mathcal{A}$ interacts with **G2** it gets random strings as responses to all its queries, and so

$$\Pr[\mathcal{A}^{\mathbf{G2}} \Rightarrow 1] = \Pr[\mathcal{A}^{\$(.,.),\$(.,.)} \Rightarrow 1]. \tag{8.12}$$

Hence, using (8.9), (8.10), (8.11) and (8.12), we have

$$\begin{aligned} \mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{STES}[\delta]}(\mathcal{A}) &= \Pr[\delta \xleftarrow{\$} \mathsf{Func}(\ell, L) : \mathcal{A}^{\Pi_\delta(.,.),\Pi_\delta^{-1}(.,.)} \Rightarrow 1] - \Pr[\mathcal{A}^{\$(.,.),\$(.,.)} \Rightarrow 1] \\ &\leq \Pr[\mathcal{A}^{\mathbf{G2}} \text{ sets } \mathsf{bad}]. \end{aligned} \tag{8.13}$$

8.3.2 Collision Analysis

The rest of the proof is devoted to computing a bound on $\Pr[\mathcal{A}^{\mathbf{G2}} \text{ sets } \mathsf{bad}]$. Here $\mathcal{A}$ is an arbitrary adversary which asks q queries each consisting of a message/cipher of length $m\ell$ bits and a tweak of ℓ bits. If COLLD denotes the event that there is a collision in $\mathcal{D}$ as

Initialization:

$\mathcal{D} \xleftarrow{\$} \{\mathsf{fStr}\}$; $\tau' \xleftarrow{\$} \{0,1\}^{\ell_1}$; $\tau'' \xleftarrow{\$} \{0,1\}^{\ell_2}$; $\beta \xleftarrow{\$} \{0,1\}^{\ell}$;

Response to the sth query:

Case $ty^s = enc$**:**	**Case** $ty^s = dec$**:**
101. $C^s \xleftarrow{\$} \{0,1\}^{\lvert P \rvert}$	101. $P^s \xleftarrow{\$} \{0,1\}^{\lvert C \rvert}$
102. $C_1^s \leftarrow \mathsf{bits}(C^s, 1, \ell)$;	102. $P_1^s \leftarrow \mathsf{bits}(P^s, 1, \ell)$;
103. $C_2^s \leftarrow \mathsf{bits}(C^s, \ell+1, 2\ell)$;	103. $P_2^s \leftarrow \mathsf{bits}(P^s, \ell+1, 2\ell)$;
104. $C_3^s \leftarrow \mathsf{bits}(C^s, 2\ell+1, \lvert P \rvert)$;	104. $P_3^s \leftarrow \mathsf{bits}(P^s, 2\ell+1, \lvert C \rvert)$;
return$(C_1^s \| C_2^s \| C_3^s)$;	***return***$(P_1^s \| P_2^s \| P_3^s)$;

Finalization:

FIRST PHASE

for $s \leftarrow 1$ to q **do**

  Case $ty^s = enc$**:**

    $F_1^s \leftarrow P_2^s \oplus \beta \oplus \mathsf{MLUH}_{\tau'}^{d,b}(P_3^s \| T^s) \oplus$
        $\mathsf{MLUH}_{\tau''}^{d,b}(P_1^s \oplus \beta \oplus \mathsf{MLUH}_{\tau'}^{d,b}(P_3^s \| T^s))$;

    $F_2^s \leftarrow C_1^s \oplus (\beta \lll 1) \oplus \mathsf{MLUH}_{\tau'}^{d,b}(C_3^s \| T) \oplus$
        $\mathsf{MLUH}_{\tau''}^{d,b}(C_2^s \oplus (\beta \lll 1) \oplus \mathsf{MLUH}_{\tau'}^{d,b}(C_3^s \| T^s))$;

    $M^s \leftarrow P_1^s \oplus P_2^s \oplus C_1^s \oplus C_2^s$;

    $\mathcal{D} \leftarrow \mathcal{D} \cup \{F_1^s\} \cup \{F_2^s\} \cup \{M^s\}$;

  Case $ty^s = dec$**:**

    $F_2^s \leftarrow C_1^s \oplus \mathsf{MLUH}_{\tau'}^{d,b}(C_3^s \| T^s) \oplus$
        $\mathsf{MLUH}_{\tau''}^{d,b}(C_2^s \oplus \mathsf{MLUH}_{\tau'}^{d,b}(C_3^s \| T^s))$;

    $F_1^s \leftarrow P_2^s \oplus \mathsf{MLUH}_{\tau'}^{d,b}(P_3^s \| T) \oplus$
        $\mathsf{MLUH}_{\tau''}^{d,b}(P_1^s \oplus \mathsf{MLUH}_{\tau'}^{d,b}(P_3^s \| T^s))$;

    $M^s \leftarrow P_1^s \oplus P_2^s \oplus C_1^s \oplus C_2^s$;

  $\mathcal{D} \leftarrow \mathcal{D} \cup \{F_1^s\} \cup \{F_2^s\} \cup \{M^s\}$;

endfor;

SECOND PHASE

$\mathsf{bad} \leftarrow \mathsf{false}$;

if (two variables in $\mathcal{D}$ are equal) **then** $\mathsf{bad} \leftarrow \mathsf{true}$;

Figure 8.5: Game **G2**.

described in the Game **G2**, then

$$\Pr[\mathcal{A}^{\mathbf{G2}} \text{ sets } \mathsf{bad}] = \Pr[\mathsf{COLLD}]. \tag{8.14}$$

We shall now concentrate on finding an upper bound for $\Pr[\mathsf{COLLD}]$. The following simple result states a property of rotation that we will require later.

Lemma 8.1. *If β is chosen uniformly at random from $\{0,1\}^{\ell}$ and X is any ℓ-bit string, then* $\Pr[\beta \oplus (\beta \lll 1) = X] = 1/2^{n-1}$.

The set $\mathcal{D}$ is as follows:

$$\mathcal{D} = \{F_1^s, F_2^s, M^s : 1 \leq s \leq q\} \cup \{\mathsf{fStr}\}.$$

All variables in $\mathcal{D}$ are distributed over $\{0,1\}^{\ell}$. As mentioned earlier, the proof is using $\mathsf{MLUH}_{\tau}^{d,b}(X)$ to instantiate $H_{\tau}(X)$, Essentially the same arguments hold when the PD con-

struction is used to instantiate H. The variables in $\mathcal{D}$ can be expressed in terms of the plaintext and ciphertext blocks as follows.

$$\begin{aligned}
F_1^s &= P_2^s \oplus \beta \oplus H_{\tau'}(P_3^s||T^s) \oplus H_{\tau''}(P_1^s \oplus \beta \oplus H_{\tau'}(P_3^s||T^s)), \\
F_2^s &= C_1^s \oplus (\beta \lll 1) \oplus H_{\tau'}(C_3^s||T^s) \oplus H_{\tau''}(C_2^s \oplus (\beta \lll 1) \oplus H_{\tau'}(C_3^s||T^s)), \\
M^s &= P_1^s \oplus P_2^s \oplus C_1^s \oplus C_2^s.
\end{aligned}$$

Noting the following points will help in following the analysis.

1. In Game **G2**, the string $\tau = \tau'||\tau''||\beta$ is selected uniformly at random and is independent of all other variables.

2. If $ty^s = enc$, then (C_1^s, C_2^s, C_3^s) is uniform and independent of all other variables; if $ty^s = dec$, then (P_1^s, P_2^s, P_3^s) is uniform and independent of all other variables.

3. In response to each query, $\mathcal{A}$ receives either (C_1^s, C_2^s, C_3^s) or (P_1^s, P_2^s, P_3^s). These variables are independent of τ and so the queries made by $\mathcal{A}$ are also independent of τ.

Using the randomness of β, or of P_i^s, or of C_j^s, the following immediately holds.

Claim 8.1. *For any* $X \in \mathcal{D} \setminus \{\mathsf{fStr}\}$, $\Pr[X = \mathsf{fStr}] = \frac{1}{2^\ell}$.

This disposes off the cases of fStr colliding with any variable in $\mathcal{D}$. From the second point mentioned above, the following result is easily obtained.

Claim 8.2. *1. For any pair of queries* s, t,
$\Pr[M^s = M^t : s \neq t] = \Pr[M^s = F_1^t] = \Pr[M^s = F_2^t] = \frac{1}{2^\ell}$.

2. Let s, t *be queries such that at least one is a decryption query and* $s \neq t$, *then*
$\Pr[F_1^s = F_1^t] = \Pr[F_2^s = F_2^t] = \frac{1}{2^\ell}$.

3. If the sth *query is an encryption query or the* t^{th} *query is a decryption query, then*
$\Pr[F_2^s = F_1^t] = \frac{1}{2^\ell}$.

By the above claims, we are left only with the following cases to settle.

1. $s \neq t$, $ty^s = ty^t = enc$ and possible collision between F_1^s and F_1^t.

2. $s \neq t$, $ty^s = ty^t = dec$ and possible collision between F_2^s and F_2^t.

3. $ty^s = enc$, $ty^t = dec$ and possible collision between F_1^s and F_2^t.

These are settled by the following two claims.

Claim 8.3. *1. If $s \neq t$ and $ty^s = ty^t = enc$, then* $\Pr[F_1^s = F_1^t] \leq \frac{1}{2^{\ell-1}}$.

2. If $s \neq t$ and $ty^s = ty^t = dec$, then $\Pr[F_2^s = F_2^t] \leq \frac{1}{2^{\ell-1}}$.

Proof. We provide the details of only the first point, the second point being similar. Consider two encryption queries (P^s, T^s) and (P^t, T^t), where $P^s = P_1^s||P_2^s||P_3^s$ and $P^t = P_1^t||P_2^t||P_3^t$. Then $(P^s, T^s) \neq (P^t, T^t)$ as $\mathcal{A}$ is not allowed to repeat queries. Recall

$$\begin{array}{ll} A_1^s = P_1^s \oplus \beta \oplus H_{\tau'}(P_3^s||T_3^s); & A_2^s = P_2^s \oplus \beta \oplus H_{\tau'}(P_3^s||T_3^s); \\ F_1^s = A_2^s \oplus H_{\tau''}(A_1^s); & F_1^t = A_2^t \oplus H_{\tau''}(A_1^t). \end{array}$$

We compute as follows.

$$\begin{aligned} \Pr[F_1^s = F_1^t] &= \Pr[F_1^s = F_1^t | A_1^s = A_1^t] \Pr[A_1^s = A_1^t] + \Pr[F_1^s = F_1^t | A_1^s \neq A_1^t] \Pr[A_1^s \neq A_1^t] \\ &\leq \Pr[F_1^s = F_1^t | A_1^s = A_1^t] + \Pr[F_1^s = F_1^t | A_1^s \neq A_1^t] \\ &= \Pr[F_1^s = F_1^t | A_1^s = A_1^t] + \Pr[H_{\tau''}(A_1^s) \oplus H_{\tau''}(A_1^t) = A_2^s + A_2^t | A_1^s \neq A_1^t] \\ &\leq \Pr[F_1^s = F_1^t | A_1^s = A_1^t] + \frac{1}{2^\ell} \end{aligned} \tag{8.15}$$

The last inequality follows from the xor universality of H. From the expressions for F_1^s and F_1^t, it follows that $\Pr[F_1^s = F_1^t | A_1^s = A_1^t] = \Pr[A_2^s = A_2^t]$. The computation of $\Pr[A_2^s = A_2^t]$ has two cases:

Case 1: $P_3^s||T^s \neq P_3^t||T^t$:

$$\begin{aligned} \Pr[F_1^s = F_1^t | A_1^s = A_1^t] &= \Pr[A_2^s = A_2^t] \\ &= \Pr[H_{\tau'}(P_3^s||T^s) \oplus H_{\tau'}(P_3^t||T^t) = P_1^s \oplus P_1^t] \\ &\leq \frac{1}{2^\ell}. \end{aligned}$$

The last inequality again follows from the xor universality of H.

Case 2: $P_3^s||T^s = P_3^t||T^t$:
As $A_1^s = A_1^t$, this implies that $P_1^s = P_1^t$. Further, since the queries must be distinct, it follows that $P_2^s \neq P_2^t$. So, we have $\Pr[F_1^s = F_1^t | A_1^s = A_1^t] = \Pr[A_2^s = A_2^t] = \Pr[P_2^s = P_2^t] = 0$.

Hence, in both cases, we have $\Pr[F_1^s = F_1^t | A_1^s = A_1^t] \leq 1/2^\ell$. Substituting this value in (8.15) the claim follows. □

So far, we have used the randomness of τ' to argue about the xor universality of H. The role of τ'' and that of β becomes clear in the following result.

Claim 8.4. *If $ty^s = enc$ and $ty^t = dec$, then* $\Pr[F_1^s = F_2^t] \leq \frac{1}{2^{\ell-1}}$.

Proof. We consider an encryption query (P^s, T^s) and a decryption query (C^t, T^t), where $P^s = P_1^s||P_2^s||P_3^s$ and $C^t = C_1^t||C_2^t||C_3^t$.

Let $\mathsf{rest}_1 = P_1^s \oplus H_{\tau'}(P_3^s||T^s)$ and $\mathsf{rest}_2 = C_2^s \oplus H_{\tau'}(C_3^s||T^s)$. Recall that F_1^s and F_2^t can be written in terms of the plaintext and ciphertext variables in the following manner.

$$\begin{aligned} F_1^s &= P_2^s \oplus \beta \oplus H_{\tau'}(P_3^s||T^s) \oplus H_{\tau''}(P_1^s \oplus \beta \oplus H_{\tau'}(P_3^s||T^s)) \\ &= P_2^s \oplus \beta \oplus H_{\tau'}(P_3^s||T^s) \oplus H_{\tau''}(\beta \oplus \mathsf{rest}_1); \\ F_2^t &= C_1^t \oplus (\beta \lll 1) \oplus H_{\tau'}(C_3^t||T^s) \oplus H_{\tau''}(C_2^t \oplus (\beta \lll 1) \oplus H_{\tau'}(C_3^t||T^t)) \\ &= C_1^t \oplus (\beta \lll 1) \oplus H_{\tau'}(C_3^t||T^t) \oplus H_{\tau''}((\beta \lll 1) \oplus \mathsf{rest}_2). \end{aligned}$$

Let $X = \beta \oplus \mathsf{rest}_1$, $Y = (\beta \lll 1) \oplus \mathsf{rest}_2$ and $Z = P_2^s \oplus \beta \oplus H_{\tau'}(P_3^s||T^s) \oplus C_1^t \oplus (\beta \lll 1) \oplus H_{\tau'}(C_3^t||T^t)$. So,

$$\begin{aligned} \Pr[F_1^s = F_2^t] &= \Pr[H_{\tau'}(X) \oplus H_{\tau'}(Y) = Z] \\ &\leq \Pr[X = Y] + \Pr[H_{\tau''}(X) \oplus H_{\tau'}(Y) = Z | X \neq Y] \\ &\leq \Pr[X = Y] + \frac{1}{2^\ell}. \end{aligned}$$

The last inequality follows from the xor universality of H with τ'' as the key. The event $X = Y$ is equivalent to $\beta \oplus (\beta \lll 1) = \mathsf{rest}_1 \oplus \mathsf{rest}_2$. By Lemma 8.1, this holds with probability $1/2^\ell$. From this the claim follows. □

Based on the Claims 8.1 to 8.4, we can conclude that for distinct $X, Y \in \mathcal{D}$, $\Pr[X = Y] \leq 1/2^{\ell-1}$. As $|\mathcal{D}| = 3q + 1$, by the union bound,

$$\Pr[\mathsf{COLLD}] \leq \binom{3q+1}{2} \frac{1}{2^{\ell-1}} = \frac{9q^2 + 3q}{2^\ell}. \tag{8.16}$$

Using (8.7), (8.13), (8.14) and (8.16)

$$\mathbf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathsf{STES}[\delta]}(\mathcal{A}) \leq \frac{9q^2 + 3q}{2^\ell} + \binom{q}{2} \frac{1}{2^\ell} = \frac{10q^2 + 3q}{2^\ell}.$$

Since $\mathcal{A}$ is an arbitrary adversary making q queries, the theorem follows. □

8.4 Hardware Implementation of STES

8.4.1 Basic Design Decisions

We implemented various architectural variants of STES. The main goal of all the designs is to obtain circuits which utilizes minimal hardware and power resources, but still we want to obtain reasonable throughput so that it matches the target application of block-wise flash

memory encryption. The important design decisions are described next.

1. **Message Lengths:** The designs presented here are all meant for fixed length messages, in particular we consider messages which are 512 byte long. This particular size matches the current size of memory blocks. The design philosophies are quite general and can be scaled suitably for other message lengths. For the application we consider, the generality of variable message lengths is not required.

2. **The Stream Ciphers:** The basic building blocks of the algorithm are a stream cipher and a xor universal hash function. In the algorithm STES presented in Figure 8.2 it is possible to plug in any secure stream cipher. For the implementations we choose three different instances: Grain128 [67], Trivium [21] and Mickey128 2.0 [5]. As these are the eStream finalists of hardware based stream ciphers and there are many works in the existing literature which reports compact hardware implementations of these ciphers [16, 52, 59]. There can be various ways to implement these ciphers with varying amount of hardware cost. In particular, Grain128 and Trivium is amenable to parallelization and one can adopt strategies to design hardware which can give an output of only one bit per clock as in [16] or exploit the parallelization and increase the data-path to give more throughput at the cost of more hardware as in [67] and [21]. For instantiations with Grain128 and Trivium we tried different data paths for the stream ciphers and thus realized multiple implementations which provides a wide range of throughput. As said in [70] there exist no trivial way to parallelize Mickey, hence the instantiations with Mickey are all with a data path of one bit. The various parameters considered for implementing the stream ciphers are depicted in Table 8.1.

3. **The Multipliers:** Other than the stream cipher the other important component of the algorithm is the universal hash function MLUH. The main component required to implement the MLUH are finite field multipliers. In Section 8.1.2 we describe MLUH parameterized on the data path d, which signifies that the multiplications in a MLUH with data path d takes place in the field $GF(2^d)$. We consider data paths of 4 bits, 8 bits, 16 bits, 32 bits and 40 bits, the corresponding irreducible polynomials used for the implementations are provided in Table 8.2. The number of multipliers used for implementing the MLUH for each data path varies.

4. **Target FPGA:** We target our designs for Xilinx Spartan 3 and Lattice ICE40 FPGAs, as these are considered suitable for implementing hardware/power constrained designs and moreover they are cheap and one can consider deploying these FPGAs directly into a commercial device. The basic architectural overviews of these families have already been presented in Chapter 3. In particular in Spartan III the LUTs within SLICEM can be used as a 16×1-bit distributed RAM or as a 16-bit shift register (SRL16 primitive). This functionality of a 16-bit shift register have been previously exploited to achieve compact implementations of stream ciphers [16] and we also do so.

The ICE40 FPGAs though do not provide such functionalities, but their architectural design specifically supports low power implementations and our experimental results also suggest that they are much more competitive in this respect compared to Spartan 3 devices.

Field	$\|IV\|$ **(bits)**	$\|K\|$ **(bits)**	**Data paths used (bits)**
Trivium	80	80	$1, 4, 8, 16, 40, 64$
Grain128	96	128	$1, 4, 8, 16, 32$
Mickey128-2.0	96	128	1

Table 8.1: Specific parameters used to implement Trivium, Grain128 and Mickey128 2.0.

Field	**Irreducible Polynomial**	**Field**	**Irreducible Polynomial**
$GF(2^4)$	$x^4 + x + 1$	$GF(2^{32})$	$x^{32} + x^7 + x^3 + x + 1$
$GF(2^8)$	$x^8 + x^4 + x^3 + x + 1$	$GF(2^{40})$	$x^{40} + x^5 + x^4 + x^3 + 1$
$GF(2^{16})$	$x^{16} + x^5 + x^3 + x + 1$	$GF(2^{64})$	$x^{64} + x^5 + x^4 + x^3 + 1$

Table 8.2: Irreducible Polynomials.

8.4.2 Implementation of Universal Hash

An important part of the STES is the xor universal hash function. In this work two different kind of hash functions were implemented: Multilinear Universal Hash (MLUH) and Pseudo dot Product (PD). The basic component of both design is a finite field multiplier. The description of both MLUH and PD as provided in Section 8.1.2 is parameterized by d which denotes that the multiplications are performed in the field $GF(2^d)$ and the size of the output of the hash functions is same as the size of the initialization vector of the stream cipher which we denote as ℓ. We considered various values of d with the restriction that $d|\ell$. Let $b = \ell/d$, for convenience of exposition we rewrite the description of $\mathsf{MLUH}^d_K(M)$ (which is already described in Eq. 8.1)

$$
\begin{aligned}
h_1 &= M_1 \cdot K_1 \oplus M_2 \cdot K_2 \oplus ... \oplus M_m \cdot K_m \\
h_2 &= M_1 \cdot K_2 \oplus M_2 \cdot K_3 \oplus ... \oplus M_m \cdot K_{m+1} \\
&. \\
&. \\
h_b &= M_1 \cdot K_b \oplus M_2 \cdot K_{b+1} \oplus .... \oplus M_m \cdot K_{b+m-1}. \qquad (8.17)
\end{aligned}
$$

Our basic strategy of computing the MLUH is to apply b different multipliers, i.e., when d becomes larger the number of multipliers required becomes smaller. We compute column-wise, i.e., we begin by performing the b multiplications $M_1 \cdot K_1, M_1 \cdot K_2, \ldots, M_1 \cdot K_b$, these can be done in parallel as we are using b multipliers. We store the results of these multiplications separately and then in the next step we compute the products $M_2 \cdot K_2, M_2 \cdot K_3, \ldots, M_2 \cdot K_{b+1}$, again in parallel and these results are xor-ed with the previous results. We continue this until we have computed all the columns. We showcase this strategy of computing the MLUH when $d = 8$ and $\ell = 80$ with a specific architecture next.

The architecture of MLUH for $d = 8$, and $\ell = 80$ (which makes $b = 10$) is shown in Figure 8.6. Hence our design uses 10 multipliers, and each multiplier can multiply two elements in $GF(2^8)$. The architecture is composed of ten 8-bit registers, ten multipliers and ten 8-bit accumulators. All the registers are connected in cascade forming a 10-stage first in first out (FIFO) structure with parallel access to all states. In Figure 8.6 the registers are labeled as **regk1**, **regk2**,...,**regk10**. These registers are used to store ten 8-bit blocks of the key. Each multiplier takes one of its input from FIFO and the other directly from the input line m_i. Initially all registers in the FIFO and accumulators have zero value. The FIFO is fed with the key blocks $K_1, K_2, \ldots,$ etc., one in each cycle through the input line depicted k_i in the figure. After ten clock cycles the FIFO is full, i.e, the registers contains the key blocks $K_1, K_2, \ldots, K_{10}$ respectively and the input line m_i contains the message block M_1 and the multiplications in the first column of MLUH is performed, then each product is accumulated in the respective accumulators. In the next clock, the FIFO contains the key blocks $K_2, \ldots K_{11}$ and the input line m_i contains the message block M_2, and thus the second column of multiplications are computed and these results are accumulated in the respective registers. This is continued until all the columns have been processed. The final output of MULH is obtained by concatenating the final values in the accumulators. The control unit is not shown in the figure, it consist of a counter and some comparators. When the computation of the hash has finished a ready signal is put into 1.

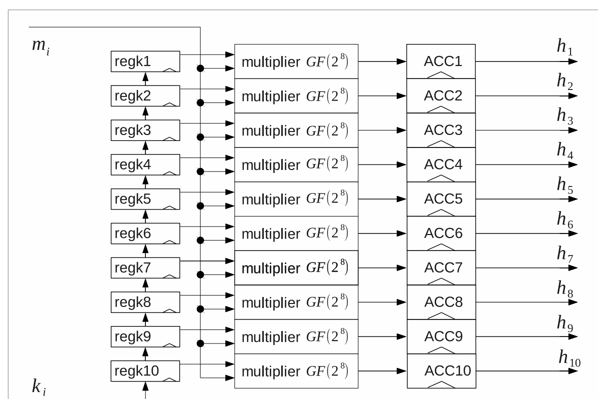


Figure 8.6: Architecture of MLUH

The PD Construction

We mentioned in Section 8.1.2 that PD (pseudo dot construction) is a variant of MLUH with some interesting differences. For hashing a m block message to a b block output, where each block in d bit long, the MLUH requires mb multiplications in $GF(2^d)$, but PD requires only $\frac{mb}{2}$ multiplications (assuming m is even). Though the total number of multiplications in PD is lesser than in MLUH, obtaining an advantage with PD in the hardware realization of STES is a bit difficult. Note that each multiplication in case of PD is of the form $(M_i \oplus K_j)(M_{i+1} \oplus K_{j+1})$, hence performing one multiplication PD requires two blocks of message and key materials which is not the case in MLUH. This issue is important in a proper design of PD as our design would be used in conjunction with a stream cipher in STES, and at times the message and key blocks used by the PD would be obtained as an output from the stream cipher. To keep this balance, we decided to construct a PD with $d/2$ bit multipliers when the message and key blocks are considered to be d bit blocks. Here we showcase an architecture for PD with 4 bit multipliers which produces a 80 bit output. Later, we use this PD construction with a stream cipher with a 8 bit data path to construct STES.

We implement the PD as is shown in Figure 8.7. The methodology adopted is the same as in case of the architecture of MLUH (shown in Figure 8.6), in the sense that here also we compute column wise. But as we use 4 bit multipliers, to get a 80 bit output we require 20 multipliers. We assume that the key blocks (K_j) and message blocks (M_i) are obtained as 8 bit blocks, but we treat each multiplication as $(M_i^H \oplus K_j^H)(M_i^L \oplus K_j^L)$, where $M_i = M_i^H || M_i^L$ and $K_j = K_j^H || K_j^L$ and $|M_i^H| = |M_i^L| = |K_j^H| = |K_j^L| = 4$. Here we have twenty registers named $\mathbf{regK1}, \mathbf{regK2}, \ldots, \mathbf{regK20}$, each of length 8 bits connected in a cascade forming a FIFO as in case of MLUH architecture. These registers contains the key blocks, and the message blocks are obtained from the input lines m_i^H and m_i^L.

This 4 bit design of the PD is not more efficient than the MLUH architecture. Though the 4 bit multipliers used in PD are smaller than the 8 bit multipliers but we require the double of them and also the double amount of registers and the extra xors required at the input of the multipliers makes this PD architecture more costly than the MLUH architecture. Moreover the number of clock cycles required in this case is also same as in case of MLUH.

The multipliers used in both PD and MLUH are Karatsuba multipliers, they were implemented following the same design strategy as presented in Section 6.6.1. The irreducible polynomials used to implement the multipliers are listed in the Table 8.2, also these multipliers are smaller than the one presented in Section 6.6.1 as they operates on smaller numbers. To keep the speed high and seeing that there are no dependencies between multiplications in MLUH and PD, after a careful re-timing process the multipliers for $d > 4$ were divided into almost balanced pipeline stages.

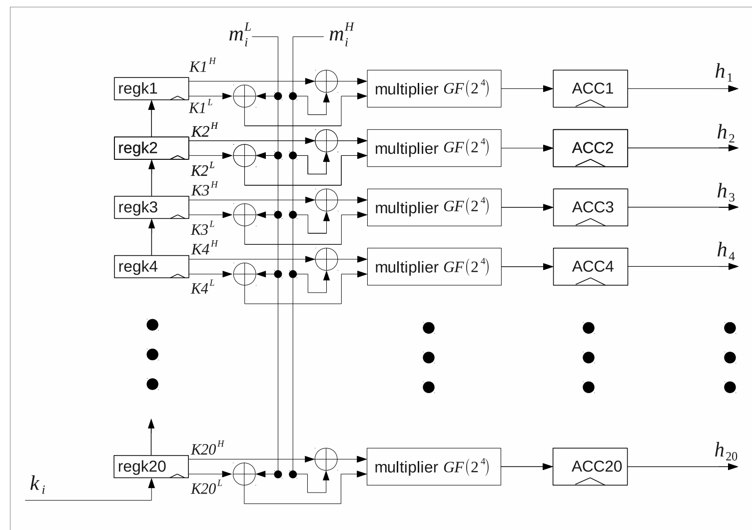


Figure 8.7: Architecture of PD.

8.4.3 Implementation of stream ciphers

In this work we consider three stream ciphers: Trivium, Grain128 and Mickey128-2.0. These stream ciphers are the eStream hardware based stream ciphers and are in general very easy to implement in hardware as they are constructed using lite simple structures shift registers and some simple boolean functions. All these three stream ciphers can be implemented using a shift register as a basic primitive.

We implement the stream cipher using various data paths, here by a data path we mean the number of bits of output the stream cipher can produce in each clock cycle. A lower data path uses less parallelism and thus can be implemented with fewer hardware resources. The various data paths that we consider for the three stream ciphers along with some other important parameters are depicted in Table 8.1.

For all the three stream ciphers, the bit-wise versions (i.e. the ones with data path of one bit) can be implemented in a very compact way in Spartan-3 devices. Spartan-3 FPGAs can configure the Look-Up Table (LUT) in a SLICEM slice as a 16-bit shift register without using the flip-flops available in each slice. Shift-in operations are synchronous with the clock, and output length is dynamically selectable. A separate dedicated output allows the cascading of any number of 16-bit shift registers to create whatever size shift register is needed. Each configurable logic block can be configured using four of the eight LUTs as a 64-bit shift register. Such an usage of the LUT in Spartan-3 is called a SRL16 primitive [145]. This SRL16 primitive can be used to implement the shift registers of the stream ciphers [16]. SRL16 supports only a single entry and a single bit shift, so if the data path is more than 1 then this primitive cannot be used and then the shift registers must be implemented using simple LUTs. We implemented bit-wise versions of Trivium and Grain128 using SRL16

primitive. A bit-wise version of Mickey128 2.0 was also implemented, but the structure of Mickey128 2.0 does not allow efficient use of SRL16. Also, we did not implement Mickey128-2.0 with data paths more than 1, as such parallelization in Mickey is not straight forward to obtain.

Here, as an example we will explain in details the a specific architecture of Trivium with a 2-bit datapath. The internal state of Trivium is a 288-bit shift register, for implementation purposes it is divided into a three registers $SR1$, $SR2$ and $SR3$ as shown in Figure 8.8. All the three shift registers have two inputs and two outputs and in each clock cycle their internal states are shifted by two positions. Initially $SR1$ and $SR2$ are initialized with the 80 bit key K and the 80 bit IV respectively. $SR3$ has as initial value the string $1^3||0^{108}$. In the Figure 8.8 it can be seen that for each shift register its feedback functions depends on some bits from it and a function computed with some bits from the previous register. For example, the feedback of shift register $SR3$ depends on some bits of $SR2$ and two bits from itself. It is easy to see in Figure 8.8 that the feedback functions for all registers and the function to compute the final outputs S_{even} and S_{odd} are replicated two times, just they have different inputs. In the case of Gran128 the way to increment the datapath also consist of replicating the feedback functions of shift registers and the output function. Increasing the datapath of Grain128 and Trivum brings a significant increase in throughput since it reduces the time used for setup and give a parallel output for the stream.

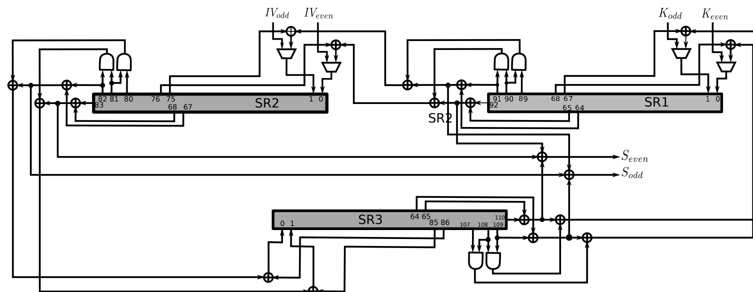


Figure 8.8: Architecture for Trivium.

8.4.4 Implementation of STES

We implemented STES with all the three stream ciphers with the data-paths specified in Table 8.1. When we consider a stream cipher with data path d in implementing STES then we use the hash function with the same value of the data path, i.e., in the hash function we use multipliers in $GF(2^d)$ if the hash function is MLUH, and if it is PD then the multipliers are in $GF(2^{\frac{d}{2}})$.

We will explain in details a 8-bit data path implementation using Trivium and MLUH, but for other instantiations of stream ciphers and hash the basic design remains the same. Note

that Trivium uses a a 80 bit IV and a 80 bit key.

In the Figure 8.9 we show the generic architecture for encrypting with STES, we shall explain the architecture with reference to the algorithm of STES (Figure 8.2) and the Feistel network (Figure 8.3).

The circuit presented in Figure 8.9 consists of the following basic elements:

1. The MLUH constructed with 8 bit multipliers as discussed in Section 8.1.2. In the diagram this component is labeled **MLUH**.
2. Two stream cipher cores labeled **SC1** and **SC2**.
3. Two 80 bit registers **RegH1** and **RegH2** which are used to store the output of **MLUH**.
4. Nine registers labeled **regA11**, **regA12**, **regA22**, **regKH**, **regF1**, **regF2**, **regB1**, **regB21**, **regB2**. All these registers are 80 bit long and are formed by ten registers each of eight bits connected in cascade, so that they can be used as a FIFO queue. The same structure was used in the design of MLUH and PD. When implemented in Spartan 3, these registers are implemented using the SL16 primitive.
5. Five multiplexers labeled **1**, **2**, **3**, **4**, **5**.
6. The control unit whose details are not shown in the Figure.
7. All data connections except the connections which connects **MLUH** with **RegH1** and **RegH2** have a data path of 8 bits. The connections between **MLUH** and the registers **RegH1**, **RegH2** have a data path of 80 bits.
8. The input lines M_i, IV and K which receives the data and tweak, the initialization vector and the key respectively.
9. The output line C_i which outputs the cipher.

The **MLUH** computes the MLUH, it receives as inputs message blocks M_i, tweak blocks T_i and key blocks K_i and give as output the result of MLUH in its output port S. The register **RegH1** and **RegH2** receive the output from S as input, in this case $|S| = 80$ bits. The registers **RegH1** and **RegH2** are designed to give eight bit blocks as outputs in each clock cycle in their output port BO. The **MLUH** receives its input from the 4x1 multiplexer labeled **1**. Notice, that in the algorithm of STES, the MLUH is called on four different inputs. Multiplexer **1** helps in selecting these inputs. In the algorithm MLUH is called on two different keys τ' and τ'', thus, **MLUH** can receive the key from two different sources: the key τ' is received directly from the output of the stream cipher **SC1**. The key τ'' is received from the register **regKh** which is used to store τ''. To accommodate these selection of keys the input port Ki of **MLUH** receives the input from the 2×1 multiplexer **5**.

We use two stream ciphers **SC1** and **SC2**. Both take the key from the input line K of the circuit. **SC1** receives the IV from multiplexer **2**, it selects between input line IV or F_1. Multiplexer **3** feeds the IV to the stream cipher **SC2**, it selects between F_2 or M.

Next we explain the data-flow of the architecture of the Figure 8.9 with reference to the algorithm in Figure 8.2.

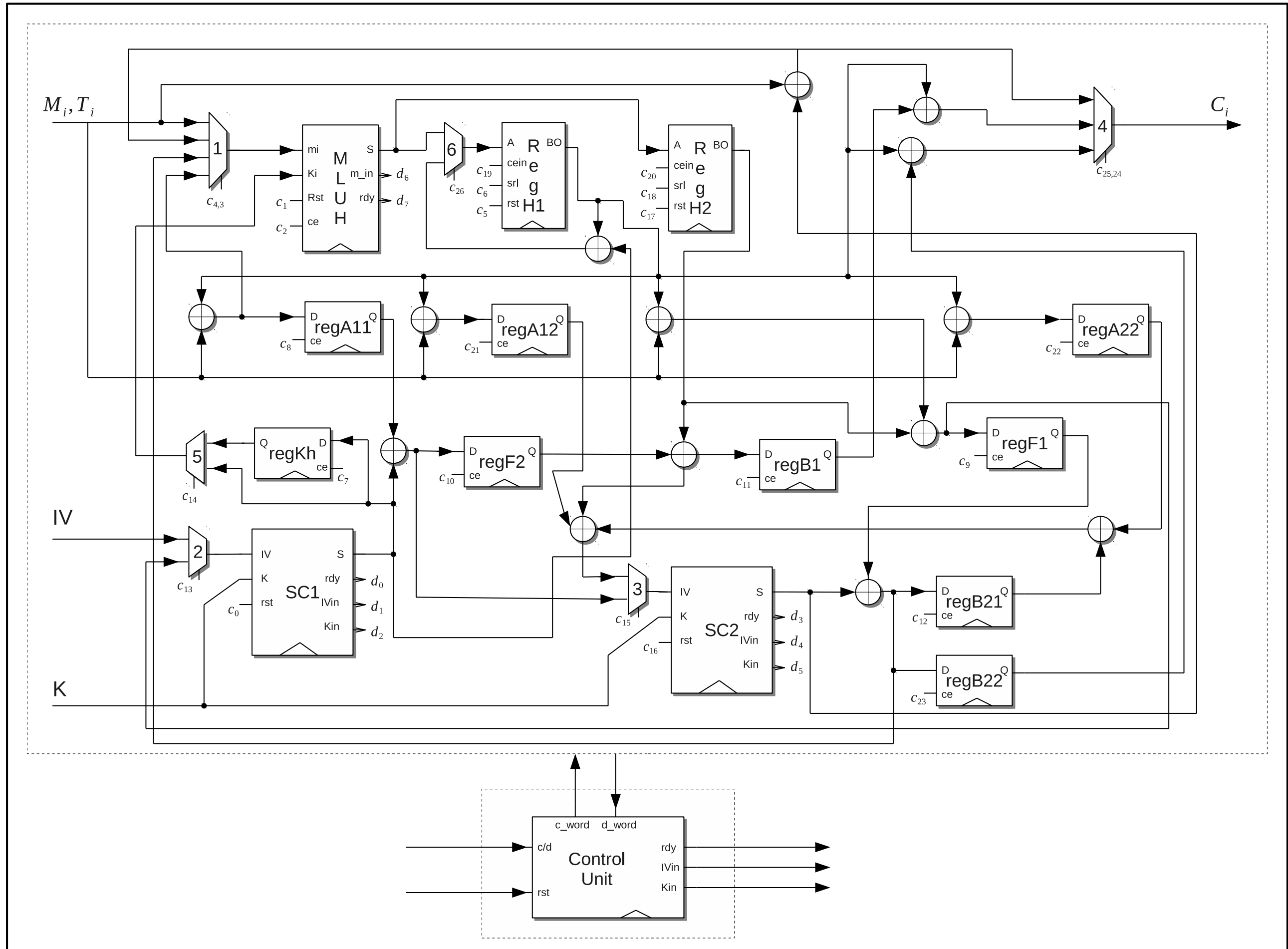


Figure 8.9: General Architecture of TES.

8.4.5 Data Flow and Timing Analysis

In this section we shall discuss the data flow in the circuit presented in Figure 8.9 and also discuss the parallelism that we achieve using the circuit and the latency of the various operations, we present a time diagram depicting the time taken by each operation in terms of clock cycles in Figure 8.10. In Figure 8.10 the basic operations are depicted with rectangles and the numbers inside the rectangles denotes the time required for the operation in clock

cycles.

We consider encryption of a message P of length 4096 bits. As per the algorithm P is parsed into three blocks P_1, P_2, P_3, where $|P_1| = |P_2| = \ell = 80$ and $|P_3||T| = 4016$. The encryption procedure starts with the computation of the key bits $\tau = \tau'||\beta||\tau''$ using the stream cipher **SC1**. It is required to generate 530 bytes of key material using the stream cipher. Our implementation for Trivium takes 154 cycles for key setup, and to generate 540 bytes it again takes 540 cycles. The key bytes generated by **SC1** is parsed as τ', β and τ''. The key bytes τ' are not stored and are immediately fed to the **MLUH** as it gets generated. Note, that these key bytes are used again and then they are again generated using SC1. Storage of this huge key material would amount to more area of the circuit in terms of extra registers hence we decided against the option of storing it. β also is not stored because it can be used while it is generated by $SC1$ and xored with the value in RegH1. τ'' is used twice inside the Feistel network, and the size of τ'' is much smaller than τ', hence it is stored in **regKh**.

As soon as the key set up phase of **SC1** is over, in each clock it generates one byte of key material and these key materials gets stored in the FIFO register of MLUH one byte per cycle. After 10 cycles the FIFO is full and thus the MLUH can start computing Z_1. It takes 524 cycles to complete the computation of Z_1 and it runs in parallel with the stream cipher, note that τ' and β are required to compute Z_1. The value of Z_1 is stored in the register **RegH1**.

Next, using Z_1 and the message blocks P_1 and P_2 obtained from the input line the values A_1 and A_2 are computed. A_1 is stored in both the registers **regA11** and **regA12**, and A_2 is stored in **regA2**.

Once A_1 have been computed, the stream cipher **SC1** has already started generating the τ'' part of the key. Thus τ'' and A_1 are fed to **MLUH** to compute H_1 (line 2 of Fig. 8.3), H_1 is stored in the register **RegH2**. Using the value of H_1 in **RegH2** and the value of A_2 in **regA2** F_1 is computed and stored in **regF1**. The value of F_1 is then fed to **SC1** as an initialization vector to compute G_1. After the initialization of **SC1** is completed and it starts producing G_1, then computation of F_2 is started and is fed to **SC2** in parallel to compute G_2. When the initialization process of **SC2** ends, G_2 starts getting computed and it is used with the value stored in **regF1** to compute B_2 which is stored in **regB21** and **regB22**. Also B_2 is fed to the **MLUH** to compute H_2 using the key stored in **regKh**. When H_2 is ready it is stored in **RegH2** and is used with the value stored in **regF2** F_2 to compute B_1, B_1 is stored in **regB1**. This completes the evaluation of the Feistel in line 17 of the algorithm in Fig. 8.2. The total computation of this phase takes 372 cycles as depicted in Fig. 8.10.

In parallel with the computation of B_1, M is computed using the values stored in **regA12**, **regA2** and **regB2** which stores the variables A_1, A_2, B_2 respectively. M is fed as IV to the **SC2** in parallel to its computation, when the initialization process of **SC2** ends the stream cipher is ready to produce C_3y. When the first eight bits of cipher text C_3 are ready, then it is fed to the **MLUH** to compute Z_2, for doing this we run the **SC1** in parallel to **SC2**

to generate the key τ'. When Z_2 is ready it is stored in **RegH2**, and it is used with the values stored in **regB1** and **regB22** B_1, B_2 to compute C_1 and C_2. As shown in Fig. 8.10, for producing C_3 a total of 492 cycles are required after initialization of the stream cipher. Z_2 is computed in parallel to C_3. After Z_2 is complete, computation of C_1 and C_2 takes 20 more cycles.

The total time taken to encrypt 512 bytes is 1748 cycles and the delay called latency before the first bit of the cipher text is produced is 1204 cycles.

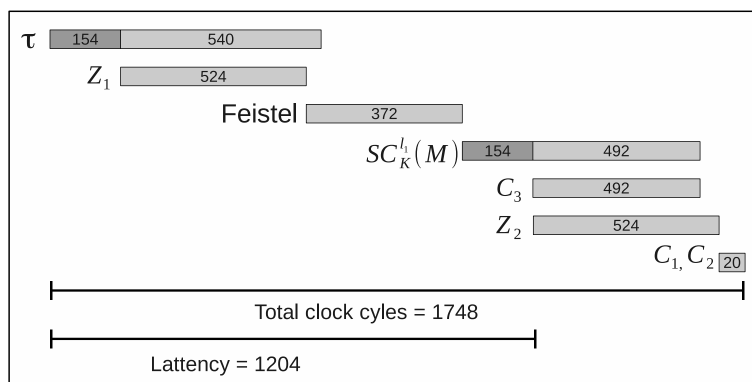


Figure 8.10: Implementation of complete TES

8.5 Experimental Results

We implemented STES on two different families of FPGAs: Lattice ICE40 and Xilinx Spartan 3. For Spartan 3 we we used the device *xc3s400-fg456* and in case of Lattice ICE40 we selected *LP8KCM225*. The place and route results in case of Spartan 3 were generated using *Xilinx-ISE* version 10.1. For ICE40 we used Silicon Blue Tech iCEcube release 2011.12.19577. We measured the power consumption of the circuits using *Xilinx Xpower Analyzer* for Spartan 3 and Power Estimator of iCEcube for ICE40.

For our implementations we report performance in terms of throughput, area and power-consumption. In case of Spartan 3 we report area in terms of number of slices and for ICE40 we report in terms of number of logic cells. It is to be noted that size of a Spartan 3 slice is almost equal to twice the size of a ICE40 logic cell.

In this section we present the experimental results in two parts. First in Section 8.5.1 we report performance data of the primitives, i.e., the stream ciphers and the hash function and in Section 8.5.2 we report results of STES using various instantiations of the primitives.

8.5.1 Primitives

In the Tables 8.3 and 8.4 we show the performance results of MLUH and PD implementations. For all the cases, we consider hashing a message of 4016 bits to 80 bits. In the Tables MLUH-db represents an MLUH with a d bit data path. And PD-4b8b represents a PD construction where the multipliers are in $GF(2^4)$ but they receive inputs and outputs as 8 bit streams (see the architecture discussed in Section 8.4.2).

It is clear from the Tables that in both Spartan 3 and ICE40 with the increase in data path the throughput increases at the cost of area. For MLUH-1b, we obtain a very high frequency, as in this case the multiplier multiplies two one bit numbers which can be implemented only using a 2×1 multiplexer. As the data path increases, the complexity of the circuit implementing the multiplication grows which increases the critical path of the circuit. For 16, 32 and 40 bit implementations we break the critical path of the multiplier by dividing it into balanced pipeline stages, the number of pipeline stages were carefully selected to maintain a high operating frequency. This is the reason why all 8, 16, 32 and 40 bit implementations operate on similar frequencies on both Spartan 3 and ICE40.

PD4b8b is an implementation of PD using 4-bit multipliers but the key and message entries have eight bytes and internally they are divided into 4-bit, so this implementation emulates an 8-bit data-path implementation. PD4b8b use twenty 4-bit multipliers which is double the number of multipliers used by MLUH-8b. But the multipliers in PD4b8b have the half of data-path of those in MLUH8b, hence the area occupied by PD4b8b is almost same as that of MLUH-8b and gives almost the same throughput. Hence using PD over MLUH does not seem to have much advantage in hardware, hence we do not present the PD construction with different data paths as we do for MLUH.

Primitive	Slices	Frequency (MHz)	Pipeline Stages	Throughput (Mbps)
MLUH-1b	158	215.11	0	210.90
MLUH-4b	247	183.76	0	719.26
MLUH-8b	452	177.46	1	1386.51
MLUH-16b	737	175.24	2	2717.25
MLUH-32b	1259	176.97	2	5425.28
MLUH-40b	1410	173.89	3	6588.13
PD4b8b	415	179.27	0	1400.69

Table 8.3: MLUH on Spartan 3.

From Table 8.4 we can see that the number of logic cells required for ICE40 FPGA is almost double than the slices required in Spartan 3. We explained in Chapter 3 that a logic cell in

ICE40 has much lesser components than in a Spartan 3 slice, which explains the difference in area in the two families. Moreover, the ICE40 implementations operate at a little lower frequencies compared to the Spartan 3 implementations, this can also be explained by the fact that as a ICE40 has lesser components so the critical path of the implementations in ICE40 are more complex in terms of logic resources.

Primitive	Logic blocks	Frequency (MHz)	Pipeline Stages	Throughput (Mbps)
MLUH 1b	325	189.78	0	189.78
MLUH 4b	461	180.85	0	707.89
MLUH 8b	810	171.24	1	1337.84
MLUH 16b	1638	170.52	2	2644.05
MLUH 32b	2531	173.90	2	5331.16
MLUH 40b	2756	174.80	3	6622.31

Table 8.4: MLUH on Lattice ICE40.

The results of the stream ciphers are shown in Tables 8.5 and 8.6. In these Tables we present the performance data of Trivium, Grain and Mickey with various data paths. In the Tables the names of the stream ciphers are suffixed with the data path.

The bit-wise implementation of Trivium and Grain128 on Spartan 3 were done using SRL16 primitives and this allowed us to obtain very compact designs: 49 Slices for Trivium-1b and 67 Slices for Grain128-1b. Grain128-1b is larger than Trivium-1b due to the complexity of its output and feedback functions. Mickey128 was implemented only with a one bit data path because there is no direct way to parallelize it.

Tables 8.5 and 8.6 shows that the increase in data path does not have much effect on the total area of Grain128 and Trivium. For example, Trivium-8b requires 148 slices and Trivium-16b requires 203 slices. Though one would expect that doubling the data path would require double the hardware resources, that is not the case. The growth in area is small because in Trivium the state is stored in a 288-bit shift register independent of the size of data-path. For wider data paths we only require to replicate the output and the feedback functions a suitable number of times.

Wider data path implementations of Grain128 also have the same behavior as implementations of Trivium. As Grain128 has a 96 bit IV hence for our requirement that the data path must divide the IV length we do not implement grain with a 40 bit data path which we do for Trivium.

Primitive	Slices	Frequency (MHz)	Setup (cycles)	Throughput (Mbps)
Trivium-1b	49	201.02	1232	201.02
Trivium-4b	120	197.38	308	789.52
Trivium-8b	148	193.49	154	1547.93
Trivium-16b	203	189.32	77	3029.16
Trivium-40b	278	187.83	31	6010.60
Trivium-64b	435	186.95	20	11964.26
Grain128-1b	67	193.00	384	193.00
Grain128-4b	175	182.54	96	730.17
Grain128-8b	232	178.80	48	1430.42
Grain128-16b	320	179.73	24	2875.64
Grain128-32b	490	173.58	6	5554.56
Mickey128-1b	182	202.80	286	202.80

Table 8.5: Stream ciphers on Spartan3.

Primitive	Logic cells	Frequency (MHz)	Setup (cycles)	Throughput (Mbps)
Trivium-1b	313	190.26	1232	190.26
Trivium-4b	329	188.54	308	754.15
Trivium-8b	347	186.13	154	1489.04
Trivium-16b	398	176.10	77	217.60
Trivium-40b	530	166.73	31	6669.20
Grain128-1b	297	192.59	384	192.58
Grain128-4b	360	162.41	96	649.64
Grain128-8b	434	145.73	48	1165.84
Grain128-16b	592	137.72	24	2203.53
Grain128-32b	997	136.84	12	4378.88
Mickey128-1b	420	169.74	288	169.74

Table 8.6: Stream ciphers on Lattice ICE40.

8.5.2 Experimental results on STES

Using the primitives described in Section8.5.1 we construct STES. The performance results are shown in Tables 8.7 and 8.8. The Tables shows data for STES implemented with various stream cipher instantiations and data paths. The Tables also show the power consumption characteristics for the implementations.

In Figures 8.11 and 8.12 we present the data in Tables 8.7 and 8.8 for STES instantiated with

Trivium and Grain128 in a pictorial form. Figure 8.11 shows the growth of area, throughput and total power for STES using Trivium and Figure 8.12 shows the same for STES using Grain. Note that the plots are in logarithmic scale. We can observe that with increase in data path the growth of throughput is much faster than the growth of area, this reflects the characteristic of Trivium as shown in Table 8.5. The growth of power consumption is the slowest.

The implementations which use Grain128 are faster than the ones using Trivium, because the implementations with Grain needs less clock cycles in comparison with implementations with Trivium.

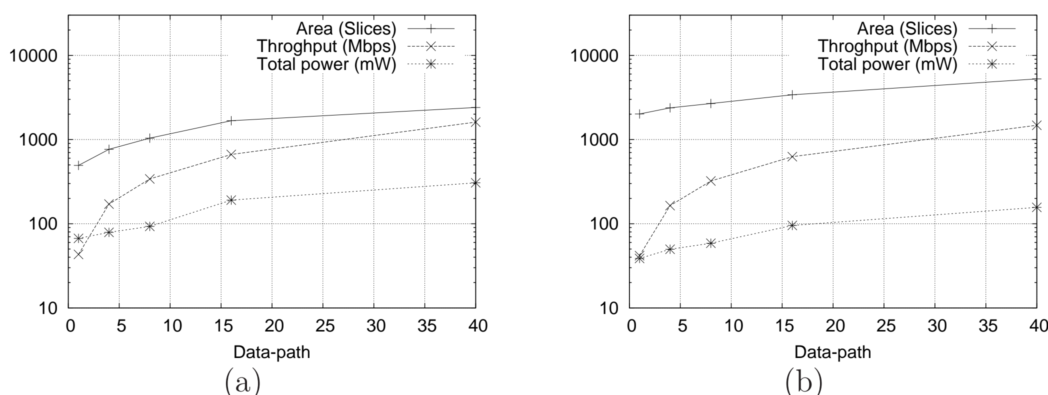


Figure 8.11: Growth of Area, Throughput and Total power for STES-T: (a) Spartan 3 (b) Lattice ICE40

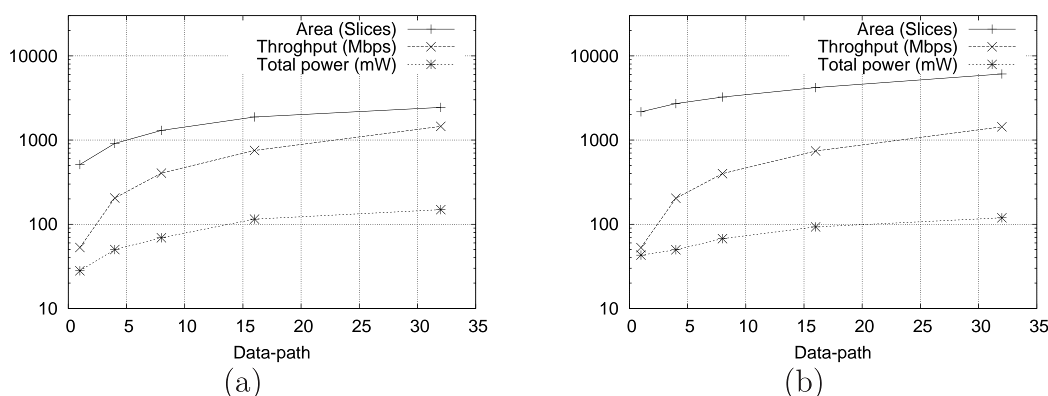


Figure 8.12: Growth of Area, Throughput and Total power for STES-G: (a) Spartan 3 (b) Lattice ICE40

STES-T-PD4b8b is comparable with STES-T-8b in terms of area and throughout hence we can conclude that for this application the use of Pseudo Dot Product is not very useful, as

Mode	Slices	Cycles	Frequency (MHz)	Throughput (Mbps)	TPA	Static power (mW)	Dynamic power (mW)	Tolal power (mW)
STES-T-1b	493	13765	145.00	43.18	21.37	38	29	67
STES-T-4b	766	3449	144.05	171.07	54.52	38	41	79
STES-T-8b	1038	1749	143.71	336.56	80.07	38	55	93
STES-T-16b	1672	871	141.51	665.46	97.17	38	153	191
STES-T-40b	2400	355	139.65	1611.31	133.15	59	247	306
STES-G-1b	512	10501	135.58	52.88	25.22	37	28	65
STES-G-4b	912	2633	131.66	204.81	54.83	37	50	87
STES-G-8b	1302	1321	130.39	404.29	75.81	38	69	107
STES-G-16b	1879	667	122.32	751.16	97.60	57	115	172
STES-G-32b	2439	339	120.18	1452.06	145.35	58	141	199
STES-M-1b	755	10117	132.76	53.75	17.38	38	34	72
STES-T-PD4b8b	1012	1729	143.90	342.79	82.70	38	56	94

Table 8.7: STES on Spartan 3. STES-T: STES using Trivium, STES-G: STES using Grain128, STES-M: STES using Mickey128, all with MLUH with the specified data path. STES-T-PD4b8b: Small TES using Trivium and PD4b8b as a hash function.

a significant improvement of area or throughput is not seen in this case. For this reason we do not implement this version in ICE40.

In the Table 8.8 we can see the experimental results for implementations of STES on ICE40. The comparative behavior reflected in the Table 8.8 is almost the same as the behavior of implementation on Spartan 3 shown in Table 8.7. In general the implementation on ICE40 are slower than the implementations on Spartan 3, but the power consumption on ICE20 is much better. In particular we observed that the static power consumption in ICE40 remains constant for all variants. This is probably due to the fact that ICE40 was specifically designed to be used in low power applications, hence its architecture has special characteristics which allows it to run with a very low power consumption.

As the implementations on ICE40 performs the best in terms of power consumption, hence we measured the performance of all our designs when the operating frequency was fixed to 100MHz. The ICECube software allows such simulations. In the Table 8.9 we show the power consumption of all implementations in ICE40 when operating at a fixed frequency of 100 Mhz. In Figure 8.13 we show the data in Table 8.9 graphically. As expected, if we lower the operating frequency the circuits consume considerably lesser amount of power.

Mode	Logic cells	Cycles	Frequ-ency (MHz)	Throu-ghput (Mbps)	TPA	Static power (mW)	Dynamic power (mW)	Tolal power (mW)
STES-T-1b	2013	13765	140.29	41.75	5.06	0.16	38.49	38.65
STES-T-4b	2379	3449	138.15	164.07	16.84	0.16	49.60	49.76
STES-T-8b	2676	1729	165.78	321.66	29.35	0.16	58.45	58.61
STES-T-16b	3402	871	133.07	625.78	44.91	0.16	95.17	95.33
STES-T-40b	5252	355	128.08	1477.79	68.65	0.16	156.27	156.42
STES-G-1b	2165	10501	135.26	52.76	5.95	0.16	42.55	42.91
STES-G-4b	2708	2633	130.87	203.59	18.35	0.16	49.63	49.78
STES-G-8b	3242	1321	128.59	398.71	30.03	0.16	67.26	67.42
STES-G-16b	4204	667	120.76	741.58	43.07	0.16	92.77	92.93
STES-G-32b	6092	339	118.66	1434.81	57.50	0.16	119.19	119.35
STES-M-1b	1720	10117	130.75	52.94	7.51	0.16	42.49	42.65

Table 8.8: STES in Lattice ICE40. STES-T: STES using Trivium, STES-G: STES using Grain128, STES-M: STES using Mickey128, all with MLUH with the specified data path.

Mode	Trough-put (Mbps)	Static power (mW)	Dynamic power (mW)	Tolal power (mW)
STES-T-1b	29.76	0.16	27.44	27.60
STES-T-4b	118.76	0.16	33.71	33.87
STES-T-8b	236.90	0.16	43.07	43.23
STES-T-16b	470.27	0.16	71.52	71.68
STES-T-40b	1153.80	0.16	122.08	122.08
STES-G-1b	39.01	0.16	32.35	32.51
STES-G-4b	155.57	0.16	37.92	38.08
STES-G-8b	310.07	0.16	52.31	52.47
STES-G-16b	614.09	0.16	76.81	76.97
STES-G-32b	1208.26	0.16	100.35	100.51
STES-M-1b	40.49	0.16	35.69	35.85

Table 8.9: STES in Lattice ICE40 at a frequency of 100 Mhz.

8.5.3 Comparison with Block Cipher Based Constructions

As mentioned earlier STES is highly motivated by the construction presented in [128], here we present some results and estimations on the construction in [128].

The construction in [128] does not use stream cipher, it uses a block cipher in counter mode

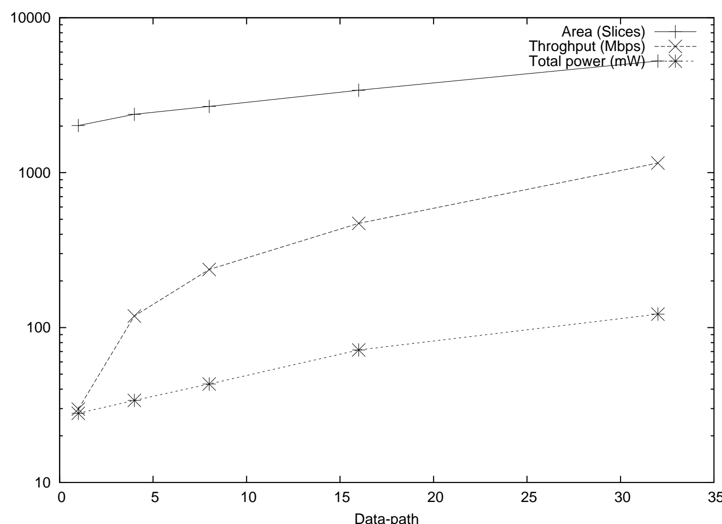


Figure 8.13: Growth of Area, Throughput and Total power for STES-T operating at 100 Mhz in ICE40

to do the bulk encryption, and the suggested hash functions are either normal polynomial hashes or BRW polynomials. The performance of the construction in [128] when implemented using a AES with 128 bit key and a normal polynomial hash is shown in Table 8.10. The Table reports four implementations, which are described below:

TES-sAESs-1s: TES in [128] implemented with a sequential AES128 and a fully parallel 128 bit Karatsuba Multiplier (the same AES and multiplier as used in the last row of Table 5.8) in Spartan 3.

TES-AESs-4s: Sequential AES with one 4 stage pipelined 128 bit multiplier implemented in Spartan 3.

TES-AESp-4s: 10 stage pipelined AES with 4 stage pipelined 128 bit multiplier implemented in Virtex 5.

TES-sAES-1s: This is an estimation based on a very compact AES reported in [121], and a polynomial hash which uses four 32 bit multipliers as used in case of MLUH-32b. The estimation is based on the data in [121] that the AES occupies 167 slices and takes 42 cycles to produce a single block of cipher. The estimated slices is obtained by summing the slices of the components and the frequency is estimated by considering that the critical path of the circuit would be given by the component with the highest critical path. Real implementations may change these data.

The results in Table 8.10 shows that the implementations of the TES described in [128] with a sequential AES in Spartan 3 takes up much more area than our designs with stream cipher

Mode	Slices	Cycles	Frequency (MHz)	Throughput (Mbps)	B-RAMS	Static power (mW)	Dynamic power (mW)	Tolal power (mW)
TES-AESs-1s Spartan 3	6170	388	45.58	480.70	11	191	182	372
TES-AESs-4s Spartan 3	6389	403	74.07	752.10	11	192	245	437
TES-AESp-4s Virtex 5	4902	111	287.44	10596.44	0	1243	2276	3519
TES-sAESs-1s Spartan 3 *	2800	2694	71.51	108.62	3	-	-	-

Table 8.10: . TES implemented using AES-128 and a polynomial hash on different platforms. TES-AESs-1s: Sequential AES with one fully parallel 128 bit multiplier, TES-AESs-4s: Sequential AES with one 4 stage pipelined 128 bit multiplier, TES-AESp-4s: Ten staged pipelined AES encryption core with 4 stage pipelined 128 bit multiplier. TES-sAESs-1s (estimation): A small AES (167 slices, 3 block RAMs, with latency of 42 cycles per block) and four 32-bit multipliers.

and our designs with data path of more than 8 bits achieves higher throughput at the cost of smaller area and lower power consumption.

TES-AESp-4s is a huge design and it does not fit in a Spartan 3 device (note the slices in Virtex 5 have much more resources than the slices in Spartan 3 and the of slices in these two families are not quite comparable). TES-AESp-4s achieves throughput similar to the designs of HMCH[Poly] and HEH[Poly] reported in Chapter 7, but it cannot be in any sense considered as a light weight design. But the performance TES-AESp-4s do show that the TES in [128] can achieve quite high throughput.

The design philosophy adopted in TES-sAESs-1s is probably best comparable to our stream cipher based designs. As in TES-sAESs-1s we intend to use a very compact AES. The estimation shows that such an implementation would also occupy quite a large area but not achieve a good throughput.

Use of light weight Block Ciphers: In the current days there have been numerous proposals for light weight block ciphers like PRESENT [13], KATAN, KTANTAN [20], KLEIN [58], LED [62] etc. These block ciphers are designed to optimize the hardware resources required to implement them. In a generic description of a TES any secure block cipher can be used, thus there is no technical difficulty in plugging in a light weight block cipher in an existing description of a TES and this would lead to a low cost design compared to the AES alternatives that we just discussed. But a thing to note is the light weight block ciphers which we mentioned are mainly designed to be used in specific applications like in RFID

	Read Speed (Mb/s)	Write Speed (Mb/s)
Class 4	323.77	32.77
Class 10	163.84	131.07
Class 10 extreme	778.24	368.64
USB 3.0	819.2	409.6
USB 2.0	245.76	163.84

Table 8.11: Highest speed rates of SD cards and USB memories of different types which are currently sold by Kingston

authentication etc., and are not designed for bulk encryption in a mode. The light weight block ciphers have small block lengths, for example all the schemes mentioned above have a block length of 64 bits of lower. Such small block lengths would restrict their use in TES as the block length of the block cipher used in a block cipher based TES is an important security parameter. Recall that all existing block cipher based TES enjoys a security lower bound of $c\sigma^2/2^n$, where σ is the query complexity of the adversary, c is a small constant and n the block length of the underlying block cipher. Thus, the security guarantees provided by the known reductions are not sufficient if n has a value less or equal to 64. Thus we feel that given our current state of knowledge it would not be advisable to use block ciphers of small block lengths for constructing TES. It may be so that the current reductions are not tight, or there exist possibility of new constructions with better than quadratic security bounds, existing light weight block ciphers can be useful in such scenarios.

8.5.4 Discussions

The main design goal of STES was to obtain a TES which can be implemented in a compact form and would have low power consumption. Our experiments validate that STES do achieve these goals to a large extent.

In the Introduction we presented a Table of the speeds recommended by the SD standard. The commercially available memories does not achieve the values specified in the standard. In Table 8.11 we present the maximum speed of the various classes of memories sold by Kingston. The first three rows corresponds to SD cards and the last two rows gives data of USB memories. Our implementations with 16 bit or higher data paths can surpass the read speeds of all these types of memories, and such speed would be achieved with a very low hardware footprint and power consumption.

Chapter 9

Side Channel Attacks on Some TES

Marcos is gay in San Francisco, black in South Africa, an Asian in Europe, a Chicano in San Isidro, an anarchist in Spain, a Palestinian in Israel, a Mayan Indian in the streets of San Cristobal, ..., a Zapatista in the mountains.

Subcomandandante Insurgente Marcos

Till now we have focused on constructions of secure and efficients TES and their hardware implementations. In this Chapter we would for the first time deviate a bit from the previous theme. Here we would focus on a class of attacks called *side channel attacks* which are applicable to any cryptographic implementation.

The art of science of analyzing crypto-systems to find various vulnerabilities in it is collectively called cryptanalysis. Classical cryptanalysis considers that a cryptographic algorithm runs in an abstract device like the Turing machine. Thus, in classical cryptanalysis the only focus is on the algorithm itself and it is independent of the way in which an algorithm is implemented. But in real life any algorithm has to be implemented in a physical device to make it useful. Real physical devices are different from abstract machines in the sense that they always emit (or leak) certain types of information related to the computation being performed. It has been noted that a computing device may leak electromagnetic radiation, timing information, sound etc. as a product of computation, and these leaked signals may be used to predict the type of computation going on in a machine. Utilizing these leaked information for the purpose of cryptanalysis is now called side channel analysis. Side channel analysis has gained a lot of importance in the current days and it has been demonstrated that careless implementation of algorithms which are otherwise considered secure in the light of classical cryptanalysis can be trivially broken if certain side channel information is available to an adversary.

Most of the proposed TES have a security proof attached to it, these proofs gives us confi-

dence regarding the security of the algorithm, but the guarantee against side channel vulnerabilities are not provided by such security proofs. To our knowledge, a systematic study of side channel vulnerabilities in TES have not be done yet and it is a promising research area which may be practically very useful. In this Chapter we aim to address this issue a little bit by trying to analyze side channel weaknesses of some TES. The modes that we choose here are EME and EME-2. We show that these modes do not provide the claimed security guarantees if we consider adversaries with access to some side channel information. The possible side channel weakness of EME and two of its derivatives EME^+ and EME^* were pointed out in an appendix of [110], but the true vulnerabilities were not analyzed in details. Thus, the work presented in this Chapter can be considered as an extension of that reported in [110], but our analysis is substantially different from that in [110].

The study presented here can be seen as the first step towards analyzing side channel vulnerabilities in TES, as our study is only theoretical in nature and is not backed by real experiments and measurements. Such experiments would be necessary to judge the extent of the weakness and to design proper counter measures, but in the study presented in this Chapter we do not cover this aspects. The material in this Chapter have been previously published in [96]

9.1 Adversaries with Access to Side Channel Information

Modern security guarantees provided by security proofs, are based on complexity theoretic arguments and assumptions on abstract notions of computations. For security definitions, one models the adversary as a probabilistic polynomial time algorithm with access to inputs/outputs of a protocol. The adversary knows the algorithm which produces the outputs for his chosen inputs, but has no knowledge of a secret quantity which is called the key. Moreover, as the computation model is assumed to be an abstract one, hence the state of the algorithm is also assumed to be invisible to an adversary, who cannot know the branches taken, subroutines called, etc. by the algorithm during its execution.

However, this is not a realistic scenario, since computations must be done on a physical device, and that device can leak various kinds of information. This can motivate attacks from the adversary who by applying measurements on the leaked information may be able to gain access to sensitive data related to the computation being performed. In the past few years there have been numerous studies which point out insecurity of many established cryptographic algorithms if certain side channel information is available to the adversary.

Researchers have considered the possibilities of using different types of side channel information for breaking crypto-systems. Some of the categories of side channel attacks are timing attacks [85], power analysis attacks, differential power analysis attacks [86], electromagnetic radiation attacks [112], fault attacks [14] etc. These attacks utilize the leakages that are

```
Algorithm xtimes(L)
1.  b ← msb(L)
2.  L ← L << 1
3.  if b = 1,
4.      L ← L ⊕ Q
5.  return L
```

Figure 9.1: The algorithm *xtimes*.

associated with any type of computation that takes place in a physical device. Interested readers can consult [98] and [78] for more details on these techniques.

9.2 Side Channel Weakness in the *xtimes* operation

Recall that for $L \in GF(2^n)$, we treat L as a polynomial of degree less than n with coefficients in $\{0, 1\}$. By *xtimes(L)* we mean the multiplication of the monomial $x \in GF(2^n)$ with the polynomial L modulo the irreducible polynomial $q(x)$ representing the field.

The implementation of the *xtimes* operation is very efficient. Let $q(x)$ denote the n degree irreducible polynomial representing the field $GF(2^n)$, let Q be the n bit representation of the polynomial $q(x) \oplus x^n$. Then xL can be realized by the algorithm in Figure 9.1.

This is the most efficient (and the usual) way of implementing *xtimes* where the basic operations involved are a left shift and a conditional xor. As it is obvious from the algorithm that line 4 gets executed only when the most significant bit (MSB) of L is a one. The power utilization in this algorithm would be different in the cases where $\mathsf{msb}(L) = 0$ and $\mathsf{msb}(L) = 1$. This difference of power consumption if measured can give information regarding the MSB of L.

The above described weakness of the *xtimes* operation is widely known. The *xtimes* operation on an n bit string can also be implemented by linear feedback shift registers (LFSR). The vulnerability of LFSRs to side channel attacks has been extensively studied and also there are experimental evidences that such systems leak a lot of information [77] [18]. Thus with the support of the evidence as found in the literature and without going into technical/experimental details of side channel attacks in the rest of this chapter we shall make the following assumption:

Assumption 9.1. *If the operation xL is implemented according to the algorithm xtimes as shown in Figure 9.1 then the MSB of L can be obtained as a side channel information.*

Repeated application of *xtimes* on L can reveal much more information about L. In particular, based on Assumption 1 we can state the following proposition.

```
Algorithm Recover(Q, k)
1.  D ←< d_{n−1}, d_{n−2}, ..., d_0 >← 0^n;
2.  B ← Empty String;
3.  for i = 1 to k,
4.      b ← SCNL(L_1 ← xL) ;(SCNL(L_1 ← xL) gives the MSB of L)
5.      if d_{n−1} = 0,
6.          B ← B||b;
7.      else,
8.          B ← B||b̄
9.      end if
10.     D ← D << 1;
11.     if b = 1,
12.         D ← D ⊕ Q
13.     end if
14.     L ← L_1
15. end for
16. return B
```

Figure 9.2: The algorithm to recover k bits of L.

Proposition 9.1. *If xtimes is applied k ($k \leq n$) times successively on L then the k most significant bits of L can be recovered.*

In lieu of proof of the above proposition we present the procedure to recover the k bits of L in the algorithm in Figure 9.2.

The algorithm *Recover* as shown in Figure 9.2 takes in as input the number of bits to be recovered k along with another n bit string Q which encodes the polynomial $x^n \oplus q(x)$. The algorithm also have access to some side-channel information, which gives it the information of the MSB of L, this is shown as $\mathsf{SCNL}(L_1 \leftarrow xL)$ in line 4. The algorithm, initializes an n bit string D with all zeros (d_i represents the i-th bit of D in the algorithm) and initializes B by an empty string. The output of the algorithm is a k bit string B whose bits would be the same as the k most significant bits of L. It is not difficult to see the correctness of the algorithm. Note that we simulate in D the same changes that takes place in L. After executing the i-th iteration of the for loop we obtain in line 4 the most significant bit of $x^{i-1}L$. This bit would be equal to the i-th bit of L if the MSB of D is zero, otherwise the i-th bit of L would be the complement of the most significant bit of $x^{i-1}L$.

Algorithm EME.Encrypt$_K^T(P)$	**Algorithm** EME.Decrypt$_K^T(C)$
1. Partition P into $P_1, P_2, \ldots, P_m$	1. Partition C into $C_1, C_2, \ldots, C_m$
2. $L \leftarrow xE_K(0^n)$	2. $L \leftarrow xE_K(0^n)$
3. **for** $i \leftarrow 1$ to m **do**	3. **for** $i \leftarrow 1$ to m **do**
4. $PP_i \leftarrow x^{i-1}L \oplus P_i$	4. $CC_i \leftarrow x^{i-1}L \oplus C_i$
5. $PPP_i \leftarrow E_K(PP_i)$	5. $CCC_i \leftarrow E_K^{-1}(CC_i)$
6. **end for**	6. **end for**
7. $SP \leftarrow PPP_2 \oplus PPP_3 \oplus \ldots PPP_m$	7. $SC \leftarrow CCC_2 \oplus CCC_3 \oplus \ldots \oplus CCC_m$
8. $MP \leftarrow PPP_1 \oplus SP \oplus T$	8. $MC \leftarrow CCC_1 \oplus SC \oplus T$
9. $MC \leftarrow E_K(MP)$	9. $MP \leftarrow E_K^{-1}(MC)$
10. $M \leftarrow MP \oplus MC$	10. $M \leftarrow MP \oplus MC$
11. **for** $i \leftarrow 2$ to m **do**	11. **for** $i \leftarrow 2$ to m **do**
12. $CCC_i \leftarrow PPP_i \oplus x^{i-1}M$	12. $PPP_i \leftarrow CCC_i \oplus x^{i-1}M$
13. **end for**	13. **end for**
14. $SC \leftarrow CCC_2 \oplus CCC_3 \oplus \ldots CCC_m$	14. $SP \leftarrow PPP_2 \oplus PPP_3 \oplus \ldots PPP_m$
15. $CCC_1 \leftarrow MC \oplus SC \oplus T$	15. $PPP_1 \leftarrow MP \oplus SP \oplus T$
16. **for** $i \leftarrow 1$ to m **do**	16. **for** $i \leftarrow 1$ to m **do**
17. $CC_i \leftarrow E_K(CCC_i)$	17. $PP_i \leftarrow E_K^{-1}(PPP_i)$
18. $C_i \leftarrow x^{i-1}L \oplus CC_i$	18. $P_i \leftarrow x^{i-1}L \oplus PP_i$
19. **end for**	19. **end for**
20. **return** $C_1, C_2, \ldots, C_m$	20. **return** $P_1, P_2, \ldots, P_m$

Figure 9.3: Encryption and Decryption using EME.

9.3 The Attack on EME

Based on Assumption 9.1 and Proposition 9.1, we shall develop an attack on the EME mode. The EME mode have already been described in Chapter 4, but for convenience we again describe the detailed encryption and decryption algorithms in Figure 9.3.

According to the description in Figure 9.3, EME uses three layers of *xtimes* operation. These operations can leak information about the internal variables L and M. Utilizing this leaked information one can attack the mode. We show two attacks. One attack is a distinguishing attack, which shows that an adversary with oracle access to only the encryption oracle of the mode and the side channel information can distinguish with probability 1 between the real oracle from the one which produces only random strings. We also show a stronger attack, where the adversary can successfully decrypt any given ciphertext C by querying the decryption oracle with ciphertexts other than C. Similarly, without knowledge of the key an adversary can produce a valid cipher text from a given plaintext P and tweak T by querying the encryption oracle (but not at P).

The main observation that makes these attacks possible is Proposition 1. From the algorithm of EME in Figure 9.3 it is clear that on encryption or decryption of m plaintext or ciphertext blocks *xtimes* is applied m times on $E_K(0^n)$ and $m-1$ times on M. This information would be crucial in mounting the attacks. For an m block query the side channel information that

we are interested in is the information regarding M and $E_K(0^n)$ further we shall say that the M-side-channel and L-side-channel gives the side channel information regarding M and $E_K(0^n)$. So after an m block encryption or decryption query the M-side-channel and the L-side-channel will give the $(m-1)$ significant bits of M and m significant bits of $E_K(0^n)$ respectively.

9.3.1 The Distinguishing Attack

First we note down the basic steps followed by the adversary:

1. Apply an arbitrary encryption query of n blocks with an arbitrary tweak.
   - Obtain the n bits of $E_K(0^n)$ from the L-side-channel .
   - Compute $L = xE_K(0^n)$.
2. Apply an encryption query with plaintext L and tweak $x^{-1}L$. Let C be the response of this query.
3. If C is equal to $(1 \oplus x)x^{-1}L$ output EME otherwise output random.

It is easy to see why this attack works. When applying the query in step 1, the adversary recovers the n bits of $E_K(0^n)$. From line 2 of the algorithm of EME in Figure 9.3 we can see that $L = xE_K(0^n)$, thus the adversary can compute L. Following the algorithm of EME in Figure 9.3 we see that for the second query (in step 2) the value of PP_1 would be 0^n hence the value of PPP_1 would be $E_K(0^n)$. As the query consists of only one block, so values of both SP and SC would be zero. So, MP would be computed as $PPP_1 \oplus T$. Note that $T = x^{-1}L = E_K(0^n)$. So MP would be 0^n and CCC_1 would also be 0^n. Thus, we would obtain the output as

$$\begin{aligned} C &= E_K(0^n) \oplus L = E_K(0^n) \oplus xE_K(0^n) \\ &= (1 \oplus x)x^{-1}(xE_K(0^n)) = (1 \oplus x)x^{-1}L \end{aligned}$$

9.3.2 The Stronger Attack

Here we describe a stronger attack, in which assuming that the adversary has access to the M-side-channel and L-side-channel can decrypt any given ciphertext by querying the decryption oracle with ciphertexts other than the ciphertext in question. Before we present the attack we note down an important but obvious characteristics of EME in the proposition below:

Proposition 9.2. *An oracle access to the blockcipher E_K is enough to encrypt any plaintext $P_1||P_2||\ldots||P_m$ with arbitrary tweak T using the EME mode of operation which uses the key K. Similarly, with an oracle access to both E_K^{-1} and E_K one can decrypt any arbitrary ciphertext $C_1, C_2, \ldots, C_m$ with an arbitrary tweak T which has been produced by the EME mode of operation with key K.*

The truth of the above proposition can be easily verified from the algorithm of EME in Figure 9.3. Following the algorithm of EME in Figure 9.3, if we write the dependence of the ciphertext (resp. plaintext) with the plaintext (resp. ciphertext), then the only unknown terms would be of the form $E_K(X)$, for some X, which can be obtained by querying the oracle E_K at X. Similar argument hold for the decryption oracle.

In the attack that follows we shall show that given access to the encryption algorithm of EME along with the L-side-channel and M-side-channel, an adversary can use it as an oracle for the blockcipher $E_K()$. Similarly given an access to the decryption algorithm of EME along with the M-side-channel and L-side-channel, the adversary can use it as an oracle for E_K^{-1}. So, by Proposition 9.2 the adversary can compute ciphertext (plaintext) corresponding to any plaintext (ciphertext) for EME. We describe the steps undertaken by the adversary to obtain $E_K(X)$ given an oracle access to the encryption algorithm of EME in Figure 9.4. We assume that the procedure $\mathbf{AdvSCA}^{\mathrm{EME}_K(.,.)}(X)$ has an access to to the EME encryption algorithm, we also assume that n (the block length of the block cipher E_K) is even [1].

Proposition 9.3. *The procedure $\mathbf{AdvSCA}^{\mathrm{EME}_K(.,.)}(X)$ as shown in Figure 9.4 outputs the first n-1 bits of $E_K(X)$ (which we denote by $\mathsf{take}_{n-1}(E_K(X))$).*

Proof. In step 1 the information regarding L is obtained. For the query in step 2 according to the algorithm (see algorithm of EME in Figure 9.3), we obtain $PP_i = Y$. Thus,

$$\begin{aligned} SP &= PPP_2 \oplus PPP_3 \oplus \cdots \oplus PPP_n \\ &= E_K(Y) \oplus (\underbrace{E_K(Y) \oplus \cdots \oplus E_K(Y)}_{n-2}) = E_K(Y) \end{aligned}$$

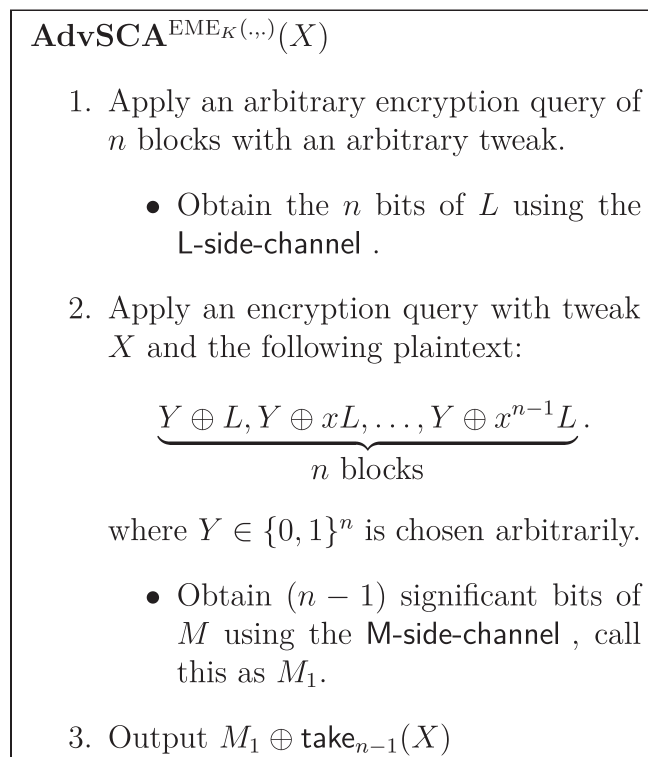
So we have,

$$MP = PPP_1 \oplus SP \oplus X = E_K(Y) \oplus E_K(Y) \oplus X = X$$

Thus, $MC = E_K(X)$, and $M = MP \oplus MC = X \oplus E_K(X)$. Hence $M \oplus X = E_K(X)$. In step 2, only $n-1$ bits of M would be obtained, hence the procedure $\mathbf{AdvSCA}^{\mathrm{EME}_K(.,.)}(X)$ outputs $n-1$ bits of $E_K(X)$. ☐

Now we design another adversary which on given access to the EME encryption algorithm and X and $\mathsf{take}_{n-1}(E_K(X))$ can produce $E_K(X)$ with high probability. We call this procedure as $\mathbf{AdvSCB}^{\mathrm{EME}_K(.,.)}(X, \mathsf{take}_{n-1}(E_K(X)))$, which is shown in Figure 9.5.

[1] This assumption is not strong, as we do not know of any blockcipher whose block length is odd.



AdvSCA$^{\mathrm{EME}_K(.,.)}(X)$

1. Apply an arbitrary encryption query of n blocks with an arbitrary tweak.
   - Obtain the n bits of L using the L-side-channel .
2. Apply an encryption query with tweak X and the following plaintext:

   $$\underbrace{Y \oplus L, Y \oplus xL, \ldots, Y \oplus x^{n-1}L}_{n \text{ blocks}}.$$

   where $Y \in \{0,1\}^n$ is chosen arbitrarily.
   - Obtain $(n-1)$ significant bits of M using the M-side-channel , call this as M_1.
3. Output $M_1 \oplus \mathsf{take}_{n-1}(X)$

Figure 9.4: The side channel adversary with access to EME encryption algorithm producing $n-1$ bits of $E_K(X)$ for arbitrary $X \in \{0,1\}^n$.

> **AdvSCB**$^{\text{EME}_K(.,.)}(X, \mathsf{take}_{n-1}(E_K(X)))$
>
> 1. Apply an arbitrary encryption query of n blocks with an arbitrary tweak.
>   - Obtain the n bits of L using the L-side-channel .
> 2. $Z \leftarrow \mathsf{take}_{n-1}(E_K(X))$
> 3. $S \leftarrow$**AdvSCA**$^{\text{EME}_K(.,.)}(Z||1 \oplus x^{-1}L)$
> 4. Apply a query with tweak $Z||1$ and plaintext $X \oplus L$
>   - Get the output as C
> 5. If $\mathsf{take}_{n-1}(C \oplus L) = S$ output $Z||1$ else output $Z||0$

Figure 9.5: The side channel adversary with access to EME encryption algorithm producing $E_K(X)$.

Proposition 9.4. *Let the procedure* **AdvSCB**$^{\text{EME}_K(.,.)}(X, \mathsf{take}_{n-1}(E_K(X)))$ *be as described in Figure 9.5, then*

$$\Pr[\textbf{AdvSCB}^{\text{EME}_K(.,.)}(X, \mathsf{take}_{n-1}(E_K(X)) = E_K(X)] \geq 1 - \frac{1}{2^{n-1}}$$

Proof. Let us first see what is done in the procedure described in Figure 9.5. The adversary knows X and the $(n-1)$ significant bits of $E_K(X)$. He wants to predict the missing bit of $E_K(X)$. We call the $(n-1)$ bits of $E_K(X)$ as Z. He guesses that the missing bit is 1 and tries to verify if his guess is correct. First let us concentrate on the query made at step 4. The query is made with a tweak $Z||1$ and a single block of plaintext $X \oplus L$. If his guess regarding the last bit of $E_K(X)$ is correct, then $Z||1$ would be $E_K(X)$, and in such a case the response C obtained would be

$$C = L \oplus E_K(E_K(X) \oplus E_K(0^n)) = L \oplus E_K(E_K(X) \oplus x^{-1}L). \tag{9.1}$$

This can be easily verified from the algorithm of EME in Figure 9.3. In the third step of the procedure, S is the output of **AdvSCA**$^{\text{EME}_K(.,.)}$ on $Z||1 \oplus x^{-1}L$. So according to Proposition 9.3

$$S = \mathsf{take}_{n-1}(E_K(Z||1 \oplus x^{-1}L)). \tag{9.2}$$

So if the guess is correct then from eq. (9.1) and eq. (9.2) we get that

$$\mathsf{take}_{n-1}(C \oplus L) = S.$$

Thus, if the last bit of $E_K(X)$ is 1, then the procedure will always output the correct value of $E_K(X)$. On the other hand if the guess is wrong, i.e., the last bit of $E_K(X)$ is zero, then the response to the query in step 4 would be as

$$C \quad = \quad L \oplus E_K(E_K(X) \oplus E_K(0^{n-1}1))$$

And in this case the check in step 5 will pass with a probability less than $\frac{1}{2^{n-1}}$. Thus, the probability with which a correct guess can be made is greater than $(1 - \frac{1}{2^{n-1}})$. □

So using the procedures described in Fig 9.4 and Fig 9.5 the side channel adversary can compute $E_K(X)$ for a X of his choice with very high probability. Using the same technique the adversary can compute $E_K^{-1}(X)$ for any X given access to the decryption algorithm of EME.

Thus, we can conclude that given access to the encryption and decryption algorithms of EME and the relevant side channel information, an adversary can compute the encryption of a plain-text P of his choice without querying the encryption algorithm at P. Similarly, (s)he can decrypt any ciphertext C without querying the decryption algorithm at C.

9.4 EME2 Mode of Operation

EME2 adds certain functionalities which are not present in EME, for example, EME2 can handle arbitrary long messages (recall that EME cannot securely encrypt messages larger than n blocks long). Additionally EME2 can handle arbitrarily long tweaks (EME can only handle n bit tweaks, where n is the block length of the block cipher). The description of EME2 is a bit different from that of EME, the primary difference being that it encrypts the tweak. The description of EME2 is given in Figure 9.6. The description given in Fig 9.6 is not the full description, if we assume that both the tweak length and the block length are multiples of the block length of the block cipher and the number of blocks are less less or equal to the block length of the block cipher, then the original description of EME2 translates to the description given in Figure 9.6. But the above stated restrictions are not valid for the EME2 mode, the full description of the mode can handle plaintexts which do not satisfies these restrictions. For the full description of the mode see [131].

The main difference of the restricted description of EME2 compared to EME is in the handling of the tweak, as it can handle arbitrarily long tweaks and converts an arbitrary long tweak to a n bit value which is used in the mode. Also, EME2 uses three n bit keys,

Algorithm EME2.Encrypt$^{T}_{K_1,K_2,K_3}(P)$
1. Partition P into $P_1, P_2, \ldots, P_m$
2. **if** len(T)= 0 **then** $T^* = E_{K_1}(K_3)$
3. **else**
4. Partition the tweak T to $T_1, T_2, \ldots, T_r$
5. **for** $i = 1$ to r
6. $K_3 \leftarrow xK_3$
7. $TT_i \leftarrow E_{K_1}(K_3 \oplus T_i) \oplus K_3$
8. $T^* = TT_1 \oplus TT_2 \oplus \cdots \oplus TT_r$
9. **endif**
10. $L \leftarrow K_2$
11. **for** $i = 1$ to m
12. $PPP_i \leftarrow E_{K_1}(L \oplus P_i)$
13. $L \leftarrow xL$
14. **end for**
15. $MP \leftarrow PPP_1 \oplus PPP_2 \oplus \cdots \oplus PPP_m \oplus T^*$
16. $MC \leftarrow E_{K_1}(MP)$
17. $M \leftarrow MP \oplus MC$
18. **for** $i \leftarrow 2$ to m
19. $M \leftarrow xM$
20. $CCC_i \leftarrow PPP_i \oplus M$
21. **end for**
22. $CCC_1 \leftarrow MC \oplus CCC_2 \oplus \cdots \oplus CCC_m \oplus T^*$
23 $L \leftarrow K_2$
24. **for** $i \leftarrow 1$ to m
25. $C_i \leftarrow E_{K_1}(CCC_i) \oplus L$
26. $L \leftarrow xL$
27. **end for**
28. **return** $C_1, C_2, \ldots, C_m$

Figure 9.6: Encryption using EME2.

for processing the tweak it uses the key K_3, and the value of L which is used to mask the plain-texts and the ultimate outputs is first set to the value of K_2 and the bulk encryption is done by the key K_1.

9.5 A Distinguishing Attack on EME2

As evident from the algorithm in Figure 9.6 there are four layers of *xtimes* operations performed in the algorithm. Thus, based on Assumption 9.1 and Proposition 9.1 one can get information about K_2, K_3 and M from the algorithm. We will call them as the K2-side-channel, K3-side-channel and M-side-channel respectively. Using these side-channel information one can mount a distinguishing attack on EME2. The adversary performs the following steps:

1. Apply an n block encryption query with a no tweak.
   - Obtain K_2 using the K2-side-channel.
2. Apply an arbitrary encryption query with a n block tweak.
   - Obtain K_3 using the K3-side-channel.
3. Apply an encryption query with a one block tweak where $T = 0$, and a $n-1$ block message $P = P_1||P_2||\ldots||P_{n-1}$, where $P_i = x^{i-1}K_2 \oplus xK_3$
   - Obtain $(n-2)$-bits of M using the M-side-channel . Call this as M_1.
4. Apply an encryption query with a one block tweak $T = 0$, and one block of message $P = P_1 = xK_3 \oplus K_2$
   - Obtain the corresponding ciphertext and call it C_1.
5. If the first $(n-2)$ bits of $C_1 \oplus xK_3 \oplus K_2$ is equal to M_1 output "EME2" otherwise "random".

To see why this attack works, according to the algorithm in Figure 9.6, for query 3 we have the following:

Firstly, as there is a single block of tweak and the tweak is zero hence we get

$$T^* = E_{K_1}(xK_3) \oplus xK_3 \tag{9.3}$$

further, from line 12 of Figure 9.6 we have, for all $i = 1, \ldots, n-1$,

$$PPP_i = E_K(x^{i-1}K_2 \oplus P_i) = E_K(x^{i-1}K_2 \oplus x^{i-1}K_2 \oplus xK_3) = E_K(xK_3)$$

Now, according to lines 15 of the algorithm in Figure 9.6 we have

$$\begin{aligned} MP &= PPP_1 \oplus PPP_2 \oplus \ldots \oplus PPP_{n-1} \oplus T^* & (9.4) \\ &= E_K(xK_3) \oplus T^* & (9.5) \\ &= E_K(xK_3) \oplus E_{K_1}(xK_3) \oplus xK_3 & (9.6) \\ &= xK_3 & (9.7) \end{aligned}$$

Equation (9.5) follows from eq. (9.4) because the PPP_is are all equal and we assume that n is even. Equation (9.6) follows from eq. (9.5) by substituting the value of T^* in eq. (9.3). Thus, the value of M gets computed as

$$M = MP \oplus E_{K_1}(MP) = xK_3 \oplus E_{K_1}(xK_3) \quad (9.8)$$

Thus the value M_1 obtained in step 3 is the first $(n-1)$ bits of M as in eq. (9.8).

In the query in step 4, the tweak is again single block and its value is zero, thus the value of T^* would be same as in eq. (9.3), and $PPP_1 = E_{K_1}(xK_3)$. Thus we would have $M = xK_3 \oplus E_{K1}(xK_3)$, which is same as the value of M obtained as side channel information from query 3 (eq. (9.8)). Continuing, according to the algorithm in Figure 9.6 we would have the cipher text C_1 computed as $C_1 = E_{K_1}(xK_3) \oplus K_2$.

So, we have

$$C_1 \oplus xK_3 \oplus K_2 = xK_3 \oplus E_{K_1}(xK_3) \quad (9.9)$$

So comparing eq. (9.9) and eq. (9.8), we obtain $C_1 \oplus xK_3 \oplus K_2 = M$.

The $(n-2)$ significant bits of M has been obtained from the side channel information in query 3. So if the check in step 5 is successful then with overwhelming probability the adversary can say that he is communicating with EME2.

The strong attack discussed in Section 9.3.2 for EME cannot be applied in the case of EME2. The strong attack for EME utilizes the fact that by obtaining the value of M one can obtain the block-cipher encryption of the tweak. As in EME the tweak can be freely chosen, hence one can get encryption of any string by suitably choosing the tweak. In EME2, the tweak is encrypted, this prevents one to apply the strong attack applicable to EME.

9.6 Final Remarks

We presented some attacks on EME and EME2 assuming that *xtimes* leaks some information. These attacks does not contradict the claimed security of the modes, as the security definition

and the security proofs for these modes does not assume any side-channel information being available to the adversary. Also the consequences of these attacks shown are not immediate. But, it points out that using *xtimes* indiscriminately may give rise to security weakness in true implementations.

This Chapter can be seen as a starting point for the study of side channel vulnerabilities of TES. The theoretical evidence presented here needs to be backed by experimental support to know about the severity of the weaknesses and also to design proper counter measures. We believe that the same techniques can be possibly applied to other modes, and we plan to explore such possibilities in near future.

Chapter 10

A New Model for Disk Encryption

We start from different ideological positions. For you to be a Communist or a Socialist is to be totalitarian; for me no. On the contrary, I think Socialism frees man.

Salvador Allende

Till now it has been argued that length preservation is one of the most important characteristic required for an encryption scheme to be suitable for the application of low level disk encryption. We already described that the reason for this is the assumption that the disk sectors are of fixed length and there is no extra space to accommodate any length expansion of the cipher text. In this Chapter we look into this requirement more closely. We argue that the size of a sector is always bigger than the user data that it stores, as it requires to store more information other than the user data for its proper functioning. The user data always under goes an error correction coding before it is stored. This transform obviously adds to the original length of the data, and the sectors are equipped to accommodate this expansion. As we are unable to think of a disk without the capacity of error correction, we think that in the coming years a hard disk without encryption capabilities would become un-thinkable. It is not unrealistic to think of a scenario where a disk is suitably formatted to accommodate a specific encryption scheme which results in cipher-text expansion. In particular in this Chapter we argue that a special class of authenticated encryption schemes called Deterministic Authenticated Encryption (DAE) schemes can be suitable for the application of disk encryption, and they would be a much more efficient alternative compared to the existing tweakable enciphering schemes. Furthermore we propose a new DAE which we call BCTR. BCTR is designed specifically to be used in the application of disk encryption. We prove security of BCTR and also provide a very efficient hardware implementation of BCTR. The experimental results show that BCTR is far more efficient than the TES that we report in this thesis.

The rest of this Chapter is organized as follows. In Section 10.1 we present the syntax of DAE along with a generic construction and the security definition. In Section 10.2 we analyze

the length preserving requirement for a disk encryption scheme in details and argue the pros and cons for using a DAE for this application. In Section 10.3 we describe the encryption and decryption algorithm for BCTR and compare operation counts of BCTR with existing DAEs and TESs. In Section 10.4 we state the security theorems of BCTR. In Section 10.5 we provide description of a FPGA circuit which implements BCTR and provide experimental performance data for BCTR along with comparisons with other efficient implementations of disk encryption schemes reported in this thesis. In Section 10.6 we provide complete proofs of the security theorems stated in Section 10.4.

10.1 Deterministic Authenticated Encryption Schemes

We already discussed in Section 2.5.2 that authenticated encryption schemes are a class of encryption schemes which provide security both in terms of privacy and authentication. The ciphertext produced by these schemes includes an authentication tag, and this authentication tag can be used to verify the authenticity of the ciphertext, i.e., to check if the ciphertext was altered. There have been many proposals for authenticated encryption schemes. Among them, some are very efficient and require a little more than one block-cipher call per block of message, and they provide security against chosen cipher text attacks. These schemes are nonce based, they require a quantity called nonce, which is non-repeating i.e., each plaintext needs to be encrypted with a different nonce for the purpose of security.

Deterministic authenticated encryption schemes are a class of authenticated encryption schemes which does not require nonces. They were first proposed in [119]. There were two principal motivations behind DAEs. Firstly, it was argued that DAEs may be suitable for encrypting keys, this problem is formally called the key wrap problem. The other motivation was to use DAEs where there is a possibility of "nonce misuse", i.e., where there is a possibility of accidental or adversarial nonce repetition. Note that in a nonce based AE, if a nonce is repeated then the security of the scheme is completely compromised.

There have been numerous proposals of AEs, among them there are a class of block cipher based proposals which can provide the services of both privacy and authentication for a message of m blocks by using about m block cipher calls. These schemes are called one pass AE schemes and there are many instances of such schemes as IAPM [79], OCB [118], OCB3 [87] etc. DAE schemes are inherently slower than one pass AE schemes, as in DAE two separate passes over the data is always required.

Formally, a DAE is a tuple $\Psi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$, where $\mathcal{K}$ is the key space (also viewed as the key generation algorithm, which is a randomized algorithm which samples a key from the key space), and $\mathcal{E}$ and $\mathcal{D}$ are deterministic algorithms called as encryption and decryption algorithms respectively. Let $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{H}$ be the message space, cipher space and the header space respectively associated with a DAE Ψ. The encryption algorithm $\mathcal{E}$ takes an input

in $\mathcal{K} \times \mathcal{H} \times \mathcal{X}$ and returns an element in $\mathcal{Y}$, and the decryption algorithm takes an input in $\mathcal{K} \times \mathcal{H} \times \mathcal{Y}$ and returns either an element in $\mathcal{X}$ or a special symbol $\perp$. As usual, we shall often write $\mathcal{E}_K^H(X)$ or $\mathcal{E}_K(H, X)$ to denote $\mathcal{E}(K, H, X)$, and $\mathcal{D}_K^H(Y)$ or $\mathcal{D}_K(H, Y)$ to denote $\mathcal{D}(K, H, Y)$. The message space and the cipher space are non-empty sets containing binary strings, i.e., $\mathcal{X}, \mathcal{Y} \subseteq \{0,1\}^*$, and the header space $\mathcal{H}$ contains vectors whose elements are binary strings, i.e., $\mathcal{H} \subseteq \{0,1\}^{**}$. It is required that $\mathcal{D}_K^H(Y) = X$ if $\mathcal{E}_K^H(X) = Y$ and $\mathcal{D}_K^H(Y) = \perp$ it no such $H \in \mathcal{H}$ and $X \in \mathcal{X}$ exist such that $E_K^H(X) = Y$. It is assumed that for any $K \in \mathcal{K}$, $X \in \mathcal{X}$ and $H \in \mathcal{H}$, $|\mathcal{E}_K^H(X)| = |X| + e(X, H)$, where $e : \mathcal{X} \times \mathcal{H} \to \mathbb{N}$ is called the expansion function of the DAE scheme and depends only on the length of X, the number of components of H and the length of each component of H. $s = \min_{X \in \mathcal{X}, H \in \mathcal{H}} \{e(X, H\}$ is called the stretch of the DAE scheme. A DAE is length preserving if $e(X, H) = 0$ for all $X \in \mathcal{X}$ and $H \in \mathcal{H}$. Note that the header in a DAE is the same as a tweak in a tweakable enciphering scheme, thus a length preserving DAE is a tweakable enciphering scheme.

The first proposed DAE was Synthetic Initialization Vector (SIV) by Rogaway and Shrimpton [119]. The SIV mode construction uses a secure IV based privacy only encryption mode like the counter mode or the CBC mode along with a special type of pseudorandom function which takes as input a vector of binary strings. Let $\{0,1\}^{**}$ denote the space for all vectors of binary strings, and if X and Y are vectors such that $X = (X_1, \ldots, X_m)$ and $Y = (Y_1, \ldots, Y_{m'})$, then $[[X, Y]] = (X_1, \ldots, X_m, Y_1, \ldots, Y_{m'})$. The SIV mode $\mathsf{SIV}[\boldsymbol{F}, \mathsf{Priv}]$ is constructed using a pseudo-random function $\boldsymbol{F} : \mathcal{K}_1 \times \{0,1\}^{**} \to \{0,1\}^n$ and a privacy only encryption scheme $\mathsf{Priv} = (\mathcal{K}_2, \boldsymbol{E}, \boldsymbol{D})$ where $\mathcal{K}_2$ is the key space, $\boldsymbol{E} : \mathcal{K}_2 \times \{0,1\}^n \times \{0,1\}^* \to \{0,1\}^*$ is the encryption algorithm and $\boldsymbol{D} : \mathcal{K}_2 \times \{0,1\}^n \times \{0,1\}^* \to \{0,1\}^*$ is the decryption algorithm. For an initialization vector IV, and key K the encryption and decryption algorithms are written as $\boldsymbol{E}_K^{IV}(.)$ and $\boldsymbol{D}_K^{IV}(.)$ respectively.

The $\mathsf{SIV}[\boldsymbol{F}, \mathsf{Priv}]$ construction is shown in Figure 10.1. The construction in Figure 10.1 takes in two keys K_1 and K_2 which are used in the pseudorandom function $\boldsymbol{F}$ and the IV based encryption scheme $\boldsymbol{E}$ respectively. In addition it takes in the header T (the header can be treated as a tweak, and in the further discussion we shall consider the header as a tweak) and the message P. In the procedure the tweak and the message are converted into τ using the pseudorandom function $\boldsymbol{F}$ and this τ is used as an IV for the privacy only encryption scheme $\boldsymbol{E}$. The encryption of the message P is done using $\boldsymbol{E}$. The ciphertext produced by $\boldsymbol{E}$ and τ are together considered as the ciphertext for SIV.

10.1.1 Security of DAEs

Let $\Psi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a DAE scheme. Let $\mathcal{A}$ be an adversary attacking Ψ. The adversary has access to the oracle $\mathcal{O}$. $\mathcal{O}$ can either be $\mathcal{E}_K(.,.)$, where K is generated uniformly at random from $\mathcal{K}$ or it can be $\$(.,.)$ which returns random strings of length equal to the length of the ciphertext. The adversary can query the oracle with a query of the form (T, M), where M,

Algorithm $\mathsf{SIV}.\mathrm{Encrypt}^{T}_{K_1,K_2}(P)$	**Algorithm** $\mathsf{SIV}.\mathrm{Decrypt}^{T}_{K_1,K_2}(C,\tau)$
1. $\tau \leftarrow \boldsymbol{F}_{K_1}([[P,T]])$;	1. $P = \boldsymbol{D}^{\tau}_{K_2}(C)$;
2. $C \leftarrow \boldsymbol{E}^{\tau}_{K_2}(P)$;	2. $\tau' = \boldsymbol{F}_{K_1}([[P,T]])$;
3. **return** $C\|\|\tau$;	4. **if** $\tau' = \tau$ **return** P else **return** $\perp$;

Figure 10.1: Encryption and decryption using $\mathsf{SIV}[\boldsymbol{F}, \mathsf{Priv}]$. K_1 is the key for the PRF and K_2 the key for the privacy only encryption scheme $\mathbf{E}$. The header T is a vector of binary strings.

T denotes the message and tweak respectively and and thus get back the responses from the oracle. $\mathcal{A}$ has the restriction that it cannot repeat a query. The privacy advantage of adversary $\mathcal{A}$ is defined as

$$\mathbf{Adv}^{\text{dae-priv}}_{\Psi}(\mathcal{A}) = \Pr\left[K \stackrel{\$}{\leftarrow} \mathcal{K} : \mathcal{A}^{\mathcal{E}_K(.,.)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\$(.,.)} \Rightarrow 1\right]. \qquad (10.1)$$

To define the authentication advantage we consider that $\mathcal{A}$ is given access to the encryption oracle $\mathcal{E}_K(.,.)$ and queries it with plaintexts of his choice and obtains the corresponding ciphertexts. Finally, $\mathcal{A}$ outputs a forgery, which consists of a tweak and a ciphertext $(T, \mathcal{C})$. $\mathcal{A}$ is said to be successful if $\mathcal{D}_K(T, \mathcal{C}) \neq \perp$. For querying the encryption oracle $\mathcal{A}$ follows the same restriction that the adversary does not repeat a query additionally $\mathcal{A}$ cannot output $(T, \mathcal{C})$ as a forgery if he obtained $\mathcal{C}$ as a response for his query (T, X) to the encryption oracle. The last restriction is to rule out trivial win. The authentication advantage of the adversary $\mathcal{A}$ is defined as the probability that $\mathcal{A}$ does a successful forgery, in other words

$$\mathbf{Adv}^{\text{dae-auth}}_{\Psi}(\mathcal{A}) = \Pr[\mathcal{A}^{\mathcal{E}_K(.,.,.)} \text{ forges }] \qquad (10.2)$$

We consider Ψ to be secure, if both $\mathbf{Adv}^{\text{dae-priv}}_{\Psi}(\mathcal{A})$ and $\mathbf{Adv}^{\text{dae-auth}}_{\Psi}(\mathcal{A})$ are small for every computationally bounded adversary $\mathcal{A}$.

In [119] the two notions of privacy and authentication as depicted in equations (10.1) and (10.2) were combined to give a unified security definition where the advantage of an adversary $\mathcal{A}$ in breaking a DAE scheme Ψ was defined as

$$\mathbf{Adv}^{\text{dae}}_{\Psi}(\mathcal{A}) = \Pr\left[K \stackrel{\$}{\leftarrow} \mathcal{K} : \mathcal{A}^{\mathcal{E}_K(.,.),\mathcal{D}_K(.,.)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\$(.,.),\perp(.,.)} \Rightarrow 1\right], \qquad (10.3)$$

where the oracle $\perp(.,.)$ always returns $\perp$. In [119] it was shown that the unified definition of security is equivalent to the separate notions of privacy and authenticity. In our case we

would stick to the separate definitions of privacy and authenticity, as that makes our proofs simpler.

10.2 In Support of Tagged Mode for disk Encryption

Traditionally disk sectors were 512 bytes long, which means that each sector can store 512 bytes of user data, but this does not mean that each sector is physically 512 bytes long. The user data is not stored in the format provided by the operating system to the disk. An important transformation that the data written in a sector undergoes is error correction coding and this obviously creates an expansion in length of the original data. So the 512 byte data which an operating system sends to the disk controller for writing occupies more than 512 bytes in the physical disk as it is appropriately coded for error correction within the disk controller. Moreover, each sector stores other data and gaps which are necessary for data recovery and functioning of the disk.

The formatted capacity of a hard disk is the amount of user data it can hold, but physically a hard disk has more space than its formatted capacity. This extra space required for error correction and for storing other information for functioning of the disk is called the format overhead of the disk. The manufacturers generally does not publish the format overhead of the disks that are available commercially in the market, the type and amount of extra data they store in the sectors is a trade secret. A letter by Fujitsu Corporation in 2003 declared that the then format overhead of their commercial hard disks were approximately 15% of the sector size, and in the letter it was predicted that maintaining hard disks with 512 byte sector sizes would make the format overhead to grow to 30% within 2006 [34](Page 15.). Thus, the modern day disks do allow data expansion, and in fact data expansion is inevitable as storage without error correction coding is not acceptable.

10.2.1 Which Encryption Scheme?

Till today the need of a tag less encryption scheme (length preserving) for the application of disk encryption has been emphasized, and as a solution two basic strategies have been proposed. These strategies as described in Section 4.6 have been classified as wide block modes and narrow block modes by the IEEE working group on storage security who are in the process of formulating a standard for disk encryption. The wide block modes are realized by a cryptographic primitive called tweakable enciphering scheme and they are well accepted as they provide the highest form of security possible for length preserving encryption schemes. If the need for a length preserving encryption scheme is relaxed then there exist numerous other schemes which are not length preserving that can be considered.

Authenticated encryption schemes are a class of encryption schemes which provide security

both in terms of privacy and authentication. The ciphertext produced by these schemes includes an authentication tag, and this authentication tag can be used to verify the authenticity of the ciphertext, i.e., if the ciphertext was altered. There have been many proposals for authenticated encryption schemes. Among them, some are very efficient and require a little more than one block-cipher call per block of message, and they provide security against chosen cipher text attacks. These schemes are nonce based, they require a quantity called nonce, which is non-repeating i.e., each plaintext needs to be encrypted with a different nonce for the purpose of security.

Deterministic authenticated encryption schemes were developed as a solution to the key wrap problem. They are a class of authenticated encryption schemes which does not require nonces. They are not length preserving, but they provide the same security as that of tweakable enciphering schemes. Tweakable enciphering schemes are secure in the sense of strong pseudorandom permutations, whereas deterministic authenticated encryption schemes are pseudorandom injections [119].

Authenticated encryption schemes require nonces for security. Thus, in a ciphertext produced by an authenticated encryption scheme, the nonce is also a part of the ciphertext thus making the ciphertext further long. In a real life scenario where one uses a 128 bit nonce and a 128 bit tag, the expansion of the ciphertext would be 256 bits for an AE scheme but 128 bits for a DAE scheme. Though the security provided by a AE is stronger than that provided by a DAE, but DAE security is same as in a TES [119], which is well accepted in case of disk encryption. Thus, we propose the use of deterministic authenticated encryption schemes as a possible solution to the disk encryption problem. In the next subsection we discuss about the advantages and disadvantages for using such a scheme.

It is to be noted that the way hard disks are formatted today, it does not provide extra space for cryptographic materials like authentication tags. Thus, to build an encryption functionality over a disk which is available today one needs to stick to length preserving schemes like TES. The discussion that follows tries to argue on the suitability of DAE as an encryption mechanism when hard disks are formatted in a way so that it provides space for the ciphertext expansion, and we already argued that this is not physically impossible. Thus, our proposal of using DAE does not come as an alternative to the use of TES in the currently available hard disks. Our study tries to make a realistic future proposal considering the fact that efficient cryptographic functionalities on storage medias are a call of the day, and in near future storage media would be manufactured with the consideration of easy use of efficient cryptographic algorithms on them. Another point to note is that if length expansion is allowed then it would be better to use a scheme which results in minimal ciphertext expansion, in this context DAEs are better than AEs as DAEs would not require storage of nonces, though DAEs are inherently two pass schemes and are less efficient than many existing one pass AEs.

Table 10.1: Extra format overhead

Sector size (in bytes)	Tag size (in bits)		
	64	96	128
512	1.56%	2.34%	3.13%
4096	0.19%	0.29%	0.39%
8192	0.09%	0.14%	0.19 %

10.2.2 Gains and Loses in using DAE for disk encryption.

Loss of space: As we argued that disk sectors are technically capable of accommodating extra information, but of course the extra information stored in form of a tag would mean loss of space. A DAE scheme produces a tag of fixed length for each plaintext encrypted. Thus if one fix the tag length to τ bits then for each plaintext of ℓ bits the ciphertext would be $(\ell+\tau)$ bits long. So the more the length of the plaintext the less would be the percentage loss of space. In case of disk encryption the length of the plaintext is also fixed and same as the length of a disk sector. Till last year disk sectors were 512 bytes long. 512 byte long sectors are not optimal anymore as the the aerial density of disk are ever increasing, the industries have decided to increase the length of disk sectors to 4096 bytes, and starting from this year all leading disk manufacturers are supposed to supply disks with the bigger sector sizes (4096 bytes) [34]. This increase in the sector size would make the use of DAE as a disk encryption algorithm more appealing, as the bigger the sector size the total loss incurred in space for storing the tags would be lesser. In Table 10.1 we show the amount of loss in space that would take place taking into account various disk sizes and tag lengths.

Table 10.1 shows that the extra format overhead for using a DAE with tag length 128 bits would be a negligible 0.4% given the sectors are 4096 bytes long. This is negligible considering modern disks probably needs to tolerate around 15% of format overhead (according to the letter by Fujitsu Corporation, which was stated before).

Gain in efficiency: We have discussed in details that current constructions of tweakable enciphering schemes fall into three basic categories: Encrypt-Mask-Encrypt type, Hash-ECB-Hash type, and Hash-Counter-Hash type. CMC [65], EME [66], and EME* [63] fall under the Encrypt-Mask-Encrypt group. PEP [29], TET [64], and HEH [122] fall under the Hash-ECB-Hash type; and XCB [101], HCTR [141], HCH [31], HMCH [125] fall under the Hash-Counter-Hash type. These constructions use a block cipher as the basic primitive, and in addition, some schemes utilize a universal hash function which is a Wegman-Carter type polynomial hash or a more efficient variant known as Bernstein-Rabin-Winnograd polynomials [125]. The constructions of the Hash- Counter-Hash and Hash-Encrypt-Hash type invoke two polynomial hash functions and a layer of encryption in-between. The Encrypt-Mask-Encrypt structure consists of two layers of encryption with a light weight masking

layer in-between. So, the main computational overhead of the Encrypt-Mask-Encrypt architecture is given by the block cipher calls, whereas for the other two classes of constructions, both block cipher calls and finite field multiplications amount for a significant portion of the total computational cost. Known DAE constructions require less computational overhead. As with a single layer of polynomial hashing and a layer of encryption it is possible to realize a DAE, whereas in case of TES constructions of the Hash-ECB-Hash type and Hash-Counter-Hash type require two layers of hashing (the operation counts of the existing TES and DAE constructions are provided in Tables 10.2 and 10.3 respectively).

True Authentication: DAE are authenticated encryption schemes and ciphertext produced by such schemes have authentication tags, using these tags the authenticity of the ciphertext can be verified. As TES are length preserving they cannot provide true authentication, the only guarantee that these schemes can provide is that an invalid ciphertext would get decrypted to a random plaintext and thus a high level application which uses the data would be able to detect that the cipher text was tampered. In practical terms this may have some difficulty as if the plaintext was truly random then there would be no way to detect tampering in the ciphertext. Thus, a TES is unable to provide true authentication and the type of authentication that they provide has been termed as "poor man's authentication" in [54].

10.3 BCTR: A new DAE suitable for disk encryption

We present a new DAE named BCTR. BCTR is designed so that it is suitable for low level disk encryption. In particular, the message space of BCTR is $\{0,1\}^{nm}$ where n is the block length of the underlying block cipher. Hence a message for BCTR is of fixed length containing m blocks of n bits. The tweak space is $\{0,1\}^n$ and the cipher space is $\{0,1\}^{mn+\ell}$ where ℓ is the desired tag length. Thus, the construction inherently has restrictions on the message length and the tweak length. But these restrictions are not of any significance for the disk encryption application as the messages are always of fixed length which is the formatted size of the sector and which in turn is either 512 bytes or in the coming days would be 4096 bytes. The sector address is treated as the tweak, thus the restriction in the tweak length is also of no consequence.

The encryption and decryption algorithms using BCTR are shown in Figure 10.2. The construction requires two keys, a key h for the BRW polynomial and a key K for the block-cipher. Other than the keys the encryption algorithm takes in the plain text and the tweak and returns a cipher text and the decryption algorithm takes in the cipher text and tweak and returns the plaintext. The details of the working of the algorithm are self explanatory as depicted in Figure 10.2. For this construction it is required that the irreducible polynomial representing the field $\mathbb{F}_{2^n}$ be primitive.

Algorithm BCTR.$\mathcal{E}^T_{h,K}(P_1, \ldots, P_m)$	**Algorithm** BCTR.$\mathcal{D}^T_{h,K}(C_1, \ldots, C_m, \tau)$
1. $\alpha = E_K(0)$; $\beta = E_K(1)$;	1. $\alpha = E_K(0)$; $\beta \leftarrow E_K(1)$;
2. $\gamma \leftarrow h \cdot \mathsf{BRW}_h(P_1\|\|P_2\|\| \ldots \|\|P_m\|\|T)$	2. **for** $j = 1$ to m,
3. $\tau \leftarrow E_K(\gamma \oplus \alpha)$;	3. $R_j \leftarrow E_K(\tau \oplus x^j\beta)$;
4. **for** $j = 1$ to m	4. $P_i \leftarrow C_j \oplus R_j$
5. $R_j \leftarrow E_K(\tau \oplus x^j\beta)$	5. **endfor**
6. $C_j \leftarrow R_j \oplus P_j$	6. $\gamma \leftarrow h \cdot \mathsf{BRW}_h(P_1\|\|P_2\|\| \ldots \|\|P_m\|\|T)$;
7. **endfor**	7. $\tau' \leftarrow E_K(\gamma \oplus \alpha)$
8. return $(C_1\|\|C_2\|\| \ldots \|\|C_m\|\|\tau)$	8. **if** $\tau' = \tau$ **return** $(P_1, \ldots, P_m)$ else **return** $\perp$;

Figure 10.2: Encryption and decryption using BCTR.

As evident the construction of BCTR depends on BRW polynomials. Structural properties of BRW polynomials were extensively studied in Chapter 6, in particular it was said that for $m \geq 2$, $\mathsf{BRW}_h(X_1, \ldots, X_m)$ can be computed using $\lfloor m/2 \rfloor$ multiplications and $\lg m$ squarings. Thus one can infer that to encrypt m block of messages BCTR requires $\lfloor \frac{m+1}{2} \rfloor + 1$ multiplications and $\lg(m+1)$ squarings. Computing squares in binary fields are much more efficient than multiplication. In addition it requires $(m+3)$ block-cipher calls. Out of these $(m+3)$ block-cipher calls two can be pre-computed, but this would amount to the requirement of storage of key related materials which is not recommended. The construction requires two keys, h the key for the BRW polynomial and K the block-cipher key. Requirement of a single block-cipher key is what is important, as for multiple block-cipher keys one needs to have different key schedules which may make an implementation inefficient when implemented in hardware. One can probably generate the key h using the block-cipher key and still obtain a secure construction, but this would generally mean one more block-cipher call or a storage of key related material which is same as storage of an extra key. So, we decided to keep the block-cipher key independent of the key for the BRW polynomial. The constructions requires two passes over the data, one for computing the tag τ and other for generating the ciphertext. The ciphertext is generated using a counter type mode of operation which can be parallelized, also α and β can be computed in parallel with the BRW polynomial. Thus the construction offers flexibility for efficient pipelined implementation. Also, for an efficient hardware implementation the only non-trivial blocks that are needed to be implemented are a finite field multiplier and an encryption only block-cipher. So, it is expected that a hardware implementation will have a small footprint.

Comparisons: In Table 10.2 we compare the number of operations required for tweakable enciphering schemes for fixed length messages which uses n bit tweaks with the number of operations required for BCTR.

From Table 10.2 we can see that encrypting with BCTR would be much more efficient than

Table 10.2: Comparison of BCTR with tweakable enciphering schemes for fixed length messages which uses n bit tweak. [BC]: Number of block-cipher calls; [M]: Number of multiplications, [BCK]: Number of blockcipher keys, [OK]: Other keys, including hash keys.

Mode	[BC]	[M]	[BCK]	[OK]
CMC [65]	$2m+1$	–	1	–
EME [66]	$2m+2$	–	1	–
XCB [101]	$m+1$	$2(m+3)$	3	2
HCTR [141]	m	$(2m+1)$	1	1
HCHfp [31]	$m+2$	$2(m-1)$	1	1
TET [64]	$m+1$	$2m$	2	3
Constructions in [125] using normal polynomials	$m+1$	$2(m-1)$	1	1
Constructions in [125] using BRW polynomials	$m+1$	$2+2\lfloor(m-1)/2\rfloor$	1	
BCTR	$m+3$	$1+\lfloor(m+1)/2\rfloor$	1	1

encrypting by the existing tweakable enciphering schemes which uses polynomial hashing. But, it is to be noted that the computational efficiency comes at the cost of more space requirement, but as discussed in Section 10.2.2 the extra space requirement would be insignificant in case of sectors of 4096 bytes.

In Table 10.3 we compare BCTR with the existing DAE schemes. SIV is a DAE which uses CMAC (which is a block cipher based algorithm for message authentication) along with the CBC or CTR mode of operation. This construction is fully block cipher based and does not require any multiplications. But as seen from Table 10.3 SIV requires twice the number of block cipher calls compared to BCTR. HBS and BTM are DAEs which requires both polynomial multiplications and block cipher calls. The number of block cipher calls required for BCTR is comparable to the number of block cipher calls in HBS and BTM, but BCTR requires about half the number of multiplications compared to both HBS and BTM.

10.4 Security of BCTR

The following theorems suggests that $BCTR$ is a secure DAE.

Theorem 10.1. *Let* $\Pi = Perm(n)$. *Let* $\mathcal{A}$ *be an adversary attacking* BCTR[Π] *who asks* q *queries, then*

Table 10.3: Comparison between BCTR and DAE schemes for encrypting m blocks of messages. In the DAE schemes the operation counts are based on only one block of tweak. [BC]: Number of block-cipher calls; [M]: Number of multiplications, [BCK]: Number of blockcipher keys, [OK]: Other keys, including hash keys.

Mode	$[BC]$	$[M]$	[BCK]	[OK]
SIV [119]	$2m+3$	–	2	–
HBS [75]	$m+2$	$m+3$	1	–
BTM [74]	$m+3$	m	1	–
BCTR	$m+3$	$1+\lfloor(m+1)/2\rfloor$	1	1

$$\mathbf{Adv}^{\text{dae-priv}}_{\text{BCTR}[\Pi]}(\mathcal{A}) \leq \frac{14m^2q^2}{2^n} \tag{10.4}$$

$$\mathbf{Adv}^{\text{dae-auth}}_{\text{BCTR}[\Pi]}(\mathcal{A}) \leq \frac{1}{2^n} + \frac{18m^2q^2}{2^n} \tag{10.5}$$

Theorem 10.2. *Let $E : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a block-cipher secure in the PRP sense. Let $\mathcal{A}$ be an adversary attacking* BCTR$[E]$ *who asks q queries, then their exists an adversary $\mathcal{A}'$ such that*

$$\mathbf{Adv}^{\text{dae-priv}}_{\text{BCTR}[E]}(\mathcal{A}) \leq \frac{14m^2q^2}{2^n} + \mathbf{Adv}^{\text{prp}}_{E}(\mathcal{A}') \tag{10.6}$$

$$\mathbf{Adv}^{\text{dae-auth}}_{\text{BCTR}[E]}(\mathcal{A}) \leq 2\mathbf{Adv}^{\text{prp}}_{E}(\mathcal{A}') + \frac{1}{2^n} + \frac{18m^2q^2}{2^n} \tag{10.7}$$

where $\mathcal{A}'$ asks $O(q)$ queries and run for time $t + O(q)$ where t is the running time of $\mathcal{A}$.

Theorem 10.1 depicts the information theoretic bound, i.e., it shows the behavior of BCTR when the block cipher is replaced by a permutation selected uniformly at random from Perm(n). Theorem 10.2 shows the complexity theoretic bound, where the block cipher is a pseudorandom permutation. The transition from Theorem 10.1 to Theorem 10.2 is quite standard and we shall not show it here. The proof of Theorem 10.1 is also done using the standard game playing technique and it does not bring in any new ideas. For completeness we do provide a complete proof of Theorem 10.1, we defer it to Section 10.6.

10.5 Hardware Implementation

We claimed that our proposal BCTR is more efficient than existing TES, in this Section we will show experimental evidences of that. First we give a detailed description of the proposed architecture followed by a careful timing analysis and the experimental results.

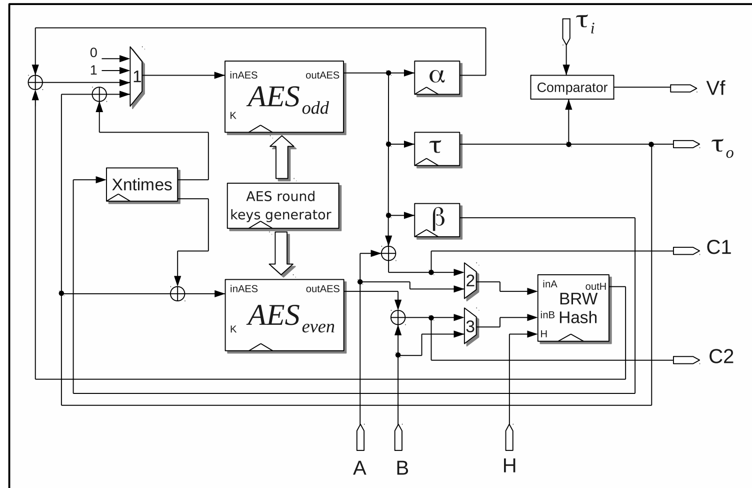


Figure 10.3: Architecture of BCTR-2

10.5.1 Proposed Architecture

A detailed study of implementations of TES were presented in Chapter 5 where the hash layers were polynomial type. BCTR uses BRW polynomial to construct a hash layer, in Chapter 6 a methodology to implement them is given taking into account the scheduling of multiplications and the storage required. For design of BCTR we consider sector sizes of 4096 bytes instead 512 bytes which was used in the previous implementations. The 4096 bytes can be seen as 256 128-bit blocks, to compute the tag (τ) in the algorithm shown in Table 10.2 all the message and tweak are used, hence the number of blocks which are hashed using the BRW polynomials is 257. We designed the architecture to compute the BRW polynomial of 256 blocks following the methodology presented in Chapter 7 and then the tweak is added and the result is multiplied by h, the powers of h are computed on the fly.

We use the same AES which was used in the implementation presented in Section 7.5. In the case of multiplier we decided to use a 4-stage-pipelined Karatsuba-Ofman multiplier, the registers were placed after a meticulous re-timing process to create balanced pipeline stages. Note that in Chapter 6 for designing the modes using BRW polynomials we used a multiplier of 3 stage. The choice was made to match the critical path of others components. In designs

of HEH[BRW] and HMCH[BRW] presented in Chapter 7 the critical path was given by AES decryption. But as this component is absent in BCTR using a 4-stage pipelined multiplier is better choice as it matches the critical path of AES encryption.

Two architectures were developed one using two AES cores which we call as BCTR-2 and another using a single AES core which we call as BCTR-1. We will only describe in detail the architecture of BCTR-2, but we will present the results of both architectures in Section 10.5.3.

The complete architecture of BCTR-2 using two AES cores is shown in Figure 10.3. The figure shows two AES cores labeled AES_{even} and AES_{odd} and the block for computing the BRW polynomials. The input lines A and B receive the plaintext and the line H receives the key h for the BRW polynomial. When used for decryption, the circuit receives the tag τ from the port labeled τ_i. If the circuit is used or encryption then the ciphertext is produced in the lines C_1, C_2 and the tag in τ_0. When used for decryption the ciphertexts are received as input from C_1, C_2 and the plaintext output is produced in P_1, P_2, In case of decryption, the port Vf is set to a zero or one depending on whether the decryption was successful or not.

We now describe the basic data flow in the circuit with reference to the algorithm in Figure 10.2. The block AES_{odd} is use to compute α, β, γ and R_j for odd values of j, while AES_{even} is used to produce R_j for even values of j. The values of α, β, γ have to be stored, for that purpose three registers are used which are labeled by the names of their contents in the figure. The block Xntimes generates the stream $x^j \cdot \beta$ for $j = 1, ..., 256$. Xntimes has two outputs, the output connected to AES_{odd} gives values computed using odd values of j and the output connected to AES_{even} gives values computed using even values of j. The inputs of BRW Hash can be from the input lines A and B for encryption or from $C1$ or C_2 for decryption, the selection of correct input is done using multiplexers 2 and 3. Only one key generator is used by both AES cores which results in some saving in area. The block BRW Hash has its own control unit composed of a ROM memory and a counter, this circuit activates a ready signal when the computation of BRW Polynomial has been completed meanwhile the control of the architecture in Figure 10.3 was constructed using a finite states machine.

10.5.2 Timing Analysis

Now we analyze the behavior of the circuit of BCTR-2 in time. In Figure 10.4 a time diagram for encryption process is shown, which clearly shows the possible parallelization. The computation of BRW polynomial and the computation of α and β can be parallelized. For computing the value γ a BRW polynomial on 256 blocks is to be computed which takes 135 cycles, further this result has to be multiplied by h which takes an additional 4 cycles (as we use a four staged pipelined multiplier), hence the total number of clocks required

for producing γ is 140 (this includes an additional cycle for synchronization). We use a pipelined AES whose latency is 11 cycles. When γ is ready it is fed to AES then after 11 clock cycles τ is produced which stored in the register $\boldsymbol{\tau}$ which is connected to the output τ_o. After τ has been generated then the generation of the stream R_j is started by the two AES cores simultaneously. The first block of the stream R_1 is ready after 11 cycles. After this, in each cycle two cipher blocks are produced. The first valid block of cipher text appear in the output after a latency of 164 clock cycles. All the ciphertext are given after 128 clock cycles hence the complete encryption of a 4096-bytes is achieved after 292 clock cycles.

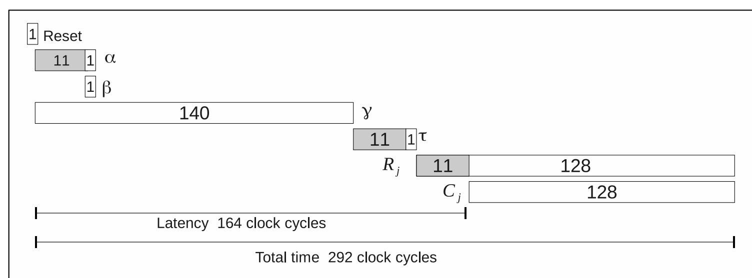


Figure 10.4: Timing diagram for BCTR-2 encryption.

In the encryption process the computation of γ cannot be parallelized with the encryption with AES, but this is not the case for decryption. This makes the decryption and verification process very different from the encryption process. The time diagram for decryption process is shown in Figure 10.5. As shown in the diagram, the circuit begins with the computation of α and β and as soon as α and β are computed then the AES cores are used to produce the plaintext blocks. As soon as two plaintext blocks are available the computation of τ can be started. The first valid block of plaintext is produced after 24 clock cycles. The total process, i.e, encryption and verification is done using only 176 clock cycles which is much less than the encryption process. This characteristic of the circuit can be very useful in the disk encryption application, as generally the number of reads from a sector is much more than the number of writes in it.

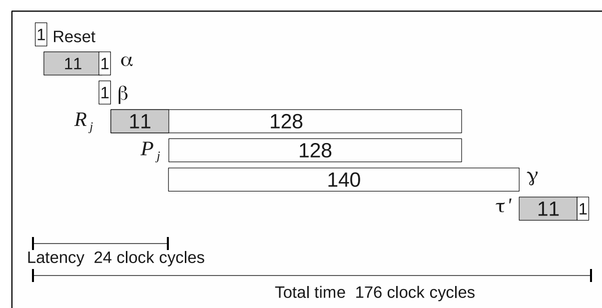


Figure 10.5: Time diagram for BCTR-2 decryption.

Table 10.4: Performance of BCTR on Virtex-5 device. AES-PEC: AES pipelined encryption core, AES-PDC: AES pipelined decryption core, AES-SDC: AES sequential decryption core, SOF : squares computed on the fly.

Mode	**Implementation Details**	**Slices**	**Frequency (MHz)**	**Throughput (Gbits/Sec)**	**TPA (Mbits/Sec)/Slice**
BCTR-2	2 AES-PEC, SOF	7876	291.29	32.69/54.23	4.15/6.88
BCTR-1	1 AES-PEC, SOF	5517	292.56	17.12/30.34	3.10/5.49
HMCH[BRW]-1 Chapter6	2 AES-PEC, 1 AES-SDC, SOF	8040	211.79	13.14	1.63
HMCH[BRW]-2 Chapter6	1 AES-PEC, 1 AES-SDC, SOF	6112	223.36	11.44	1.86
HEH[BRW]-1 Chapter6	2 AES-PEC, 2 AES-PDC, SOF	11850	202.86	15.17	1.28
HEH[BRW]-2 Chapter6	1 AES-PEC, 1 AES-PDC, SOF	8012	218.38	12.96	1.61

10.5.3 Results

We implemented both BCTR-1 and BCTR-2 in Virtex 5 xc5vlx330-2ff1760. The results reported in Table 10.4 for BCTR were obtained after place and route simulation using Xilinx ISE 10.1. In Table 10.4 we compare performance of BCTR with the implementations of HMCH[BRW] and HEH[BRW] presented in Section 7.5. The performance is measured in terms of slices, frequency, throughput and throughput per area (TPA). For BCTR-1 and BCTR-2 there are two values specified for throughput and TPA, these values are for encryption/decryption.

In HMCH[BRW]-1 and HEH[BRW]-1 two pipelined AES encryption cores were used, so they are comparable to BCTR-2. Both HMCH[BRW]-1 and HEH[BRW]-1 are larger in size than BCTR-2 as they require a AES decryption core which is not required in case of BCTR. The higher frequency achieved by both BCTR-1 and BCTR-2 is due to the use of the 4 staged pipelined multiplier, the multipliers used in case of HMCH and HEH are 3 stage pipelined. We explained before that using higher than 3 stages for HEH and HMCH does not help as the critical path for these circuits is given by the AES decryption core, thus adding more pipeline stages in the multiplier only adds on to the latency without any improvement in frequency. The frequencies achieved in both BCTR-1 and BCTR-2 are close to the frequency of the AES encryption core.

For encryption the throughput achieved by BCTR-2 is more than two times of that both HMCH[BRW]-1 and HEH[BRW]-1, and for decryption it is more than four times. The TPA metric for both BCTR-1 and BCTR-2 is also far better than the other implementations. So, it is beyond doubt that BCTR achieves better throughput and also better time/area tradeoff than any existing TES.

10.6 Deferred Proofs

For proving Theorem 10.1 we use the following lemmas.

Lemma 10.1. *Let* $X = (X_1, X_2, \dots, X_m), X' = (X'_1, X'_2, \dots, X'_m) \in [GF(2^n)]^m$ *such that* $X \neq X'$. *Let* $Y = h\mathsf{BRW}_h(X_1, X_2, \dots, X_m)$ *and* $Y' = h\mathsf{BRW}_h(X'_1, X'_2, \dots, X'_m)$. *Then for every fixed* $\delta \in GF(2^n)$

$$\Pr[Y \oplus Y' = \delta] \leq \frac{2m}{2^n},$$

The probability is taken over the random choice of h.

See [125] for a proof of this.

Lemma 10.2. *Let* $\pi \xleftarrow{\$} \mathsf{Perm}(n)$, *and* $h \xleftarrow{\$} \{0,1\}^n$. *Let* $f_{h,\pi}(.,.)$ *be a function such that given an input* $(X, T) \in \{0,1\}^{nm} \times \{0,1\}^n$, *it returns* $\pi(h\mathsf{BRW}_h(X||T) \oplus \pi(0))$. *Let* $\mathcal{B}$ *be an prf adversary attacking* $f_{h,\pi}(.,.)$ *and* $\mathcal{B}$ *asks a total of* q *queries. Then*

$$\begin{aligned} \mathbf{Adv}_f^{\mathrm{prf}}(\mathcal{B}) &= \Pr\left[\mathcal{B}^{f_{h,\pi}(.,.)} \Rightarrow 1\right] - \Pr\left[\rho \xleftarrow{\$} \mathsf{Func}(mn+n, n) : \mathcal{A}^{\rho(.)} \Rightarrow 1\right] \\ &< \frac{4mq^2}{2^n}. \end{aligned}$$

Proof. We use the game playing technique. Consider the games F0 and F1 as shown in Figure 10.6. The difference between the two games is that in F0 outputs of the permutation π is used which is constructed in the fly by the subroutine Ch-$\pi()$. In game F1, the boxed entries of Ch-$\pi()$ are removed. These games run in the same way until the bad flag is set. Hence,

$$\Pr\left[\mathcal{B}^{\mathrm{F0}} \Rightarrow 1\right] - \Pr\left[\mathcal{B}^{\mathrm{F1}} \Rightarrow 1\right] \leq \Pr[\mathcal{B}^{\mathrm{F1}} \text{ sets } \mathsf{bad}]$$

Also, the responses obtained by $\mathcal{B}$ in game F1 are elements selected uniformly at random from $\{0,1\}^n$. As $\mathcal{B}$ does not repeat a query, hence the outputs received by $\mathcal{B}$ in game F1 would be indistinguishable from those obtained from a function sampled uniformly at random from $\mathsf{Func}(mn+n, n)$. Thus,

$$\Pr\left[\mathcal{B}^{\mathrm{F1}} \Rightarrow 1\right] = \Pr\left[\rho \xleftarrow{\$} \mathsf{Func}(mn+n, n) : \mathcal{A}^{\rho(.)} \Rightarrow 1\right].$$

Hence we have

$$\mathbf{Adv}_f^{\mathrm{prf}}(\mathcal{B}) \leq \Pr[\mathcal{B}^{\mathrm{F1}} \text{ sets } \mathsf{bad}].$$

```
Subroutine Ch-π(X)
01.   Y ←$ {0,1}^n; if Y ∈ Range_π then bad ← true; [ Y ←$ ¬Range_π ]; endif;
02.   if X ∈ Domain_π then bad ← true; [ Y ← π(X) ]; endif
03.   π(X) ← Y; Domain_π ← Domain_π ∪ {X}; Range_π ← Range_π ∪ {Y}; return(Y);

Initialization:
11. for all X ∈ {0,1}^n π(X) = undefined endfor
12. EZ ←$ {0,1}^n; Domain_π ← Domain_π ∪ {0}; Range_π ← Range_π ∪ {EZ}
13. bad ← false

Respond to the s^th query (X_1^s||X_2^s||...||X_m^s, T^s) of B as follows:
31.   t^s ← BRW_h(X_1^s||X_2^s||...||X_m^s||T^s);
32.   τ^s ← Ch-π(t^s ⊕ EZ)
33.   Return (τ^s)
```

Figure 10.6: Games F0 and F1. In game F0 the subroutine Ch-π() with the boxed entries is used while in game F1 without these.

Following the construction of Ch-π(), the bad flag in game F1 is set when there is a collision in the set $Domain_\pi$ or the set $Range_\pi$. Let us list all the values that gets in the domain and range sets in the multisets $\mathcal{SS}$ and $\mathcal{RR}$. Then we would have, $\mathcal{SS} = \{0\} \cup \{t^s \oplus EZ : 1 \leq s \leq q\}$ and $\mathcal{RR} = \{EZ\} \cup \{\tau^s : 1 \leq s \leq q\}$. Let COL be the event that there is a collision in $\mathcal{SS}$ or in $\mathcal{RR}$. Then,

$$\Pr[\mathcal{B}^{\mathrm{F1}} \text{ sets } \mathsf{bad}] = \Pr[\mathsf{COL}].$$

Note that for two distinct queries u, v, we have $t^u = h\mathsf{BRW}_h(X_1^u||X_2^u||\ldots||X_m^u||T^u)$, and $t^v = h\mathsf{BRW}_h(X_1^v||X_2^v||\ldots||X_m^v||T^v)$ where $X_1^u||X_2^u||\ldots||X_m^u||T^u \neq X_1^v||X_2^v||\ldots||X_m^v||T^v$. Thus from Lemma 10.1 we have $\Pr[t^u = t^v] = 2(m+1)/2^n$. Also $\Pr[t^u \oplus EZ = 0] = 1/2^n$ as EZ is a random element of $\{0,1\}^n$, and $\Pr[\tau^u = \tau^v] = \Pr[\tau^u = EZ] = 1/2^n$, as all τ^u, τ^v and EZ are selected uniformly at random from $\{0,1\}^n$. Hence combining the above facts we get

$$\begin{aligned}\Pr[\mathsf{COL}] &= \binom{q}{2}\frac{2(m+1)}{2^n} + \frac{q}{2^n} + \binom{q+1}{2}\frac{1}{2^n} \\ &< \frac{4mq^2}{2^n}.\end{aligned}$$

Thus, we have

$$\mathbf{Adv}_f^{\mathrm{prf}}(\mathcal{B}) \leq \frac{4mq^2}{2^n}, \tag{10.8}$$

which completes the proof. □

Proof of Theorem 1: To prove the security of BCTR[Π] we replace the the block cipher E in the construction of Figure 10.2 by a permutation π chosen uniformly at random from $\Pi = \mathsf{Perm}(n)$. We call the encryption function of BCTR[Π] as $\mathcal{E}_{h,\pi}$, where $h \xleftarrow{\$} \{0,1\}^n$ is the key for the BRW polynomial. We prove the privacy bound first. We use the usual game playing technique. We briefly discuss the games below:

1. Game G0: In G0 (shown in Figure 10.7) the block-cipher is replaced by the random permutation π. The permutation π is constructed on the fly keeping record of the domain and range sets as done in the sub-routine Ch-π in Figure 10.7. Thus, G0 provide the proper encryption oracle to $\mathcal{A}$. Thus we have:

$$\Pr\left[\mathcal{A}^{\mathcal{E}_{h,\pi}(.,.,.)} \Rightarrow 1\right] = \Pr[\mathcal{A}^{\mathrm{G0}} \Rightarrow 1]. \tag{10.9}$$

2. Game G1: Figure 10.7 with the boxed entries removed represents the game G1. In G1 it is not guaranteed that Ch-π behaves like a permutation, but the games G0 and G1 are identical until the bad flag is set. Thus we have

$$\Pr[\mathcal{A}^{\mathrm{G0}} \Rightarrow 1] - \Pr[\mathcal{A}^{\mathrm{G1}} \Rightarrow 1] \leq \Pr[\mathcal{A}^{\mathrm{G1}} \text{ sets } \mathsf{bad}] \tag{10.10}$$

Note that in G1 the adversary gets random strings as output irrespective of his queries. Hence,

$$\Pr[\mathcal{A}^{\mathrm{G1}} \Rightarrow 1] = \Pr[\mathcal{A}^{\$(.,.,.)} \Rightarrow 1] \tag{10.11}$$

3. Game G2: In Game G2 (shown in Figure 10.8) we do not use the subroutine Ch-π any more but return random strings immediately after the $\mathcal{A}$ asks a query. Later we keep track of the elements that would have got in the domain and range sets of the permutation π in multi-sets $\mathcal{S}$ and $\mathcal{R}$. We set the bad flag when there is a collision in either $\mathcal{S}$ or $\mathcal{R}$. For the adversary the games G1 and G2 are identical. So,

$$\Pr[\mathcal{A}^{\mathrm{G1}} \Rightarrow 1] = \Pr[\mathcal{A}^{\mathrm{G2}} \Rightarrow 1] \tag{10.12}$$

and

$$\Pr[\mathcal{A}^{\mathrm{G1}} \text{ sets } \mathsf{bad}] = \Pr[\mathcal{A}^{\mathrm{G2}} \text{ sets } \mathsf{bad}] \tag{10.13}$$

Hence, using Eqs. (10.9), (10.10), (10.11), (10.12) and (10.13), we have

$$\Pr\left[\mathcal{A}^{\mathcal{E}_{h,\pi}(.,.,.)} \Rightarrow 1\right] - \Pr[A^{\$(.,.,.)} \Rightarrow 1] \leq \Pr[\mathcal{A}^{\mathrm{G2}} \text{ sets } \mathsf{bad}].$$

```
    Subroutine Ch-π(X)
01.    Y ←$ {0,1}^n; if Y ∈ Range_π then bad ← true; [ Y ←$ \overline{Range_π} ]; endif;
02.    if X ∈ Domain_π then bad ← true; [ Y ← π(X) ]; endif
03.    π(X) ← Y; Domain_π ← Domain_π ∪ {X}; Range_π ← Range_π ∪ {Y}; return(Y);

    Initialization:
11.    for all X ∈ {0,1}^n π(X) = undefined endfor
12.    EZ ←$ {0,1}^n; π(0) = EZ;
13.    Domain_π ← {0}; Range_π ← {EZ};
14.    EO ←$ {0,1}^n \ {EZ}; π(1) = EO;
15.    Domain_π ← Domain_π ∪ {1}; Range_π ← Range_π ∪ {EO};
16.    bad = false

Respond to the s^th encryption query (T^s; P_1^s, P_2^s, ..., P_m^s) as follows:
101. γ^s ← h · BRW(P_1^s||P_2^s||...||P_m^s||T^s);
102. τ^s ← Ch-π(γ^s ⊕ EZ);
103. for i = 1 to m,
104.   R_i^s ← Ch-π(τ^s ⊕ x^i EO);
105.   C_i^s ← R_i^s ⊕ P_i^s;
106. endfor
107. Return (C_1^s||C_2^s||...||C_{m^s}^s||τ^s)
```

Figure 10.7: Games G0 and G1

According to the definition of the privacy advantage of $\mathcal{A}$, we have

$$\mathbf{Adv}^{\text{dae-priv}}_{\text{BCTR}[\Pi]}(\mathcal{A}) \leq \Pr[\mathcal{A}^{\text{G2}} \text{ sets } \mathsf{bad}] \tag{10.14}$$

Now we need to bound $\Pr[\mathcal{A}^{\text{G2}} \text{ sets } \mathsf{bad}]$. The elements in the multi-sets $\mathcal{S}$ and $\mathcal{R}$ would be

$$\mathcal{S} = \{0,1\} \cup \{\gamma^s \oplus EZ : 1 \leq s \leq q\} \cup \{\tau^s \oplus x^i EO : 1 \leq i \leq m, 1 \leq s \leq q\} \tag{10.15}$$

$$\mathcal{R} = \{EZ, EO\} \cup \{\tau^s : 1 \leq s \leq q\} \cup \{C_i^s \oplus P_i^s : 1 \leq i \leq m, 1 \leq s \leq q\}, \tag{10.16}$$

Let COLLD be the event that there is a collision in $\mathcal{S}$ and COLLR be the event that there is a collision in $\mathcal{R}$ Using the facts that $\gamma^s = h\mathsf{BRW}_h(P_1^s||P_2^s||\ldots||P_m^s||T^s)$ and EZ, EO, τ^s, C_i^s are random elements of $\{0,1\}^n$, we have

$$
\begin{aligned}
\Pr[\mathsf{COLLD}] &\leq \frac{2q}{2^n} + \frac{2mq}{2^n} + \binom{q}{2}\frac{(2m+2)}{2^n} + \binom{mq}{2}\frac{1}{2^n} + \frac{2mq^2(m+1)}{2^n} \\
&= \frac{1}{2^n}\left(\frac{5}{2}m^2q^2 + 3mq^2 + \frac{mq}{2} + q + q^2\right)
\end{aligned}
$$

and

$$
\begin{aligned}
\Pr[\mathsf{COLLR}] &= \binom{(mq+q+2)}{2}\frac{1}{2^n} \\
&= \frac{(m^2q^2 + 2mq^2 + 3mq + q^2 + 3q + 2)}{2^{n+1}}
\end{aligned}
$$

Then we have

$$
\begin{aligned}
\Pr[\mathcal{A}^{\mathrm{G2}} \text{ sets } \mathsf{bad}] &= \Pr[\mathsf{COLLD}] + \Pr[\mathsf{COLLR}] \\
&< \frac{14m^2q^2}{2^n} \qquad (10.17)
\end{aligned}
$$

This completes the proof of the privacy bound. □

Proving the authentication bound: Before proving the authentication bound we will introduce a necessary concept of forgery of psudorandom functions.

For pseudorandom functions another adversarial goal beside indistinguishability from random functions is known as forgery. For defining forgery for a pseudorandom function $F : \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$, an adversary $\mathcal{A}$ is given oracle access to $F_K(.)$, where $K \stackrel{\$}{\leftarrow} \mathcal{K}$. Suppose $\mathcal{A}$ makes q queries X_i, $1 \leq i \leq q$ to its oracle and gets back $Y_i = F_K(X_i)$, $1 \leq i \leq q$. At the end $\mathcal{A}$ attempts a forgery by producing a pair $(\tilde{X}, t)$ such that $\mathcal{A}$ never obtained t as a response to any of his previous queries X_i $(1 \leq i \leq q)$. $\mathcal{A}$ is said to be successful if $F_K(\tilde{X}) = t$. It is well known that (for example see [126])

$$
\Pr[\mathcal{A}^{F_K(.)} \text{ forges}] \leq \mathbf{Adv}_F^{\mathrm{prf}}(\mathcal{A}) + \frac{1}{|\mathcal{Y}|} \qquad (10.18)
$$

To prove the authenticity bound, let $\mathcal{A}$ be an adversary which tries to break authenticity of BCTR[Π]. Let $\mathcal{B}$ be an adversary attacking authenticity of $f_{h,\pi}$ (see Lemma 10.2 for the description of $f_{h,\pi}$). $\mathcal{B}$ has an oracle access to $f_{h,\pi}(.,.)$, additionally let it have access

Initialization:

$\mathcal{S} \leftarrow \mathcal{R} \leftarrow \emptyset$;
$EZ \xleftarrow{\$} \{0,1\}^n$; $EO \xleftarrow{\$} \{0,1\}^n \setminus \{EZ\}$;
$\mathcal{S} \leftarrow \mathcal{S} \cup \{0,1\}$; $\mathcal{R} \leftarrow \mathcal{R} \cup \{EZ, EO\}$;

For an encryption query $(T^s; P_1^s, P_2^s, \ldots, P_{m^s}^s)$ respond as follows:

$C_1^s||C_2^s||\ldots||C_{m^s}^s||\tau^s \xleftarrow{\$} \{0,1\}^{(m+1)n}$;
Return $C_1^s||C_2^s||\ldots||C_{m^s}^s||\tau^s$;

Finalization:

First Phase
for $s = 1$ to q,
$\gamma^s \leftarrow h \cdot BRW(P_1^s||P_2^s||\ldots||P_{m^s}^s||T^s)$;
$\mathcal{S} \leftarrow \mathcal{S} \cup \{\gamma^s\}$; $\mathcal{R} \leftarrow \mathcal{R} \cup \{\tau^s\}$;
for $i = 1$ to m,
$\mathcal{S} \leftarrow \mathcal{S} \cup \{\tau^s \oplus x^i EO\}$; $\mathcal{R} \leftarrow \mathcal{R} \cup \{C_i^s \oplus P_i^s\}$
end for
end for

Second phase
bad = false;
if (some value occurs more than once in $\mathcal{S}$) **then** bad = true **endif**;
if (some value occurs more than once in $\mathcal{R}$) **then** bad = true **endif**.

Figure 10.8: Game G2: $\mathcal{S}$ and $\mathcal{R}$ are multisets.

to another oracle $\mathcal{P}(.,.)$ which on input (X, i) returns $\pi(X \oplus x^i\pi(1))$. With access to these two oracles $\mathcal{B}$ can run $\mathcal{A}$ by answering its queries in the usual manner. When $\mathcal{A}$ outputs a forgery $(T, Y_1||Y_2||\ldots||Y_m||\tau)$, $\mathcal{B}$ computes $X_i = \mathcal{P}(\tau, i)$, $1 \leq i \leq m$, using the oracle $\mathcal{P}(.,.)$, and outputs $(X_1||X_2||\ldots||X_m||T, \tau)$ as its forgery. $\mathcal{B}$ is always successful if $\mathcal{A}$ is successful, hence we have

$$\begin{aligned}\mathbf{Adv}^{\text{dae-auth}}_{\text{BCTR}[\Pi]}(\mathcal{A}) &= \Pr[\mathcal{A}^{\mathcal{E}_{h,\pi}(.,.,.)} \text{ forges}] \\ &\leq \Pr[\mathcal{B}^{f_{h,\pi}(.),\mathcal{P}(.,.)} \text{ forges}] \qquad (10.19)\end{aligned}$$

Now, we replace this oracle $\mathcal{P}(.,.)$ with an oracle $\$(.,.)$ which returns strings chosen uniformly at random from $\{0,1\}^n$ on a query (X, i). The difference of the real oracle $\mathcal{P}$ and the oracle $\$(.,.)$ can be detected by $\mathcal{B}$ only if the queries of $\mathcal{B}$ can produce a collision in the domain or range of the permutation π. This means that the difference can be detected by $\mathcal{B}$ if there is a collision in the multisets $\mathcal{S}$ or $\mathcal{R}$ as represented in equations (10.15) and (10.16) respectively.

The event of a collision in these sets were represented by COLLD and COLLR respectively. Thus we have

$$\Pr[\mathcal{B}^{f_{h,\pi}(.),\mathcal{P}(.,.)} \text{ forges}] - \Pr[\mathcal{B}^{f_{h,\pi}(.),\$(.,.)} \text{ forges}] \leq \Pr[\mathsf{COLLD}] + \Pr[\mathsf{COLLR}]. \tag{10.20}$$

Now, the oracle $\$(.,.)$ is of no help to $\mathcal{B}$ as the random strings that it returns can be generated by $\mathcal{B}$ itself hence,

$$\Pr[\mathcal{B}^{f_{h,\pi}(.),\$(.,.)} \text{ forges}] = \Pr[\mathcal{B}^{f_{h,\pi}(.)} \text{ forges}]. \tag{10.21}$$

Putting together equations (10.20), (10.21) and (10.17) we have

$$\begin{aligned}\Pr[\mathcal{B}^{f_{h,\pi}(.),\mathcal{P}(.,.)} \text{ forges}] &\leq \Pr[\mathcal{B}^{f_{h,\pi}(.)} \text{ forges}] + \frac{14m^2q^2}{2^n} \\ &< \frac{1}{2^n} + \frac{4mq^2}{2^n} + \frac{14m^2q^2}{2^n}. \end{aligned} \tag{10.22}$$

The last inequality follows from Lemma 10.2 and eq. (10.18). From eq. (10.19) and eq. (10.22) we have

$$\mathbf{Adv}^{\text{dae-auth}}_{\text{BCTR}[\Pi]}(\mathcal{A}) \leq \frac{1}{2^n} + \frac{18m^2q^2}{2^n},$$

as desired. □

10.7 Final Remarks

In this Chapter we explored the possibility of the use of DAEs for disk encryption and argued that their use can lead to more efficient solutions to the disk encryption problem. The experimental results provided in this Chapter clearly shows that BCTR can be a option for disk encryption if length expansion in the cipher text is allowed.

Chapter 11

A Proposal for Secure Backup

Freedom is indivisible. The chains on any one of my people were the chains on all of them, the chains on all of my people were the chains on me.

Nelson Mandela

We have reached the last technical Chapter of this thesis. In this Chapter we shall discuss another problem related to security of stored data which is quite different from the problems that we have addressed so far. In the introduction of this thesis we tried to motivate the problem of securing stored data by the example of a lost laptop, we mentioned that the laptop loss rates are around 2% per year [54]. A lost laptop amounts to two severe consequences: (1) the loss of the data stored in it (2) the stored data getting revealed to an adversary. The later problem has been identified to be more severe than the former one, as it is rightly argued that if sensitive data gets into the hands of an unwanted person he/she can potentially cause much greater damage than the one incurred by the mere loss of the data.

The preceding Chapters of this thesis have been mostly devoted to various aspects of problem (2). Problem (1) has not been adequately addressed in the literature. A trivial solution of problem (1) is to maintain backups of the data securely. With the advent of cheap storages and backup technologies which allows synchronous read/write operations even from remote storage devices, keeping backup is easy. So one may think that the data always gets written to two storage devices both equipped with a low level disk encryption algorithm (as provided by tweakable enciphering schemes), thus loss of one device does not result in any of the consequences stated before. In this Chapter we handle the problem of backup but in a bit different way than the trivial solution. In the trivial solution, as stated, each plain text will have associated with it two cipher texts (possibly different as one can use two different algorithms or two different keys for encryption in the two different devices), and in a normal scenario both these ciphertexts would be available to the user. We suggest a scheme which can exploit this redundancy and thus give rise to more efficient ways to solve this problem.

Our goal in this Chapter is to design a symmetric key encryption scheme which produces two ciphertexts C^{L} and C^{R} (L and R can be read as local and remote) and a tag τ for a given plaintext P with the action of one or more secret keys. The requirements on C^{L} and C^{R} would be the following:

1. C^{L}, C^{R} and P must be of same length.

2. There should exist a very efficient function g such that $g(C^{\mathsf{L}}, C^{\mathsf{R}}) = P$ and g should not require any secret parameter.

3. One must be able to recover P from either C^{L} or C^{R} by using the secret key and the tag.

The rationale behind such a goal is as follows. For the secure backup scenario, we assume plaintexts to be disk sectors and the corresponding ciphers would also get written in disk sectors, hence the two ciphertexts must be of the same length of that of the plaintext. When a user has acces to both C^{L} and C^{R} then (s)he can very efficiently produce the plain text which produced them using the function g. If one of C^{L} or C^{R} is lost then with the knowledge of the secret key and the tag, P can be recovered from one of C^{L} or C^{R}. The function g should be such that obtaining P through it should be much more efficient than obtaining P by decryption of either C^{L} or C^{R}.

Additionally, we require the scheme to be secure in some sense. For arguing about security, given plaintext P, we rule out the possibility of adversarial access to both C^{L} and C^{R}. This is not very un-natural as we may assume that C^{L} is stored in the laptop of an user where as C^{R} gets stored in a trusted location provided by his/her employer. It may be difficult for an adversary to have access to both versions of the ciphertext. The security goal for the scheme is that, it should be difficult for any computationally bounded adversary to distinguish between the outputs of the encryption scheme from random strings when the adversary can see either C^{L} or C^{R} along with the tag for messages of his choice. Additionally by looking at either C^{L} or C^{R} along with the tag for messages of his/her choice, (s)he should be unable to produce a new ciphertext tag pair which would get decrypted.

We analyze the above stated problem and propose a solution to it. Our solution is a special encryption algorithm which we call the *double ciphertext mode* (DCM). We describe the syntax of a DCM and define security for it. We provide a generic construction for a DCM using a pseudorandom function and a block-cipher, and also a specific construction which requires only one block-cipher key. We prove security for both the constructions. The material presented in this Chapter have been previously published in [26].

11.1 The Double Ciphertext Mode

We fix m, n, ℓ as positive integers. Let, $\mathcal{M} = \{0,1\}^{nm}$, $\mathcal{C} = \{0,1\}^{nm+\ell}$, $\mathcal{T} = \{0,1\}^n$ be the message space and cipher space and tweak space respectively. A double ciphertext mode (DCM) is a set of four algorithms $(\mathcal{K}, \mathcal{E}, \mathcal{D}, g)$, where $\mathcal{K}$ is the key generation algorithm (we shall also denote the key space with $\mathcal{K}$), $\mathcal{E}$ the encryption algorithm, $\mathcal{D}$ the decryption algorithm, and g a special function which we call the *recovery function*.

The encryption function $\mathcal{E} : \mathcal{K} \times \mathcal{T} \times \mathcal{M} \times \{\mathsf{R}, \mathsf{L}\} \to \mathcal{C}$ takes in a key, a tweak, a message and type (which can be either R or L)to produce a cipher. For any given $M \in \mathcal{M}$, $K \in \mathcal{K}$, $T \in \mathcal{T}$, and $\mathsf{ty} \in \{\mathsf{R}, \mathsf{L}\}$ we shall write $\mathcal{E}(K, T, M, \mathsf{ty})$ as $\mathcal{E}_K(T, M, \mathsf{ty})$ or sometimes as $\mathcal{E}_K^{T,\mathsf{ty}}(M)$. Any cipher $\mathcal{C}$ produced by the encryption function from a message M can be parsed as $\mathcal{C} = C||\tau$, where $|C| = |M|$ and $|\tau| = \ell$. We define functions cipher and tag, such that given $\mathcal{C} = C||\tau \in \mathcal{C}$, $\mathsf{cipher}(\mathcal{C}) = C$ and $\mathsf{tag}(\mathcal{C}) = \tau$. Also, for any $M \in \mathcal{M}$, $K \in \mathcal{K}$, $T \in \mathcal{T}$, $\mathsf{tag}(\mathcal{E}_K^{T,\mathsf{L}}(M)) = \mathsf{tag}(\mathcal{E}_K^{T,\mathsf{R}}(M))$.

$\mathcal{D} : \mathcal{K} \times \mathcal{T} \times \mathcal{C} \times \{R, L\} \to \mathcal{M} \cup \{\perp\}$ is the decryption algorithm, $\mathcal{D}(K, T, \mathcal{C}, \mathsf{ty})$ is denoted as $\mathcal{D}_K(T, \mathcal{C}, \mathsf{ty})$ or sometimes as $\mathcal{D}_K^{T,\mathsf{ty}}(M)$. And,

$$\begin{aligned} \mathcal{D}_K(T, \mathcal{C}, \mathsf{ty}) &= M, \text{ if } \mathcal{C} = \mathcal{E}_K(T, M, \mathsf{ty}), \\ &= \perp, \text{ otherwise.} \end{aligned}$$

The function g is a public function which does not use any key. Let $\mathcal{C}^{\mathsf{L}} = \mathcal{E}_K(T, M, \mathsf{L})$ and $\mathcal{C}^{\mathsf{R}} = \mathcal{E}_K(T, M, \mathsf{R})$, then $g(\mathsf{cipher}(\mathcal{C}^{\mathsf{L}}), \mathsf{cipher}(\mathcal{C}^{\mathsf{R}})) = M$. The existence of this special recovery function g makes a DCM scheme different from other encryption schemes. This function g enables decryption of a ciphertext given the two versions of it (i.e., both the L and R versions) without the key. For a DCM to be practically useful, it is required that the recovery function g can be computed much more efficiently than the decryption function $\mathcal{D}$.

We define DCM with fixed values of m, n and ℓ, where n can be seen as the block-length of the block-cipher. Thus, the message length is fixed to some multiple of n and the tweak length is always n. We define in this manner keeping in mind the specific application, where a more general definition is not required. But a DCM can be defined more generally where the message length, tweak length and tag length are not fixed.

11.1.1 Secure Backup Through DCM

The above syntax of a DCM has been constructed keeping in mind the specific need of a secure backup scheme. In what follows we try to describe how a DCM can be used for keeping secure backup using an example. This example is meant to be motivational, we do

not try to detail technical concerns for a true implementation of a DCM.

A user (U) works with sensitive data in his office which (s)he stores in his laptop. U cannot afford to lose the data or to reveal the data to an adversary. The employer of U has a server (S), where U can keep backups. Let us consider that U mirrors the disk of his laptop with that of the server S[1]. The disk controllers of the laptop and the server are both equipped with a DCM instantiated with the same keys. The type ty is set to L and R in the laptop and the server respectively, i.e., in the laptop the encryption algorithm $\mathcal{E}_K(.,.,\mathsf{L})$ runs and in the server the encryption algorithm $\mathcal{E}_K(.,.,\mathsf{R})$ runs. When a plaintext M is to be written in the sector with address T, then $\mathsf{cipher}(\mathcal{E}_K(T, M, \mathsf{L}))$ and $\mathsf{cipher}(\mathcal{E}_K(T, M, \mathsf{R}))$ gets written in the sector with address T of the laptop and the server respectively. The tag produced by the encryption algorithms in both the server and the laptop would be the same and one of it have to be stored. Without loss of generality we consider that in the server, the $\mathsf{tag}(\mathcal{E}_K(T, M, \mathsf{R}))$ is communicated to the operating system which stores it in another disk indexed with the address T. Note that the tag produced for the ciphertext stored in address T is not stored in the sector, but it is stored in a different location. The operating system maintains the tags with the related disk sector addresses through a software solution. Note, that the tags can be public as this is a part of the ciphertext and need not be stored securely.

Now let us discuss about decryption. In the normal scenario, when nothing unnatural has happened to the laptop of U, then to recover the contents of sector T, U requests a read from both the disks of his laptop and the server to get $\mathsf{cipher}(\mathcal{E}_K(T, M, \mathsf{L}))$ and $\mathsf{cipher}(\mathcal{E}_K(T, M, \mathsf{R}))$, and recovers M by the recovery function g. In an un-natural scenario when data in one of the disks is lost then U applies the decryption function on the disk sector along with the tag to get back the plaintext.

When a plaintext is recovered using the function g then the tag is not used, so the integrity of the message is not checked. But U can check for the integrity of the messages in the disk sectors at any point (s)he wants. He can schedule regular integrity checks similar to virus scans even if (s)he believes that the scenario is normal. But, this functionality does not amount to the type of authentication which is provided by schemes like authenticated encryption as the authenticity check in case of DCM can only be done in an off-line manner. But this should not be seen as an weakness for DCM, as in other proposals for sector wise disk encryption like in tweakable enciphering schemes also true authentication cannot be obtained. A tweakable enciphering scheme provides authenticity in the sense that if the ciphertext is tampered then upon decryption the resulting ciphertext is random. This does not amount to true authentication.

The security of a DCM (which is discussed in details in the next section) guarantees that an adversary who does not have access to both versions of the ciphertext cannot forge a

[1]The laptop server analogy is purely motivational, but two physically identical disks can be mirrored so that the sector addresses of them match and same data gets written and read simultaneously from the same addresses of the two disks. This can be physically achieved for other devices also.

ciphertext. This implies that if the server has access to only the cipher stored in it then U need not trust the server in which his backup is being kept in the sense that a forgery created by the server can be easily detected by U.

An important parameter of the application would be the communication cost between the server and the laptop. To gain from the feature of avoiding decryption in the normal scenario the communication cost should be low. The communication cost would depend on many factors including the remoteness of the server from the laptop, the communication technology being used etc.

11.1.2 Security of DCM

We want to make DCM secure in two ways, in terms of privacy and authentication. Firstly, as stated earlier, we assume that an adversary attacking DCM should have access to only one of the versions of the ciphertext, i.e., either the L or the R version. We give the adversary the liberty to see one version (either L or R) of the ciphertext along with the tag for plaintexts of his choice and we say that the adversary breaks DCM in the sense of privacy if the adversary can distinguish between the obtained ciphertexts and random strings. This is the usual notion of privacy as applicable to symmetric key encryption schemes but here the adversary is restricted in the sense that it can only view one version of the ciphertext. The other form of security that we want is that of authentication, that is an adversary must not be able to modify ciphertexts such that the modified ones gets decrypted, i.e., it should be difficult for an adversary to create a new version of ciphertext and tag pair which will get decrypted even if (s)he has seen ciphertexts tag pairs of plaintexts of his choice. We formalize these notions of security next.

Let $\Psi = (\mathcal{K}, \mathcal{E}, \mathcal{D}, g)$ be a DCM scheme. Let $\mathcal{A}$ be an adversary attacking Ψ. The adversary has access to the oracle $\mathcal{O}$. $\mathcal{O}$ can either be $\mathcal{E}_K(.,.,.)$, where K is generated uniformly at random from $\mathcal{K}$ or it can be $\$(.,.,.)$ which returns random strings of length equal to the length of the ciphertext. The adversary can query the oracle with a query of the form (T, M, ty), where M, T denotes the message and tweak respectively and $\mathsf{ty} \in \{\mathsf{L}, \mathsf{R}\}$, and thus get back the responses from the oracle. $\mathcal{A}$ has the following query restrictions:

P1. $\mathcal{A}$ cannot query the encryption oracle with the same message and tweak on both values of ty, i.e., $\mathcal{A}$ is not allowed to see both the L and R versions of the ciphertext, for the same message tweak pair.

P2. The adversary does not repeat a query.

We define the privacy advantage of adversary $\mathcal{A}$ as follows:

$$\mathbf{Adv}_{\Psi}^{\text{dcm-priv}}(\mathcal{A}) \quad = \quad \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\mathcal{E}_K(.,.,.)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\$(.,.,.)} \Rightarrow 1\right]$$

To define the authentication advantage we consider that $\mathcal{A}$ is given access to the encryption oracle $\mathcal{E}_K(.,.,.)$ and queries it with plaintexts of his choice and obtains the corresponding ciphertexts. Finally, $\mathcal{A}$ outputs a forgery, which consists of a tweak, a ciphertext and a type $(T, \mathcal{C}, \mathsf{ty})$. $\mathcal{A}$ is said to be successful if $\mathcal{D}_K(T, \mathcal{C}, \mathsf{ty}) \neq \bot$. For querying the encryption oracle $\mathcal{A}$ follows the same restrictions as described above additionally $\mathcal{A}$ have the following restrictions:

A1. $\mathcal{A}$ cannot output $(T, \mathcal{C}, \mathsf{ty})$ as a forgery if (s)he obtained $\mathcal{C}$ as a response for his query (T, X, ty) to the encryption oracle.

A2. $\mathcal{A}$ cannot output $(T, \mathcal{C}, \mathsf{ty})$ as a forgery if $g(\mathsf{cipher}(\mathcal{C}), \mathsf{cipher}(\tilde{\mathcal{C}})) = X$ where $\tilde{\mathcal{C}}$ was obtained as a response to his encryption query $(T, X, \bar{\mathsf{ty}})$.

Both restrictions are to rule out trivial wins. The second restriction is particularly interesting, note that if $\tilde{C}||\tau = \mathcal{E}_K(T, X, \mathsf{R})$ then it may be easy for an adversary to guess (or compute) C, where $C||\tau = \mathcal{E}_K(T, X, \mathsf{L})$ without querying the encryption oracle, as g is a public function and $g(C, \tilde{C}) = X$. Thus $\mathcal{A}$ may trivially produce the forgery $(T, C||\tau, \mathsf{L})$ knowing that $\tilde{C}||\tau = \mathcal{E}_K(T, X, \mathsf{R})$ and $g(C, \tilde{C}) = X$. This restriction does not lead to a practical attack, this is discussed in details in Section 11.1.3.

The authentication advantage of the adversary $\mathcal{A}$ is defined as the probability that $\mathcal{A}$ does a successful forgery, in other words

$$\mathbf{Adv}_{\Psi}^{\text{dcm-auth}}(\mathcal{A}) = \Pr[\mathcal{A}^{\mathcal{E}_K(.,.,.)} \text{ forges }]$$

We consider Ψ to be secure, if both $\mathbf{Adv}_{\Psi}^{\text{dcm-priv}}(\mathcal{A})$ and $\mathbf{Adv}_{\Psi}^{\text{dcm-auth}}(\mathcal{A})$ are small for every computationally bounded adversary $\mathcal{A}$.

11.1.3 Discussions on the Adversarial Restrictions

The security definition of DCM resembles the standard definition of security for privacy and authentication with the important difference of the adversarial restrictions. In particular the restriction P1 in case of privacy and A2 in case of authentication are peculiar to a DCM. Here we argue that these restrictions does not lead to a practical attack.

The adversarial restrictions arise from the fact that in a practical scenario we rule out the possibility that an adversary can get access to both the local and remote storage at the same time. For privacy this means that (s)he gets to see the cipher texts of only of L type or of R type, but not both. In case of authentication, this means that seeing some ciphertext tag pairs of the same type the adversary tries to forge a ciphertext tag pair of the same type. For example, if the adversary has access to type L storage then (s)he can see the ciphertexts

of the L type and ultimately produce a forgery consisting of a ciphertext of the L type and a tag. In this case, for the adversary there is no point in producing a forgery with ciphertext of R type as the adversary does not have access to that storage and thus would not be able to store his/her forged ciphertext. To translate this restriction to the adversary we could have made the adversary select and declare a type from $\{\mathsf{L}, \mathsf{R}\}$ before (s)he begins the queries, and stick to that for the whole session of queries and responses. More precisely the restrictions for privacy and authentication could have been as follows:

For privacy:

1. The adversary chooses $\mathsf{ty} \in \{\mathsf{L}, \mathsf{R}\}$, before (s)he begins the queries.
2. All queries of the adversary would be of type (T, M, ty), where ty is chosen in step 1.
3. The adversary cannot repeat a query.

For authentication:

1. The adversary chooses $\mathsf{ty} \in \{\mathsf{L}, \mathsf{R}\}$ before (s)he begins the queries.
2. All queries of the adversary would be of type (T, M, ty), where ty is chosen in step 1.
3. $\mathcal{A}$ cannot output $(T, \mathcal{C}, \mathsf{ty})$ as a forgery if (s)he obtained $\mathcal{C}$ as a response for his query (T, X, ty) to the encryption oracle.
4. $\mathcal{A}$ cannot output $(T, \mathcal{C}, \bar{\mathsf{ty}})$ as a forgery.

The above stated restrictions are most natural for a practical scenario. The above restrictions implies the restrictions that we impose on the adversary, but our restrictions are weaker. In case of privacy we only restrict the adversary to make both the queries (T, X, ty) and $(T, X, \bar{\mathsf{ty}})$ and in case of authentication we restrict a forgery attempt of $(T, \mathcal{C}, \mathsf{ty})$ if $g(\mathsf{cipher}(\mathcal{C}), \mathsf{cipher}(\tilde{\mathcal{C}})) = X$ where $\tilde{\mathcal{C}}$ was obtained as a response to the encryption query $(T, X, \bar{\mathsf{ty}})$. Note, both these cases involves mixing of the types in the queries and responses. Thus, with the assumption that the both versions of the storage are in accessible to the adversary, the restrictions that we impose will not lead to any practical attack.

11.2 DCMG: A generic construction of DCM

We construct an DCM scheme using two basic primitives a pseudo-random function and a block-cipher secure in the sense of a pseudo-random permutation. We call our construction as DCMG.

Algorithm DCMG.$\mathcal{E}^{T,\mathsf{ty}}_{K_1,K_2}(P_1||\ldots||P_m)$

1. $\tau \leftarrow F_{K_1}(P_1||P_2||\ldots||P_m||T)$;
2. **for** $j = 1$ to m ;
3. $\quad R_j \leftarrow E_{K_2}(\tau \oplus \mathsf{bin}_n(j))$;
4. $\quad$ **if** $\mathsf{ty} = \mathsf{L}$
5. $\qquad C_j \leftarrow R_j \oplus (1 \oplus x)P_j$
6. $\quad$ **else** $C_j \leftarrow R_j \oplus xP_j$
7. $\quad$ **endif**
8. **endfor**
9. return $(C_1||C_2||\cdots||C_m||\tau)$;

Algorithm DCMG.$\mathcal{D}^{T,\mathsf{ty}}_{K_1,K_2}(C_1||\ldots||C_m||\tau)$

1. **for** $j = 1$ to m ;
2. $\quad R_j \leftarrow E_{K_2}(\tau \oplus \mathsf{bin}_n(j))$;
3. $\quad$ **if** $\mathsf{ty} = \mathsf{L}$
4. $\qquad P_j \leftarrow (C_j \oplus R_j)(1 \oplus x)^{-1}$
5. $\quad$ **else** $P_j \leftarrow (C_j \oplus R_i)x^{-1}$
6. $\quad$ **endif**
7. **endfor**
8. $\tau' \leftarrow F_{K_1}(P_1||P_2||\ldots||P_m||T)$;
9. **if** $\tau' = \tau$ **return** $(P_1||\cdots||P_m)$ else **return** $\perp$;

Figure 11.1: Encryption and decryption using DCMG$[F, E]$. K_1 is the key for the PRF and K_2 the block-cipher key. $\mathsf{len}(P_i) = \mathsf{len}(C_i) = n$, $1 \leq i \leq m$.

Let $F : \mathcal{K}_1 \times \{0,1\}^{mn+n} \rightarrow \{0,1\}^n$ be a pseudo-random function and $E : \mathcal{K}_2 \times \{0,1\}^n \rightarrow \{0,1\}^n$ be a blockcipher secure in the sense of a pseudo-random permutation. We construct a DCM scheme using F and E which we call as DCMG$[F, E]$. The message space of DCMG is $\{0,1\}^{mn}$ and the tweak space is $\{0,1\}^n$. We describe the encryption and decryption algorithms using DCMG in Figure 11.1. A schematic diagram of DCMG is shown in Figure 11.2.

The algorithms in Figure 11.1 are self explanatory. Using the pseudo-random function F a tag τ is produced which acts as the initialization vector of a counter type mode of operation. The counter mode produces blocks of pseudo-random strings, which are mixed with the plaintext. For producing ciphertexts of two different types the output of the counter mode are mixed in two different way. The specific way in which this mixing is done is to enable the existence of an efficient function g. In this case the function g is simple, we define g as $g(X, Y) = X \oplus Y$. Let $C^{\mathsf{L}}||\tau = \mathcal{E}^{T,\mathsf{L}}_{K_1,K_2}(P_1, \ldots, P_m)$ and $C^{\mathsf{R}}||\tau = \mathcal{E}^{T,\mathsf{R}}_{K_1,K_2}(P_1, \ldots, P_m)$, It is easy to verify that $C^L \oplus C^R = P_1||\ldots||P_m$.

11.2.1 Characteristics of the construction

1. **Efficiency:** The construction requires a pseudo-random function (PRF) which can be replaced by a secure keyed message authentication code. Other than the call to the PRF, the scheme requires m block-cipher calls to encrypt a m block message.

   Decryption is costlier than encryption as decryption requires all the operations necessary for encryption plus a multiplication by x^{-1} or $(1 \oplus x)^{-1}$ per block of message depending on whether we are decrypting the R version or the L version. For decrypting

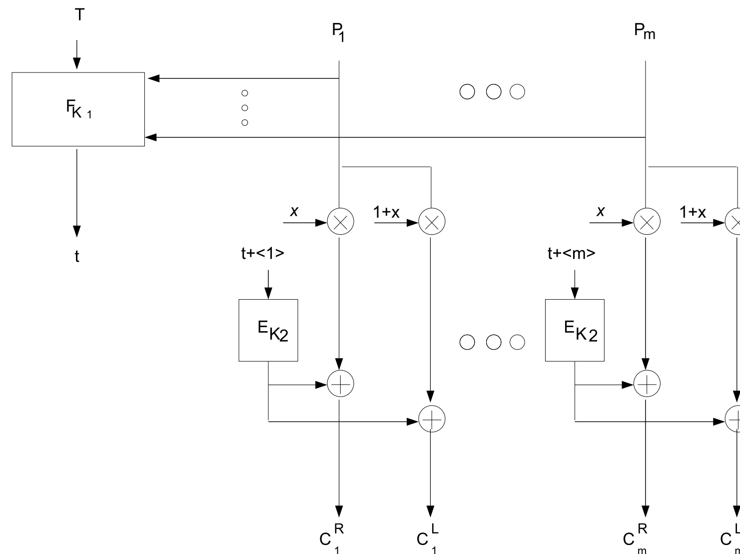


Figure 11.2: A schematic diagram of DCMG. In this diagram the tag produced by the pseudorandom function F is denoted by t, and $< m >$ denotes the n bit binary representation of m.

the R version we need to compute $x^{-1}X$ for some $X \in GF(2^n)$. Multiplication by x^{-1} can be easily done in $GF(2^n)$, we illustrate this by an example. Let us consider the field $GF(2^{128})$ represented by the irreducible polynomial $\tau(x) = x^{128} \oplus x^7 \oplus x^2 \oplus x \oplus 1$, let Q be the n bit binary string representation of the polynomial $x^{-1}(\tau(x) \oplus 1) = x^{127} \oplus x^6 \oplus x \oplus 1$. Then, for any $X \in GF(2^n)$, $x^{-1}X$ can be realized as follows,

$$x^{-1}X = \begin{cases} X >> 1 & \text{if } \mathsf{lsb}(X) = 0 \\ (X >> 1) \oplus Q & \text{otherwise.} \end{cases}$$

Where $X >> 1$ represents right shift of X by one bit and $\mathsf{lsb}(X)$ denotes the least significant bit of X. Thus, decryption of the R version of a ciphertext has a little overhead more than the encryption.

Multiplication by $(1 \oplus x)^{-1}$ amounts to a full multiplication. So, decryption of the L version requires m block cipher calls and m finite field multiplications in addition to the call to the PRF. Here, $(1 \oplus x)^{-1}$ can be pre-computed, thus explicit computation of the inverse would not be required. Also note that the multiplication required can be performed quite efficiently using pre-computation as one of the operands is fixed, and this operand ($(1 \oplus x)^{-1}$) does not contain any key related materials. As required, the function g can be computed much more efficiently than the decryption function, as for recovering a m block message with g, one needs to perform only m xors of n bit strings.

2. **Number of keys:** This generic construction requires more than one key, one for the block-cipher and another (or more) for the PRF depending on the number of keys required by the PRF. This requirement can be reduced, later we describe a particular construction which requires only one block-cipher key.

3. **Message length restrictions:** The construction only works for fixed length messages which are multiples of the block length of the block-cipher. We do not know any trivial way to remove this restriction from DCMG, as if there is a broken block (a block of length less than the block length of the block-cipher)then there could be a ciphertext expansion (see lines 5 and 6 of the encryption algorithm in Figure 11.1). But this restriction is not severe keeping in mind the application, as it is assumed that the two ciphertexts produced would be disk sectors to be stored in two different disk volumes.

4. **The tweak length:** The construction is specified for fixed length tweaks, where the tweak length is the same as the block length of the underlying blockcipher. For the application of the disk sector encryption, the sector address is considered as the tweak which is likely to be of fixed length.

5. **Similarity with DAE:** One can note the similarity between the generic construction of a DCM with that of the SIV scheme described in Figure 10.1. A SIV scheme which uses the counter-mode for the secure privacy only scheme can be suitably re-worked to make a DCM following the construction in Section 11.2. The stream of random bits which would be generated by the counter mode needs to be xor-ed with the plaintext in two different ways to get the two different versions of the ciphertext. But in the generic DAE scheme as described in [119] any privacy only scheme can be used and also have the provision of vector headers (headers are equivalent to the tweak), so we do not use the syntax of a DAE to define DCM, but we just note the similarity.

   The security of DAE as described in Section 10.1.1 is also similar to that of a DCM, but a DAE adversary does not query a type to its oracle, as there is no such possibility in case of a DAE. In [119] a combined advantage for both security and authentication was proposed, which is elegant, but we feel that the proofs are still simpler if the privacy and authenticity advantages are stated separately, so we define the privacy and authentication advantages separately.

11.2.2 Security of DCMG

The following theorems suggests the security of DCMG

Theorem 11.1. *Let* $\Upsilon = \mathsf{Func}(mn+n, n)$ *and* $\Pi = \mathsf{Perm}(n)$*. Let* $\mathcal{A}$ *be an adversary*

attacking $\mathrm{DCMG}[\Upsilon, \Pi]$ *who asks* q *queries, then*

$$\mathbf{Adv}^{\text{dcm-priv}}_{\mathrm{DCMG}[\Upsilon,\Pi]}(\mathcal{A}) \leq \frac{m^2q^2}{2^n} \tag{11.1}$$

$$\mathbf{Adv}^{\text{dcm-auth}}_{\mathrm{DCMG}[\Upsilon,\Pi]}(\mathcal{A}) \leq \frac{1}{2^n} \tag{11.2}$$

Theorem 11.2. *Let* $F : \mathcal{K}_1 \times \{0,1\}^{mn+n} \to \{0,1\}^n$ *be a PRF and* $E : \mathcal{K}_2 \times \{0,1\}^n \to \{0,1\}^n$ *be a block-cipher secure in the PRP sense. Let* $\mathcal{A}$ *be an adversary attacking* $\mathrm{DCMG}[F, E]$ *who asks* q *queries, then there exist adversaries* $\mathcal{A}'$ *and* $\mathcal{A}''$ *such that*

$$\mathbf{Adv}^{\text{dcm-priv}}_{\mathrm{DCMG}[F,E]}(\mathcal{A}) \leq \frac{m^2q^2}{2^n} + \mathbf{Adv}^{\text{prf}}_{F}(\mathcal{A}') + \mathbf{Adv}^{\text{prp}}_{E}(\mathcal{A}'') \tag{11.3}$$

$$\mathbf{Adv}^{\text{dcm-auth}}_{\mathrm{DCMG}[F,E]}(\mathcal{A}) \leq \frac{1}{2^n} + \frac{m^2q^2}{2^n} + \mathbf{Adv}^{\text{prf}}_{F}(\mathcal{A}') + \mathbf{Adv}^{\text{prp}}_{E}(\mathcal{A}'') \tag{11.4}$$

where $\mathcal{A}'$ *and* A'' *asks* $O(q)$ *queries and run for time* $t + O(q)$ *where* t *is the running time of* $\mathcal{A}$.

Theorem 11.1 depicts the information theoretic bound where as Theorem 11.2 shows the complexity theoretic bound. Transition from Theorem 11.1 to 11.2 is quite standard and we shall not present it. We provide the full proof of Theorem 11.1 in Section 11.4. But here we shall provide some motivation regarding why Theorem 11.1 is true.

In the construction which is being referred to in Theorem 11.1 the pseudorandom function F is replaced by a function ρ chosen uniformly at random from $\Upsilon = \mathrm{Func}(mn + n, n)$ and the blockcipher E is replaced by a permutation π chosen uniformly at random from $\Pi = \mathrm{Perm}(n)$. To prove the privacy bound note that the inputs to the function ρ would be all new, as the adversary is not allowed to repeat a query. Also, as ρ is a random function so the tag (which is the output of ρ) would be random. The only way the adversary can distinguish the output of $\mathrm{DCMG}[\Upsilon, \Pi]$ from random strings if there is a collision in the domain or the range sets of the permutation π, the probability of this collision in $m^2q^2/2^n$, which suggests the bound in eq. (11.1). For proving authenticity we give access of the random permutation π to the adversary, as this permutation π is independent of the random function ρ the adversary have no added advantage of forgery. With access to π the adversary will know what would be the input to the random function ρ for the forgery (s)he produces. Given his query restrictions, the probability that the adversary can produce a successful forgery would be less than $1/2^n$ (for details see the full proof in Section 11.4).

11.3 Constructing a DCM Using BRW Polynomials

Here we provide a particular construction of a DCM, where we design the pseudorandom function using a special type of polynomial called the BRW polynomial. Also our construction requires only one blockcipher key. We call the construction as DCM-BRW. BRW polynomials have been proposed for use in message authentication codes and tweakable enciphering schemes [10, 125], where significant performance gains are obtained as computing these polynomials requires about half the number of multiplications compared to computing an usual polynomial. Usage of a BRW polynomial in construction of a DCM also gives us significant gain in performance by reducing the number of multiplications required.

11.3.1 The Construction

The generic construction DCMG that we proposed in Section 11.2 uses a pseudorandom function along with a block-cipher in a counter type mode of operation. In the specific construction DCM-BRW we replace the general pseudorandom function with a specific pseudorandom function which uses a BRW polynomial. The encryption and decryption algorithms using DCM-BRW are shown in Figure 11.3. The construction requires two keys, a key h for the BRW polynomial and a key K for the block-cipher. Other than the keys the encryption algorithm takes in the plaintext, the tweak and the type and returns a ciphertext and the decryption algorithm takes in the ciphertext, tweak and type and returns the plaintext. The details of the working of the algorithm are self explanatory as depicted in Figure 11.3. For this construction it is required that the irreducible polynomial representing the field $GF(2^n)$ is primitive.

From Chapter 6 we can say that to encrypt m block of messages DCM-BRW requires $\lfloor \frac{m+1}{2} \rfloor + 1$ multiplications and $\lg(m+1)$ squarings. Computing squares in binary fields are much more efficient than multiplication. In addition it requires $(m+3)$ block-cipher calls. Out of these $(m+3)$ block-cipher calls two can be pre-computed, but this would amount to the requirement of storage of key related materials which is not recommended. The construction requires two keys, h the key for the BRW polynomial and K the block-cipher key. Requirement of a single block-cipher key is what is important, as for multiple block-cipher keys one needs to have different key schedules which may make an implementation inefficient when implemented in hardware. One can probably generate the key h using the block-cipher key and still obtain a secure construction, but this would generally mean one more block-cipher call or a storage of key related material which is same as storage of an extra key. Also having a different key for the polynomial provides more flexibility during changing of keys. So, we decided to keep the block-cipher key independent of the key for the BRW polynomial.

The constructions requires two passes over the data, one for computing the tag τ and other for generating the ciphertext. The ciphertext is generated using a counter type mode of

Algorithm DCM-BRW.$\mathcal{E}_{h,K}^{T,\mathsf{ty}}(P_1,\ldots,P_m)$	**Algorithm** DCM-BRW.$\mathcal{D}_{h,K}^{T,\mathsf{ty}}(C_1,\ldots,C_m,\tau)$
1. $\alpha = E_K(0)$; $\beta = E_K(1)$;	1. $\alpha = E_K(0)$; $\beta \leftarrow E_K(1)$;
2. $\gamma \leftarrow h \cdot \mathsf{BRW}_h(P_1\|\|P_2\|\|\ldots\|\|P_m\|\|T)$	2. **for** $j = 1$ to m,
3. $\tau \leftarrow E_K(\gamma \oplus \alpha)$;	3. $R_j \leftarrow E_K(\tau \oplus x^j\beta)$;
4. **for** $j = 1$ to m	4. **if** $\mathsf{ty} = \mathsf{L}$
5. $R_j \leftarrow E_K(\tau \oplus x^j\beta)$	5. $P_i \leftarrow (C_j \oplus R_j)(1 \oplus x)^{-1}$
6. **if** $\mathsf{ty} = \mathsf{L}$	6. **else** $P_j \leftarrow (C_j \oplus R_j)x^{-1}$
7. $C_j \leftarrow R_j \oplus (1 \oplus x)P_j$	7. **endif**
8. **else** $C_j \leftarrow R_j \oplus xP_j$	8. **endfor**
9. **endif**	9. $\gamma \leftarrow h \cdot \mathsf{BRW}_h(P_1\|\|P_2\|\|\ldots\|\|P_m\|\|T)$;
10. **endfor**	10. $\tau' \leftarrow E_K(\gamma \oplus \alpha)$
11. return $(C_1\|\|C_2\|\|\ldots\|\|C_m\|\|\tau)$	11. **if** $\tau' = \tau$ **return** $(P_1,\ldots,P_m)$ else **return** $\perp$;

Figure 11.3: Encryption and decryption using DCM-BRW.

operation which can be parallelized, also α and β can be computed in parallel with the BRW polynomial. Thus the construction offers flexibility for efficient pipelined implementation. Also, for an efficient hardware implementation the only non-trivial blocks that are needed to be implemented are a finite field multiplier and a encryption only block-cipher. So, it is expected that a hardware implementation will have a small footprint.

11.3.2 Comparisons

To our knowledge we do not know of any other existing cryptographic scheme which provides the same functionality that is provided by a DCM. As mentioned before one can use a tweakable enciphering scheme to do secure backup, by writing the cipher in two different locations. Also, a certain class of DAE schemes can be used for constructing a DCM. Hence, we compare the efficiency of DAE-BRW with existing tweakable enciphering schemes (TES) and DAE constructions. As DCM is completely a different primitive compared to either a TES or a DAE so these comparisons should be interpreted carefully, in particular we want to make explicit the following points before we present the real comparisons:

1. Any DCM scheme is not meant to be a competitor for any existing TES. The security guarantees that a TES provides are completely different from that of DCM. A TES is tag-less, but one needs to store a tag for any DCM scheme which amounts to extra storage. Moreover, the property of key-less recovery using the function g cannot be provided by a TES. Thus the comparison only point out the case of efficiency, but the difference in functionality and security guarantees should be considered while

interpreting the efficiency comparisons.

2. Similarly, a DAE scheme which was proposed as a solution for the key-wrap problem [119] does not provide the security or functionality of a DCM. But as we mentioned, a certain class of DAE schemes can be modified to obtain a DCM. Also the DCM-BRW can be modified to make it a DAE, the only modification required would be to output $R_j \oplus P_i$ (refer to the encryption algorithm in Figure 11.3) instead of mixing R_j in two different ways with the plaintext to obtain two ciphertext. With this modification DCM-BRW would be converted into a secure DAE scheme which works on single block headers. But this is not the point that we are trying to focus here. The comparisons with DAE are meant to show that how efficient DCM-BRW would be compared to other possible constructions of DCM which can be easily derived by modifying existing DAE constructions. The efficiency in our construction compared to other DAE constructions comes from the use of BRW polynomials which require less multiplications.

3. The way we present the algorithms for DCM-BRW (also DCMG) one may think that to produce the two versions of the ciphertexts one need two invocations of the algorithm, we presented in this way to make it clear that depending on the parameter ty the algorithm produces different ciphertexts for different messages, this way of presentation helps us to argue about the security in a better manner in the proofs. But it is clear from the algorithms that from a single invocation of the algorithm both versions of the ciphertext can be produced and this would not involve any significant extra computational overhead (except a few xors).

Comparison with Tweakable Enciphering Schemes: In Table 11.1 we compare the number of operations required for tweakable enciphering schemes for fixed length messages which uses n bit tweaks with the number of operations required for DCM-BRW.

From Table 11.1 we can see that encrypting with DCM-BRW would be much more efficient than encrypting by any of the existing tweakable enciphering schemes. But, it is to be noted that the security guarantee that a tweakable enciphering scheme provides is very different from that provided by a DCM, hence this comparison needs to be appropriately interpreted.

Comparison with Deterministic Authenticated Encryption Schemes: As we have mentioned, deterministic authenticated encryption (DAE) schemes which uses counter mode of operation can be easily converted into a DCM scheme by only using additional xor operations. There are three DAE schemes reported till date. The SIV mode [119] uses CMAC along with the counter mode, and [74,75] uses a variant of a polynomial hash with a counter mode. All these schemes can be reworked to construct a DCM. But as evident from Table 11.2 DCM-BRW would be much more efficient than any of them. For the DAE schemes the operation counts shown in Table 11.2 are based on only one block of tweak.

There are striking structural similarities between DCM-BRW and BCTR though DCM-BRW provides a completely different functionality. But an implementation of BCTR can be very

Table 11.1: Comparison of DCM-BRW with tweakable enciphering schemes for fixed length messages which uses n bit tweak. [BC]: Number of block-cipher calls; [M]: Number of multiplications, [BCK]: Number of blockcipher keys, [OK]: Other keys, including hash keys.

Mode	[BC]	[M]	[BCK]	[OK]
CMC [65]	$2m+1$	–	1	–
EME [66]	$2m+1$	–	1	–
XCB [101]	$m+1$	$2(m+3)$	3	2
HCTR [141]	m	$(2m+1)$	1	1
HCHfp [31]	$m+2$	$2(m-1)$	1	1
TET [64]	$m+1$	$2m$	2	3
Constructions in [125] using normal polynomials	$m+1$	$2(m-1)$	1	1
Constructions in [125] using BRW polynomials	$m+1$	$2+2\lfloor (m-1)/2 \rfloor$	1	
DCM-BRW	$m+3$	$1+\lfloor (m+1)/2 \rfloor$	1	1

Table 11.2: Comparison between DCM-BRW and DAE schemes for encrypting m blocks of messages. In the DAE schemes the operation counts are based on only one block of tweak. [BC]: Number of block-cipher calls; [M]: Number of multiplications, [BCK]: Number of blockcipher keys, [OK]: Other keys, including hash keys.

Mode	$[BC]$	$[M]$	[BCK]	[OK]
SIV [119]	$2m+3$	–	2	–
HBS [75]	$m+2$	$m+3$	1	–
BTM [74]	$m+3$	m	1	–
DCM-BRW	$m+3$	$1+\lfloor (m+1)/2 \rfloor$	1	1

easily converted to that of DCM-BRW and DCM-BRW will have the same performance in hardware as BCTR. For this reason, we do not provide separate description of implementations of DCM-BRW.

11.3.3 Security of DCM-BRW

The following theorem specifies the security of DCM-BRW.

Theorem 11.3. *Let* $\Pi = \mathit{Perm}(n)$. *Let* $\mathcal{A}$ *be an adversary attacking* DCM-BRW[Π] *who asks* q *queries, then*

$$\mathbf{Adv}^{\text{dcm-priv}}_{\text{DCM-BRW}[\Pi]}(\mathcal{A}) \leq \frac{14m^2q^2}{2^n} \tag{11.5}$$

$$\mathbf{Adv}^{\text{dcm-auth}}_{\text{DCM-BRW}[\Pi]}(\mathcal{A}) \leq \frac{1}{2^n} + \frac{18m^2q^2}{2^n} \tag{11.6}$$

Theorem 11.4. *Let* $E : \mathcal{K} \times \{0,1\}^n \rightarrow \{0,1\}^n$ *be a block-cipher secure in the PRP sense. Let* $\mathcal{A}$ *be an adversary attacking* DCM-BRW[E] *who asks* q *queries, then there exists an adversary* $\mathcal{A}'$ *such that*

$$\mathbf{Adv}^{\text{dcm-priv}}_{\text{DCM-BRW}[E]}(\mathcal{A}) \leq \frac{14m^2q^2}{2^n} + \mathbf{Adv}^{\text{prp}}_{E}(\mathcal{A}') \tag{11.7}$$

$$\mathbf{Adv}^{\text{dcm-auth}}_{\text{DCM-BRW}[E]}(\mathcal{A}) \leq 2\mathbf{Adv}^{\text{prp}}_{E}(\mathcal{A}') + \frac{1}{2^n} + \frac{18m^2q^2}{2^n} \tag{11.8}$$

where $\mathcal{A}'$ *asks* $O(q)$ *queries and run for time* $t + O(q)$ *where* t *is the running time of* $\mathcal{A}$.

The authentication and privacy bounds for DCM-BRW are same as in BCTR, this is because of their structural similarities, and the proof of Theorem 11.3 also closely resembles the proof for BCTR, with some subtle differences. We provide the full proof of Theorem 11.3 in the next section.

11.4 Proofs

Proof of Theorem 1

To prove the security of DCM we replace the pseudorandom function F in Figure 11.1 by a function ρ chosen uniformly at random from $\Upsilon = \mathsf{Func}(mn+n, n)$, and the block cipher E by a permutation π chosen uniformly at random from $\Pi = \mathsf{Perm}(n)$. We call the encryption function of DCM[Υ, Π] as $\mathcal{E}_{\rho,\pi}$. We consider that $\mathcal{A}$ interacts with the game DCM1 as depicted in Figure 11.4. Notice that in DCM1, the proper oracles for $\mathcal{A}$ are provided,

the random function ρ and the random permutation π are constructed on the fly using the subroutines Ch-$\pi(.)$ and Ch-$\rho(.)$ respectively. The domain and range sets of the permutation π are maintained in the sets $Domain_\pi$ and $Range_\pi$ respectively, and appropriate checks are made so that the subroutine Ch-$\pi(X)$ indeed behaves like a permutation. No checks are necessary in case of Ch-$\rho(X)$, as according to the query restrictions of $\mathcal{A}$, Ch-$\rho(.)$ is always called with a fresh (new) input.

For a game G, let $\Pr[\mathcal{A}^G \Rightarrow 1]$ denote the probability that $\mathcal{A}$ outputs 1 by interacting with the game G. Then we have,

$$\Pr\left[\mathcal{A}^{\mathcal{E}_{\rho,\pi}(.,.,.)} \Rightarrow 1\right] = \Pr[\mathcal{A}^{\mathrm{DCM1}} \Rightarrow 1]. \tag{11.9}$$

We modify game DCM1 by deleting the boxes entries in Figure 11.4 and call the modified game as DCM2. By deleting the boxed entries it cannot be guaranteed that Ch-π is a permutation as though we do the consistency checks but we do not reset the values of Y in Ch-$\pi(.)$. Then the games DCM1 and DCM2 are identical except when the bad flag is set, thus we have

$$\Pr[\mathcal{A}^{\mathrm{DCM1}} \Rightarrow 1] - \Pr[\mathcal{A}^{\mathrm{DCM2}} \Rightarrow 1] \leq \Pr[\mathcal{A}^{\mathrm{DCM2}} \text{ sets } \mathsf{bad}] \tag{11.10}$$

In the game DCM2, $\mathcal{A}$ always gets random strings as a response to an encryption query. This can be seen in lines 101 and 103 of the game DCM2, where τ^s and R_i^s $(1 \leq i \leq m)$ gets set to a random n bit strings. Thus,

$$\Pr[\mathcal{A}^{\mathrm{DCM2}} \Rightarrow 1] = \Pr\left[\mathcal{A}^{\$(.,.,.)} \Rightarrow 1\right]. \tag{11.11}$$

So using Equations (11.9), (11.10) and (11.11) we have

$$\begin{aligned}\mathbf{Adv}^{\text{dcm-priv}}_{\mathrm{DCMG}[\Upsilon,\Pi]}(\mathcal{A}) &= \Pr\left[\mathcal{A}^{\mathcal{E}_{\rho,\pi}(.,.,.)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\$(.,.,.)} \Rightarrow 1\right] \\ &\leq \Pr[A^{\mathrm{DCM2}} \text{ sets } \mathsf{bad}] \end{aligned} \tag{11.12}$$

Now, we make a game which no more use subroutines Ch-π and Ch-ρ, and immediately returns random strings to the adversary in response to his encryptions queries. Later we maintain the multi-sets D_π and R_π where we list the elements that were supposed to be inputs and outputs of the permutation. Finally we check the collisions in the multi-sets D_π and R_π, and if there is any collision we set the bad flag to true. We call this game as DCM3, which is shown in Figure 11.5. The games DCM2 and DCM3 are indistinguishable to the adversary, because in both cases it gets random strings in response to his queries. Hence the probability with which DCM2 sets bad is same as the probability with which DCM3 sets bad. Thus we get,

Subroutine Ch-$\pi(X)$

11. $Y \xleftarrow{\$} \{0,1\}^n$; **if** $Y \in Range_\pi$ **then** bad ← true; $\boxed{Y \xleftarrow{\$} \overline{Range_\pi}}$; **endif**;
12. **if** $X \in Domain_\pi$ **then** bad ← true; $\boxed{Y \leftarrow \pi(X)}$; **endif**
13. $\pi(X) \leftarrow Y$; $Domain_\pi \leftarrow Domain_\pi \cup \{X\}$; $Range_\pi \leftarrow Range_\pi \cup \{Y\}$; **return**$(Y)$;

Subroutine Ch-$\rho(M)$

14. $Y \xleftarrow{\$} \{0,1\}^n$;
15. **return**(Y);

Initialization:

16. $Domain_\pi \leftarrow Range_\pi \leftarrow \emptyset$;
17. **for** all $X \in \{0,1\}^n$, $\pi(X) =$ **undefined endfor**
18. bad ← false

Respond to the sth encryption query of $\mathcal{A}$, $(T^s; P_1^s||P_2^s||\ldots||P_m^s; \mathsf{ty}^s)$ as follows:

101. $\tau^s \leftarrow \rho(P_1^s||P_2^s||\ldots||P_m^s||T^s)$;
102. **for** $i = 1$ to m,
103. $R_i^s \leftarrow \pi(\tau^s \oplus \mathsf{bin}_n(i))$;
104. **if** $\mathsf{ty}^s = \mathsf{L}$ **then**
105. $C_i^s \leftarrow R_i^s \oplus (1 \oplus x)P_i^s$;
106. **else**
107. $C_i^s \leftarrow R_i^s \oplus xP_i^s$;
108. **endif**
109. **endfor**
110. **return** $(C_1^s||C_2^s||\ldots||C_{m^s}^s||\tau^s)$

Figure 11.4: Games DCM1 and DCM2: DCM2 is the game without the boxed entries in the subroutine Ch-$\pi()$

Respond to the sth encryption query $(T^s; P_1^s \| P_2^s \| \ldots \| P_{m^s}^s; \mathsf{ty}^s)$ as follows $\quad C_1^s \| C_2^s \| \ldots \| C_m^s \| \tau^s \xleftarrow{\$} \{0,1\}^{(m+1)n}$ $\quad$ **Return** $C_1^s \| C_2^s \| \ldots \| C_{m^s}^s \| \tau^s$
Finalization:
FIRST PHASE **for** $s = 1$ to q $\quad$ **for** $i = 1$ to m, $\quad\quad D_\pi \leftarrow D_\pi \cup \{\tau^s \oplus \mathsf{bin}_n(i)\}$ $\quad\quad$ **if** $\mathsf{ty}^s = \mathsf{L}$, **then** $\delta_i^s \leftarrow (1 \oplus x)P_i^s$; $\quad\quad$ **else** $\delta_i^s \leftarrow xP_i^s$ $\quad\quad$ **end if** $\quad\quad R_\pi \leftarrow R_\pi \cup \{C_i^s \oplus \delta_i^s\}$ $\quad$ **end for** **end for**
SECOND PHASE $\quad \mathsf{bad} \leftarrow \mathsf{false}$; $\quad$ **if** (some value occurs more than once in D_π) **then** $\mathsf{bad} = \mathsf{true}$ **end if**; $\quad$ **if** (some value occurs more than once in R_π) **then** $\mathsf{bad} = \mathsf{true}$ **end if**.

Figure 11.5: Game DCM3: D_π and R_π are multisets, which are initially empty. We assume that $\mathcal{A}$ makes q queries.

$$\Pr[\mathcal{A}^{\mathrm{DCM2}} \text{ sets } \mathsf{bad}] = \Pr[\mathcal{A}^{\mathrm{DCM3}} \text{ sets } \mathsf{bad}] \tag{11.13}$$

Thus from equations (11.12) and (11.13) we obtain

$$\mathbf{Adv}^{\text{dcm-priv}}_{\mathrm{DCMG}[\Upsilon,\Pi]}(\mathcal{A}) \leq \Pr[\mathcal{A}^{\mathrm{DCM3}} \text{ sets } \mathsf{bad}] \tag{11.14}$$

Now our goal would be to bound $\Pr[A^{\mathrm{DCM3}}$ sets $\mathsf{bad}]$. We can see in Game DCM3 that the bad flag is set when there is a collision in either D_π or R_π. So if COLD_π and COLR_π denote the events of a collision in D_π and R_π respectively then we have

$$\Pr[\mathcal{A}^{\mathrm{DCM3}} \text{ sets } \mathsf{bad}] = \Pr[\mathsf{COLD}_\pi] + \Pr[\mathsf{COLR}_\pi]. \tag{11.15}$$

Now, we shall bound the collision probabilities in the domain and range sets. From Figure 11.5 we see that $D_\pi = \{\tau^s \oplus \mathsf{bin}_n(i) : 1 \leq i \leq m, 1 \leq s \leq q\}$, where τ^s is an element chosen uniformly at random from $\{0,1\}^n$. And, $R_\pi = \{C_i^s \oplus \delta_i^s : 1 \leq i \leq m, 1 \leq s \leq q\}$, where $\delta_i^s = (1 \oplus x)P_i$ when $\mathsf{ty}^s = \mathsf{L}$ and $\delta_i^s = xP_i$ when $\mathsf{ty}^s = \mathsf{R}$, and C_i^s is chosen uniformly at random from $\{0,1\}^n$.

Thus, the probability of collisions between two elements in D_π is given by $\Pr[\mathsf{COLD}_\pi] = \Pr[\tau^s \oplus \mathsf{bin}_n(i) = \tau^t \oplus \mathsf{bin}_n(j)]$ where s and t are two different queries, $1 \leq i, j \leq m$, this probability is at most 2^{-n}, as τ^s is always a string chosen uniformly at random from $\{0,1\}^n$. The number of elements in D_π is equal to qm, thus we have

$$\Pr[\mathsf{COLD}_\pi] \leq \binom{mq}{2}\frac{1}{2^n} < \frac{m^2q^2}{2^{n+1}} \tag{11.16}$$

As each C_i^s is chosen uniformly at random from $\{0,1\}^n$ and R_π contains mq elements, hence we have

$$\Pr[\mathsf{COLR}_\pi] \leq \frac{m^2q^2}{2^{n+1}} \tag{11.17}$$

Finally we have

$$\mathbf{Adv}^{\text{dcm-priv}}_{\mathrm{DCMG}[\Upsilon,\Pi]}(\mathcal{A}) \leq \Pr[\mathsf{COLD}_\pi] + \Pr[\mathsf{COLR}_\pi] \leq \frac{m^2q^2}{2^n} \tag{11.18}$$

which completes the proof for the privacy bound.

Proving the authentication bound: To bound the authentication advantage, we give an oracle access of the random permutation π to the adversary $\mathcal{A}$. We denote the adversary

$\mathcal{A}$ with an oracle access to π as $\mathcal{A}(\pi)$. Thus we can say that

$$\mathbf{Adv}^{\text{dcm-auth}}_{\text{DCMG}[\Upsilon,\Pi]}(\mathcal{A}) \leq \mathbf{Adv}^{\text{dcm-auth}}_{\text{DCMG}[\Upsilon,\Pi]}(\mathcal{A}(\pi)),$$

as $\mathcal{A}$ without oracle access to π can do no better than $\mathcal{A}(\pi)$. $\mathbf{Adv}^{\text{dcm-auth}}_{\text{DCMG}[\Upsilon,\Pi]}(\mathcal{A}(\pi))$ is bounded by the probability that $\mathcal{A}(\pi)$ produces a forgery $(\mathcal{C}, T, \mathsf{ty})$, such that $\mathcal{D}_{\rho,\pi}(\mathcal{C}, T, \mathsf{ty}) \neq \perp$. A forgery $(C_1||\ldots||C_m||\tau, T, \mathsf{ty})$ can be successful if $\rho(X||T) = \tau$, where $X = X_1||X_2||\ldots||X_m$, and $X_i = (1 \oplus x)^{-1}[C_i \oplus \pi(\tau \oplus \mathsf{bin}_n(i))]$ when $\mathsf{ty} = \mathsf{L}$ and $X_i = x^{-1}[C_i \oplus \pi(\tau \oplus \mathsf{bin}_n(i))]$ when $\mathsf{ty} = \mathsf{R}$.

As $\mathcal{A}(\pi)$ has an oracle access to π, so it can compute X for any chosen $\mathcal{C}$. Thus, when it produces a forgery $(T, \mathcal{C}, \mathsf{ty})$ it knows what would be the input to the function ρ and also the target output for his forgery. But according to the query restrictions, the adversary never produces $(T, C||\tau, \mathsf{ty})$ as a forgery if it obtained $C||\tau$ as an reply to an encryption query of the form (T, X, ty). Also it does not produce $(T, C||\tau, \mathsf{ty})$ as a forgery if it obtained $\tilde{C}||\tau$ as a response to its query $(T, X, \bar{\mathsf{ty}})$, where $g(C, \tilde{C}) = X$. Thus either the input $X||T$ to ρ is new or the output τ is new, thus the probability that $\rho(X||T) = \tau$ is at most $1/2^n$, hence we have

$$\mathbf{Adv}^{\text{dcm-auth}}_{\text{DCMG}[\Upsilon,\Pi]}(\mathcal{A}) \leq \mathbf{Adv}^{\text{dcm-auth}}_{\text{DCMG}[\Upsilon,\Pi]}(\mathcal{A}(\pi)) \leq \frac{1}{2^n}. \tag{11.19}$$

Which completes the proof. □

Proof of Theorem 3

Proof of Theorem 3: To prove the security of DCM-BRW[Π] we replace the the block cipher E in the construction of Figure 11.3 by a permutation π chosen uniformly at random from $\Pi = \mathsf{Perm}(n)$. We call the encryption function of DCM-BRW[Π] as $\mathcal{E}_{h,\pi}$, where $h \xleftarrow{\$} \{0,1\}^n$ is the key for the BRW polynomial. We prove the privacy bound first. We consider the same game playing technique as used in the proof of Theorem 1. We briefly discuss the games below:

1. Game G0: In G0 (shown in Figure 11.6) the block-cipher is replaced by the random permutation π. The permutation π is constructed on the fly keeping record of the domain and range sets as done in the sub-routine Ch-π in Figure 11.6. Thus, G0 provide the proper encryption oracle to $\mathcal{A}$. Thus we have:

$$\Pr\left[\mathcal{A}^{\mathcal{E}_{h,\pi}(.,.,.)} \Rightarrow 1\right] = \Pr[\mathcal{A}^{\text{G0}} \Rightarrow 1]. \tag{11.20}$$

2. Game G1: Figure 11.6 with the boxed entries removed represents the game G1. In G1 it is not guaranteed that Ch-π behaves like a permutation, but the games G0 and G1 are identical until the bad flag is set. Thus we have

$$\Pr[\mathcal{A}^{\mathrm{G0}} \Rightarrow 1] - \Pr[\mathcal{A}^{\mathrm{G1}} \Rightarrow 1] \leq \Pr[\mathcal{A}^{\mathrm{G1}} \text{ sets } \mathsf{bad}] \tag{11.21}$$

Note that in G1 the adversary gets random strings as output irrespective of his queries. Hence,

$$\Pr[\mathcal{A}^{\mathrm{G1}} \Rightarrow 1] = \Pr[\mathcal{A}^{\$(.,.,.)} \Rightarrow 1] \tag{11.22}$$

3. Game G2: In Game G2 (shown in Figure 11.8) we do not use the subroutine Ch-π any more but return random strings immediately after the $\mathcal{A}$ asks a query. Later we keep track of the elements that would have got in the domain and range sets of the permutation π in multi-sets $\mathcal{S}$ and $\mathcal{R}$. We set the bad flag when there is a collision in either $\mathcal{S}$ or $\mathcal{R}$. For the adversary the games G1 and G2 are identical. So,

$$\Pr[\mathcal{A}^{\mathrm{G1}} \Rightarrow 1] = \Pr[\mathcal{A}^{\mathrm{G2}} \Rightarrow 1] \tag{11.23}$$

and

$$\Pr[\mathcal{A}^{\mathrm{G1}} \text{ sets } \mathsf{bad}] = \Pr[\mathcal{A}^{\mathrm{G2}} \text{ sets } \mathsf{bad}] \tag{11.24}$$

Hence, using Eqs. (11.20), (11.21), (11.22), (11.23) and (11.24), we have

$$\Pr\left[\mathcal{A}^{\mathcal{E}_{h,\pi}(.,.,.)} \Rightarrow 1\right] - \Pr[A^{\$(.,.,.)} \Rightarrow 1] \leq \Pr[\mathcal{A}^{\mathrm{G2}} \text{ sets } \mathsf{bad}].$$

According to the definition of the privacy advantage of $\mathcal{A}$, we have

$$\mathbf{Adv}^{\text{dcm-priv}}_{\text{DCM-BRW}[\Pi]}(\mathcal{A}) \leq \Pr[\mathcal{A}^{\mathrm{G2}} \text{ sets } \mathsf{bad}] \tag{11.25}$$

Now we need to bound $\Pr[\mathcal{A}^{\mathrm{G2}} \text{ sets } \mathsf{bad}]$. The elements in the multi-sets $\mathcal{S}$ and $\mathcal{R}$ would be

$$\mathcal{S} = \{0,1\} \cup \{\gamma^s \oplus EZ : 1 \leq s \leq q\} \cup \{\tau^s \oplus x^i EO : 1 \leq i \leq m, 1 \leq s \leq q\} \tag{11.26}$$

$$\mathcal{R} = \{EZ, EO\} \cup \{\tau^s : 1 \leq s \leq q\} \cup \{C_i^s \oplus \delta_i : 1 \leq i \leq m, 1 \leq s \leq q\}, \tag{11.27}$$

where $\delta_i^s = xP_i$ when $\mathsf{ty}^s = \mathsf{L}$ and $\delta_i^s = (x \oplus 1)P_i$ when $\mathsf{ty}^s = \mathsf{R}$. Let COLLD be the event that there is a collision in $\mathcal{S}$ and COLLR be the event that there is a collision in $\mathcal{R}$ Using the facts that $\gamma^s = h\mathsf{BRW}_h(P_1^s||P_2^s||\ldots||P_m^s||T^s)$ and EZ, EO, τ^s, C_i^s are random elements of $\{0,1\}^n$, we have

$$\begin{aligned}
\Pr[\mathsf{COLLD}] &\leq \frac{2q}{2^n} + \frac{2mq}{2^n} + \binom{q}{2}\frac{(2m+2)}{2^n} + \binom{mq}{2}\frac{1}{2^n} + \frac{2mq^2(m+1)}{2^n} \\
&= \frac{1}{2^n}\left(\frac{5}{2}m^2q^2 + 3mq^2 + \frac{mq}{2} + q + q^2\right)
\end{aligned}$$

Subroutine Ch-$\pi(X)$

01. $Y \xleftarrow{\$} \{0,1\}^n$; **if** $Y \in Range_\pi$ **then** $\mathsf{bad} \leftarrow \mathsf{true}$; $\boxed{Y \xleftarrow{\$} \overline{Range_\pi}}$; **endif**;
02. **if** $X \in Domain_\pi$ **then** $\mathsf{bad} \leftarrow \mathsf{true}$; $\boxed{Y \leftarrow \pi(X)}$; **endif**
03. $\pi(X) \leftarrow Y$; $Domain_\pi \leftarrow Domain_\pi \cup \{X\}$; $Range_\pi \leftarrow Range_\pi \cup \{Y\}$; **return**$(Y)$;

Initialization:

11. **for** all $X \in \{0,1\}^n$ $\pi(X) = \mathsf{undefined}$ **endfor**
12. $EZ \xleftarrow{\$} \{0,1\}^n$; $\pi(0) = EZ$;
13. $Domain_\pi \leftarrow \{0\}$; $Range_\pi \leftarrow \{EZ\}$;
14. $EO \xleftarrow{\$} \{0,1\}^n \setminus \{EZ\}$; $\pi(1) = EO$;
15. $Domain_\pi \leftarrow Domain_\pi \cup \{1\}$; $Range_\pi \leftarrow Range_\pi \cup \{EO\}$;
16. $\mathsf{bad} = \mathsf{false}$

Respond to the sth encryption query $(T^s; P_1^s||P_2^s||\ldots||P_m^s; ty^s)$ as follows:

101. $\gamma^s \leftarrow h \cdot BRW(P_1^s||P_2^s||\ldots||P_m^s||T^s)$;
102. $\tau^s \leftarrow$ Ch-$\pi(\gamma^s \oplus EZ)$;
103. **for** $i = 1$ to m,
104. $R_i^s \leftarrow$ Ch-$\pi(\tau^s \oplus x^i EO)$;
105. **if** $ty^s = L$ **then** $C_i^s \leftarrow R_i^s \oplus xP_i^s$;
106. **else** $C_i^s \leftarrow R_i^s \oplus (x \oplus 1)P_i^s$;
107. **endif**
108. **endfor**
109. **Return** $(C_1^s||C_2^s||\ldots||C_{m^s}^s||\tau^s)$

Figure 11.6: Games G0 and G1

and

$$\begin{aligned}\Pr[\mathsf{COLLR}] &= \binom{(mq+q+2)}{2}\frac{1}{2^n} \\ &= \frac{(m^2q^2+2mq^2+3mq+q^2+3q+2)}{2^{n+1}}\end{aligned}$$

Then we have

$$\begin{aligned}\Pr[\mathcal{A}^{\mathrm{G2}} \text{ sets } \mathsf{bad}] &= \Pr[\mathsf{COLLD}] + \Pr[\mathsf{COLLD}] \\ &< \frac{14m^2q^2}{2^n} \qquad (11.28)\end{aligned}$$

This completes the proof of the privacy bound.

Subroutine Ch-$\pi(X)$

01. $Y \xleftarrow{\$} \{0,1\}^n$; **if** $Y \in Range_\pi$ **then** bad $\leftarrow$ true; $\boxed{Y \xleftarrow{\$} \overline{Range_\pi}}$; **endif**;
02. **if** $X \in Domain_\pi$ **then** bad $\leftarrow$ true; $\boxed{Y \leftarrow \pi(X)}$; **endif**
03. $\pi(X) \leftarrow Y$; $Domain_\pi \leftarrow Domain_\pi \cup \{X\}$; $Range_\pi \leftarrow Range_\pi \cup \{Y\}$; **return**$(Y)$;

Initialization:

11. **for** all $X \in \{0,1\}^n$ $\pi(X) =$ undefined **endfor**
12. $EZ \xleftarrow{\$} \{0,1\}^n$; $\pi(0) = EZ$;
13. $Domain_\pi \leftarrow \{0\}$; $Range_\pi \leftarrow \{EZ\}$;
14. $EO \xleftarrow{\$} \{0,1\}^n \setminus \{EZ\}$; $\pi(1) = EO$;
15. $Domain_\pi \leftarrow Domain_\pi \cup \{1\}$; $Range_\pi \leftarrow Range_\pi \cup \{EO\}$;
16. bad $=$ false

Respond to the sth encryption query $(T^s; P_1^s||P_2^s||\ldots||P_m^s; ty^s)$ as follows:

101. $\gamma^s \leftarrow h \cdot BRW(P_1^s||P_2^s||\ldots||P_m^s||T^s)$;
102. $\tau^s \leftarrow$ Ch-$\pi(\gamma^s \oplus EZ)$;
103. **for** $i = 1$ to m,
104. $R_i^s \leftarrow$ Ch-$\pi(\tau^s \oplus x^i EO)$;
105. **if** $ty^s = L$ **then** $C_i^s \leftarrow R_i^s \oplus xP_i^s$;
106. **else** $C_i^s \leftarrow R_i^s \oplus (x \oplus 1)P_i^s$;
107. **endif**
108. **endfor**
109. **Return** $(C_1^s||C_2^s||\ldots||C_{m^s}^s||\tau^s)$

Figure 11.7: Game G2: $\mathcal{S}$ and $\mathcal{R}$ are multisets.

Proving the authentication bound: To prove the authenticity bound, let $\mathcal{A}$ be an adversary which tries to break authenticity of DCM-BRW[Π]. Let $\mathcal{B}$ be an adversary attacking authenticity of $f_{\pi,h}$ (see Lemma 10.2 for the description of $f_{\pi,h}$). $\mathcal{B}$ has an oracle access to $f_{h,\pi}(.,.)$, additionally let it have access to another oracle $\mathcal{P}(.,.)$ which on input (X,i) returns $\pi(X \oplus x^i\pi(1))$. With access to these two oracles $\mathcal{B}$ can run $\mathcal{A}$ by answering its queries in the usual manner. When $\mathcal{A}$ outputs a forgery $(T, Y_1||Y_2||\ldots||Y_m||\tau, \mathsf{ty})$, $\mathcal{B}$ computes X_i, $1 \le i \le m$, using the oracle $\mathcal{P}(.,.)$ as follows

$$X_i = \begin{cases} (\mathcal{P}(t,i) \oplus Y_i)(1 \oplus x)^{-1} & \text{if } \mathsf{ty} = \mathsf{L} \\ (\mathcal{P}(t,i) \oplus Y_i)x^{-1} & \text{if } \mathsf{ty} = \mathsf{R} \end{cases}$$

and outputs $(X_1||X_2||\ldots||X_m||T, \tau)$ as its forgery. Thus we have that

Initialization:

$\mathcal{S} \leftarrow \mathcal{R} \leftarrow \emptyset$;
$EZ \xleftarrow{\$} \{0,1\}^n$; $EO \xleftarrow{\$} \{0,1\}^n \setminus \{EZ\}$;
$\mathcal{S} \leftarrow \mathcal{S} \cup \{0,1\}$; $\mathcal{R} \leftarrow \mathcal{R} \cup \{EZ, EO\}$;

For an encryption query $(T^s; P_1^s||P_2^s||\ldots||P_{m^s}^s; ty^s)$ respond as follows:

$C_1^s||C_2^s||\ldots||C_{m^s}^s||\tau^s \xleftarrow{\$} \{0,1\}^{(m+1)n}$;
Return $C_1^s||C_2^s||\ldots||C_{m^s}^s||\tau^s$;

Finalization:

FIRST PHASE

```
for s = 1 to q,
  γ^s ← h · BRW(P_1^s||P_2^s||...||P_{m^s}^s||T^s); τ^s ←$ {0,1}^n;
  S ← S ∪ {γ^s}; R ← R ∪ {τ^s};
  for i = 1 to m,
    if ty^s = L then δ_i^s = xP_i^s
    else δ_i^s = (x+1)P_i^s;
    end if
    S ← S ∪ {τ^s ⊕ x^i EO}; R ← R ∪ {C_i^s ⊕ δ_i^s}
  end for
end for
```

SECOND PHASE

bad = false;
if (some value occurs more than once in $\mathcal{S}$) **then** bad = true **endif**;
if (some value occurs more than once in $\mathcal{R}$) **then** bad = true **endif**.

Figure 11.8: Game G2: $\mathcal{S}$ and $\mathcal{R}$ are multisets.

$$\begin{aligned}\mathbf{Adv}^{\text{dcm-auth}}_{\text{DCM-BRW[II]}}(\mathcal{A}) &= \Pr[\mathcal{A}^{\mathcal{E}_{h,\pi}(.,.,.)} \text{ forges}] \\ &\leq \Pr[\mathcal{B}^{f_{h,\pi}(.),\mathcal{P}(.,.)} \text{ forges}] \qquad (11.29)\end{aligned}$$

Now, we replace this oracle $\mathcal{P}(.,.)$ with an oracle $\$(.,.)$ which returns strings chosen uniformly at random from $\{0,1\}^n$ on a query (X, i). The difference of the real oracle $\mathcal{P}$ and the oracle $\$(.,.)$ can be detected by $\mathcal{B}$ only if the queries of $\mathcal{B}$ can produce a collision in the domain or range of the permutation π. This means that the difference can be detected by $\mathcal{B}$ if there is a collision in the multisets $\mathcal{S}$ or $\mathcal{R}$ as represented in equations (11.26) and (11.27) respectively. The event of a collision in these sets were represented by COLLD and COLLR respectively.

Thus we have

$$\Pr[\mathcal{B}^{f_{h,\pi}(.),\mathcal{P}(.,.)} \text{ forges}] - \Pr[\mathcal{B}^{f_{h,\pi}(.),\$(.,.)} \text{ forges}] \leq \Pr[\mathsf{COLLD}] + \Pr[\mathsf{COLLR}]. \quad (11.30)$$

Now, the oracle $\$(.,.)$ is of no help to $\mathcal{B}$ as the random strings that it returns can be generated by $\mathcal{B}$ itself hence,

$$\Pr[\mathcal{B}^{f_{h,\pi}(.),\$(.,.)} \text{ forges}] = \Pr[\mathcal{B}^{f_{h,\pi}(.)} \text{ forges}]. \quad (11.31)$$

Putting together equations (11.30), (11.31) and (11.28) we have

$$\begin{aligned} \Pr[\mathcal{B}^{f_{h,\pi}(.),\mathcal{P}(.,.)} \text{ forges}] &\leq \Pr[\mathcal{B}^{f_{h,\pi}(.)} \text{ forges}] + \frac{14m^2q^2}{2^n} \\ &< \frac{1}{2^n} + \frac{4mq^2}{2^n} + \frac{14m^2q^2}{2^n}. \end{aligned} \quad (11.32)$$

The last inequality follows from Lemma 10.2 and eq. 10.18 (forgery of a pseudorandom function). From eq. (11.29) and eq. (11.32) we have

$$\mathbf{Adv}^{\text{dcm-auth}}_{\text{DCM-BRW[II]}}(\mathcal{A}) \leq \frac{1}{2^n} + \frac{18m^2q^2}{2^n},$$

as desired. □

11.5 Remarks

We studied a new type of cryptographic scheme called the Double Ciphertext Mode. We gave a generic construction and a particular construction using BRW polynomials. We also explored an application where this mode can be useful.

The proposal of DCM first appeared in [25], there we posed the problem of generalizing DCM to a multi-cipher text mode where the ciphertext gets divided into more than two shares. Later in [135] DCM was generalized for multiple ciphertexts.

Chapter 12

Conclusions and Future Work

If you don't feel that this is your time yet, don't keep your appointment. Nothing is gained by forcing the issue. If you want to survive you must be crystal clear and deadly sure of yourself.

Don Juan Matus

Starting from Chapter 5 to Chapter 11 we presented original research covering several aspects of disk encryption and related problems. In each chapter we have made some final remarks where we drew our conclusion regarding the work presented in that chapter. In this chapter we again summarize our conclusions. This Chapter is divided into two sections. In Section 12.1 we present our conclusions and summarize the contributions and in Section 12.2 we list some topics and problems of immediate interest which were not considered in this thesis.

12.1 Conclusions and Summary of Contributions

The primary contribution of this thesis can be categorized into three different types: hardware implementations, new constructions of schemes for disk encryption and side channel analysis. We will summarize the principal contributions of this thesis along with the conclusions in these three parts separately.

Hardware Implementations

In Chapter 5 we presented optimized hardware implementations of six existing proposals of TES. Our choice of the schemes covers all reported "efficient" schemes. We analyzed the potential for parallelism for each of the chosen modes and discussed the achieved performance and hardware costs. We also provided experimental data regarding hardware resources and throughput. Our study confirms for the first time that many proposed modes can be efficiently used for the in-place disk encryption application. This study of performance of

TES is the first of its kind in the literature, and we hope that this performance data would help the community for comparative evaluation of TES when they get deployed widely in commercial devices in the near future.

The implementations in Chapter 5 uses as basic blocks a 10-stages pipelined AES core and a fully-parallel Karatsuba multiplier. We did a rigorous study to find opportunities to exploit parallelism in the algorithms considered. We think, that given the basic design decisions the clock cycle counts that we achieve in the reported architectures cannot be further reduced. Thus, our architectures can be considered to be highly optimal. Our target device for these architectures were Virtex 4 FPGAs but the same philosophy of design can be applied to other families of FPGAs and even for ASICs.

Based on the different experiments reported in Chapter 5, we concluded that the HEH scheme outperformed the other five TES in most scenarios except when considering what we called, the *encryption-only AES core scenario*, where EME emerged as the best of all the modes studied. On the other side of the spectrum, the XCB was the TES that consistently occupied the last place with respect to all important metrics, namely, area, time and the throughput per area. It is worthwhile to mention here that XCB is a candidate for a standard for wide block modes, our experimental data puts us in doubt regarding the basis for which XCB was chosen as a candidate for standardization.

In Chapter 6 we studied BRW polynomials from a hardware implementation perspective and designed an efficient architecture to evaluate BRW polynomials. The design of the architecture was based on a combinatorial analysis of the structural properties of BRW polynomials. Our experiments show that BRW polynomials are an efficient alternative over normal polynomials. Moreover in Chapter 7 we explored constructions of hardware architectures for tweakable enciphering schemes using BRW polynomials and the results show that using BRW polynomials are a far better alternative to normal polynomials in terms of speed for design of TES.

The implementations of TES presented in Chapters 5 and 7 were developed with the goal of obtaining the best speed and we did not consider to minimize the area (hardware resources) occupied by the implementations. Finally the implementations obtained in Chapters 5 can be used to encrypt/decrypt hard disks with 3 Gb/s SATA technology while the implementations presented in Chapter 7 can be easily used in the new 6 Gb/s SATA technology. So these implementations are suitable to encrypt/decrypt hard disks of various categories.

Side Channel Vulnerabilities

In Chapter 9 we explored the side channel vulnerabilities of EME and EME2 and we found some attacks against them assuming that *xtimes* leaks some information. These attacks do not contradict the claimed security of the modes, as the security definition and the security proofs for these modes so not assume any side-channel information being available to the adversary. Also the consequences of these attacks shown are not immediate. But,

it points out that using *xtimes* indiscriminately may give rise to security weakness in true implementations.

New Constructions

In this thesis we propose three new constructions STES (Chapter 8), BCTR (Chapter 10) and DCM (Chapter 11).

STES is a unique construction of a TES in the sense that it uses a stream cipher in place of a block cipher. The motivation behind designing STES was to design a TES which would be suitable for use in constrained devices like mobile phones. We succesfully used lightweight cryptographic primitives like stream cipher and a special types of universal hashes called multilinear universal hashes to construct STES. The security of STES was formally proved, and the obtained security bound is competitive with the security of the existing TESs. We also implemented several variants of STES using different stream ciphers and different data paths. Our experimental results show that STES can be used in constrained devices due to its small hardware footprint and its low power consumption. STES is the first proposal of lightweight disk encryption, we showed that our implementations can be useful to encrypt/decrypt almost all commercially available SD-Cards and USB pen-drives.

After a thorough review of the physical structure of a hard disk, we concluded that the size of a physical sector is larger than its storage capacity because it has extra space to enable implementation of error correction codes and also more extra space to store information necessary for the functionality of the disk. Knowing that there is extra space in the physical sector we proposed to use a non length preserving cryptographic algorithm for disk encryption. In particular we suggested the use of deterministic authenticated encryption schemes. By using such schemes we need to pay as a cost the extra space to store a tag, but among other things we gain a lot in terms of efficiency. In Chapter 10 We proposed a new DAE suitable for disk encryption application called BCTR. BCTR uses a polynomial hash based on BRW polynomials and a modified counter mode. We prove that BCTR is a secure DAE. We also show that BCTR is more efficient than any existing TES. We also implemented BCTR in hardware using the same techniques presented in Chapters 6 and 7. Our experimental results provide strong evidence that DAEs are more efficient than TES and our analysis of the physical sector shows that DAEs can be used instead of TES for disk encryption application if disks are formatted properly to accommodate cryptographic materials.

In Chapter 11 we propose a new cryptographic primitive called double ciphertext mode (DCM). DCM has the curious property that from a single plaintext it produces two related ciphertexts, and if these two ciphertexts are available then they can be used to recover the plaintext from them without the use of the key. We feel that this property of DCM can find use in the application of designing secure backup systems. We present two constructions of the DCM primitive namely DCMG and DCM-BRW, we prove security of both these constructions and argue abut their efficiency.

12.2 Future Work

Though this thesis presents comprehensive study of disk encryption schemes, there are many issues which have not been adequately treated in this work and require further investigation, we present some such issues chronologically next.

1. **Exploiting Parallelism:** The architectures presented in Chapters 5 and 7 exploits the parallelism of the algorithms to the fullest possible extent assuming the message lengths are same as the length of the sector, thus these architectures are optimal when a single sector is encrypted. In a practical application multiple sectors may be written or read at the same time from a disk. This opens up the possibility of identifying ways of parallelizing across sectors. The structure of almost all algorithms studied in this thesis would allow such parallelism and such parallelization may yield architectures which would be much more efficient than those reported here.

2. **BRW Polynomials:** In Chapter 6 we studied the structural properties of BRW polynomials in details to construct an efficient hardware to compute them. An interesting product of our study is the algorithm Schedule which gives a linear ordering of the multiplications that are to be computed to evaluate a BRW polynomial. We provided a full characterization of the behaviour of the algorithm Schedule for small values of m. Though for our and all other practical purposes this would be enough but a full characterization for arbitrary values of m may be an interesting combinatorial exercise. Such a characterization may also tell us which configurations of the collapsed forest would admit a full pipeline given a number of pipeline stages. This study can help in defining a weaker form of optimality (as mentioned in Section 7.3), which would be achievable in all cases. We provide a method for counting the number of extra storage locations for each configuration of the collapsed forest and a given number of pipeline stages. A combinatorial analysis may yield a closed form formula for counting the extra storage locations. These theoretical exercises are worth doing, and we plan to do it in near future.

3. **Real Life Deployments:** The implementations presented in this thesis are all prototypical. A real life deployment may open up new problems. A low level encryption algorithm would be a part of a hard disk, and given that the commercial hard disks available today have a closed design, hence a real deployment in an academic institution looks difficult. We plan to design an USB hub with the capability of encryption, which can be used for USB memories. Such a real implementation would involve many issues which we did not take care. One important issue which we have identified is regarding key management, and this requires special attention.

4. **Side Channel Analysis:** As we have already mentioned, the study on side channel vulnerabilities of TES presented in Chapter 9 is far from complete. We plan to design

some experiments to measure real leakages in TES implementations. In the recent days there have been some activities involving attempts to develop a theoretical model of leakages in cryptographic algorithms/implementations [51, 111]. Though these studies are in nascent form and does not say much about real life situations, but we plan to study these models with the goal of developing a TES which would be theoretically leakage resistant.

5. **Out of Model Attacks:** In designing of modes, the goal is to attain security in accordance to a strict security definition. For example in case of TES the construction should behave like a tweakable strong pseudorandom permutation under certain weak assumptions, or in case of a DAE the construction should provide security both against distinguishing attacks or forgeries. In the current days, there have been some interest in analyzing the vulnerabilities of modes against "out of model" attacks. They are called out of model as the security definition does not gives any guarantees that the modes can resist such attacks. For example, in the security model of DAE (or AE) it is guaranteed that ciphertext generated using a secure DAE can be forged only with a negligible probability using reasonable computational resources. But further one can ask, what happens if an adversary succeeds in making a forgery, some interesting questions that one may ask in this regard can be the following:

   - Will this forgery help him/her to commit more forgeries much easily?
   - Can this forgery help the adversary to recover the key?

   Of course the above list is not exhaustive. Note that in the standard security model of DAE the first forgery is very difficult to obtain, hence the above questions becomes irrelevant from a theoretical point of view. But these questions may have practical significance in some scenarios and finding answers to such questions involves analyzing the modes beyond the security model and its proof. Such analysis always helps in better evaluation of the security characteristic of a mode. Such out of model attacks for modes though have been known to the community for long in the context of various modes [53, 138], and there have been a renewed interest in such attacks as in the newly designed competition for authenticated ciphers [40], some such attacks have been pointed out as important research agenda. The study presented in this thesis does not contain any analysis of the schemes beyond the standard security model. We would like to address these issues in near future.

   Note that side channel attacks are also a class of "out of model" attacks, but the goal of side channel analysis is to analyze implementations of cryptographic algorithms for information leakages. The out of model attacks that we state here are independent of implementations.

   Regarding TES an immediate question that arises is their security characteristic in the multi user setting. In a recent work [32] security issues of message authentication

codes in the multi user settings was analyzed, we believe that these techniques can be used or extended to study other modes, and we plan to do it.

6. **Better Security Bounds:** In case of TES, all existing block cipher based constructions have a quadratic security bound of the form $c\sigma^2/2^n$, where c is a small constant. Such a bound though seems enough for the currently available computing and storage resources, but in future one may like to have schemes with better security bounds. Designing modes with better than quadratic security bounds have been an active research area and schemes with better than quadratic security bounds for authenticated encryption, message authentication codes and some other modes are known [72,73,153]. Constructing TES with better than quadratic security bound is an open research area and we wish to contribute in this direction.

7. **Other Security Notions:** There are other security notions of TES which are being currently studied. One interesting question which was raised in the IEEE SISWG was the following:

   What happens if a disk encryption scheme encrypts its own key?

   This may happen in real life when the key itself is stored in the disk and is encrypted by the disk encryption algorithm. So if an adversary has access to the encryption of the key, then can she get some extra advantage? If the knowledge of the encryption of the key does not give an adversary any extra advantage then the encryption algorithm is called key dependent message (KDM) secure. The status of KDM security for TES is mostly unknown. There have been a recent study [7] which proposes modification of EME against a class of KDM attacks, but the real insecurity of EME against such attacks is not known. We want to do a systematic study in this direction for TES.

Publications of the Author Related to the Thesis

A1. Debrup Chakraborty, **Cuauhtemoc Mancillas-López**, Francisco Rodríguez-Henríquez, Palash Sarkar,“ Hardware implementations of BRW polynomials and tweakable enciphering schemes. **IEEE Transactions on Computers**, vol.62, no.2, pp.279-294, February 2013.

A2. Debrup Chakraborty, **Cuauhtemoc Mancillas-López**, “Double ciphertext mode: a proposal for secure backup", **International Journal of Applied Cryptography** vol. 2, no. 3, pp. 271-287, 2012.

A3. **Cuauhtemoc Mancillas-López**, Debrup Chakraborty and Francisco Rodríguez-Henríquez, “Reconfigurable Hardware Implementations of Tweakable Enciphering Schemes", **IEEE Transactions on Computers**, vol. 59, no. 11, pp. 1547-1561, November 2010.

A4. **Cuauhtemoc Mancillas-López**, Debrup Chakraborty and Francisco Rodríguez-Henríquez, “On some weaknesses in the disc encryption schemes EME and EME2", **Proceedings of the International Conference on Information Systems Security**, ICISS 2009, *Lecture Notes in Computer Science* 5905, pp. 265-279, Kolkata, India, 2009.

Bibliography

[1] Reconfigurable Systems Undergo Revival. *The Economist*, 351(8120):89, 1999.

[2] IEEE P1619 Security in Storage Working Group (SISWG). IEEE Computer Society, March 2007. Available at: http://siswg.org/.

[3] P. Alfke, I. Bolsens, B. Carter, M. Santarini, and S. Trimberger. It's an FPGA!: The birth of a fabless model. *Solid-State Circuits Magazine, IEEE*, 3(4):15 –20, fall 2011.

[4] SD Association. `www.sdcard.org`.

[5] Steve Babbage and Matthew Dodd. The MICKEY Stream Ciphers. In Robshaw and Billet [115], pages 191–209.

[6] Donald G. Bailey. *Design for Embedded Image Processing on FPGAs*. John Wiley & Sons, Ltd, 2011.

[7] Mihir Bellare, David Cash, and Sriram Keelveedhi. Ciphers that securely encipher their own keys. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 423–432. ACM, 2011.

[8] Mihir Bellare and Phillip Rogaway. The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs. In Vaudenay [139], pages 409–426.

[9] Mihir Bellare, Phillip Rogaway, and David Wagner. The EAX Mode of Operation. In *FSE*, volume 3017 of *Lecture Notes in Computer Science*, pages 389–407. Springer, 2004.

[10] Daniel J. Bernstein. Polynomial Evaluation and Message Authentication, 2007. `http://cr.yp.to/papers.html\#pema`.

[11] Jean-Luc Beuchat, Jérémie Detrey, Nicolas Estibals, Eiji Okamoto, and Francisco Rodríguez-Henríquez. Fast Architectures for the η_T Pairing over Small-Characteristic Supersingular Elliptic Curves. *Computers, IEEE Transactions on*, 60(2):266 –281, feb. 2011.

[12] Ramesh Karri Bo Yang, Sambit Mishra. A High Speed Architecture for Galois/Counter Mode of Operation (GCM). Cryptology ePrint Archive, Report 2005/146, 2005. `http://eprint.iacr.org/`.

[13] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

[14] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In *EUROCRYPT*, pages 37–51, 1997.

[15] S. Brown. FPGA Architectural Research: A Survey. *Design Test of Computers, IEEE*, 13(4):9 –15, winter 1996.

[16] Philippe Bulens, Kassem Kalach, FranÃğois-Xavier Standaert, and Jean-Jacques Quisquater. FPGA Implementations of eSTREAM Phase-2 Focus Candidates with Hardware Profile. 1 2007. http://sasc.cry.

[17] Philippe Bulens, François-Xavier Standaert, Jean-Jacques Quisquater, Pascal Pellegrin, and Gaël Rouvroy. Implementation of the AES-128 on Virtex-5 FPGAs. In Vaudenay [140], pages 16–26.

[18] Sanjay Burman, Debdeep Mukhopadhyay, and Kamakoti Veezhinathan. LFSR Based Stream Ciphers Are Vulnerable to Power Attacks. In Srinathan et al. [134], pages 384–392.

[19] Christophe De Cannière, Orr Dunkelman, and Miroslav Knezevic. Katan and ktantan - a family of small and efficient hardware-oriented block ciphers. In Clavier and Gaj [37], pages 272–288.

[20] Christophe De Cannière, Orr Dunkelman, and Miroslav Knezevic. KATAN and KTANTAN - a family of small and efficient hardware-oriented block ciphers. In Clavier and Gaj [37], pages 272–288.

[21] Christophe De Cannière and Bart Preneel. Trivium. In Robshaw and Billet [115], pages 244–266.

[22] Anne Canteaut and Kapalee Viswanathan, editors. *Progress in Cryptology - INDOCRYPT 2004, 5th International Conference on Cryptology in India, Chennai, India, December 20-22, 2004, Proceedings*, volume 3348 of *Lecture Notes in Computer Science*. Springer, 2004.

[23] D.G. Cantor, G. Estrin, and R. Turn. Logarithmic and Exponential Function Evaluations in a Variable Structure Digital Computer, journal = IRE Trans. Electronic Computers. EC- 11:155–164, 1962.

[24] Larry Carter and Mark N. Wegman. Universal Classes of Hash Functions (Extended Abstract). In John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison, editors, *STOC*, pages 106–112. ACM, 1977.

[25] Debrup Chakraborty and Cuauhtemoc Mancillas-López. Double ciphertext mode : A proposal for secure backup. *IACR Cryptology ePrint Archive*, 2010:369, 2010.

[26] Debrup Chakraborty and Cuauhtemoc Mancillas-López. Double ciphertext mode: a proposal for secure backup. *IJACT*, 2(3):271–287, 2012.

[27] Debrup Chakraborty, Cuauhtemoc Mancillas-López, Francisco Rodriguez-Henriquez, and Palash Sarkar. Efficient hardware implementations of brw polynomials and tweakable enciphering schemes. *IEEE Transactions on Computers (to appear)*, 2013. Available as IACR ePrint report 2011/161.

[28] Debrup Chakraborty and Mridul Nandi. An Improved Security Bound for HCTR. In Kaisa Nyberg, editor, *FSE*, volume 5086 of *Lecture Notes in Computer Science*, pages 289–302. Springer, 2008.

[29] Debrup Chakraborty and Palash Sarkar. A New Mode of Encryption Providing a Tweakable Strong Pseudo-random Permutation. In Robshaw [114], pages 293–309.

[30] Debrup Chakraborty and Palash Sarkar. A General Construction of Tweakable Block Ciphers and Different Modes of Operations. *IEEE Transactions on Information Theory*, 54(5):1991–2006, 2008.

[31] Debrup Chakraborty and Palash Sarkar. HCH: A New Tweakable Enciphering Scheme Using the Hash-Counter-Hash Approach. *IEEE Transactions on Information Theory*, 54(4):1683–1699, 2008.

[32] Sanjit Chatterjee, Alfred Menezes, and Palash Sarkar. Another look at tightness. In Ali Miri and Serge Vaudenay, editors, *Selected Areas in Cryptography*, volume 7118 of *Lecture Notes in Computer Science*, pages 293–319. Springer, 2011.

[33] Hao Chen, Yu Chen, and D.H. Summerville. A Survey on the Application of FPGAs for Network Infrastructure Security. *Communications Surveys Tutorials, IEEE*, 13(4):541 –561, quarter 2011.

[34] P. Chicoine, M. Hassner, M. Noblitt, G. Silvus, B. Weber, and E. Grochowski. Hard Disk Drive Long Data Sector White Paper. The International Disk Drive Equipments and Materials Association, 2007. `http://www.idema.org/wp-content/plugins/download-monitor/download.php?id=1184`.

[35] P. Chow, Soon Ong Seo, J. Rose, K. Chung, G. Paez-Monzon, and I. Rahardja. The Design of an SRAM-Based Field-Programmable Gate Array. I. Architecture. *IEEE Trans. VLSI Syst.*, 7(2):191–197, 1999.

[36] C. Claus, W. Stechele, M. Kovatsch, J. Angermeier, and J. Teich. A Comparison of Embedded Reconfigurable Video-Processing Architectures. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 587 –590, sept. 2008.

[37] Christophe Clavier and Kris Gaj, editors. *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*. Springer, 2009.

[38] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[39] Zhibin Dai and Dilip K. Banerji. Routability Prediction for Field Programmable Gate Arrays with a Routing Hierarchy. In *VLSI Design*, pages 85–90. IEEE Computer Society, 2003.

[40] Tanja Lange Daniel J. Bernstein. Authenticated ciphers, 2012. `http://cr.yp.to/talks/2012.01.16/slides.pdf`.

[41] Doug Whiting, Russ Housley, Niels Ferguson. Counter with CBC-MAC (CCM). In *Submission to NIST*, 2002.

[42] Dworkin, Morris J. SP 800-38E. Recommendation for Block Cipher Modes of Operation: the XTS-AES Mode for Confidentiality on Storage Devices. Technical report, Gaithersburg, MD, United States, 2010.

[43] Gerald Estrin. Organization of Computer Systems-the Fixed Plus Variable Structure Computer. *Managing Requirements Knowledge, International Workshop on*, 0:33, 1960.

[44] Gerald Estrin. Reconfigurable Computer Origins: The UCLA Fixed-Plus-Variable (F+V) Structure Computer. *IEEE Annals of the History of Computing*, 24(4):3–9, 2002.

[45] Gerald Estrin and C. R. Viswanathan. Organization of a “Fixed-Plus-Variable” Structure Computer for Computation of Eigenvalues and Eigenvectors of Real Symmetric Matrices. *J. ACM*, 9(1):41–60, 1962.

[46] Hongbing Fan, Jiping Liu, Yu-Liang Wu, and Chak-Chung Cheung. On Optimum Switch Box Designs for 2-D FPGAs. In *DAC*, pages 203–208. ACM, 2001.

[47] Hongbing Fan, Jiping Liu, Yu-Liang Wu, and Chak-Chung Cheung. On Optimum Designs of Universal Switch Blocks. In Manfred Glesner, Peter Zipf, and Michel Renovell, editors, *FPL*, volume 2438 of *Lecture Notes in Computer Science*, pages 142–151. Springer, 2002.

[48] Hongbing Fan, Jiping Liu, Yu-Liang Wu, and Chak-Chung Cheung. On Optimal Hyperuniversal and Rearrangeable Switch Box Designs. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 22(12):1637–1649, 2003.

[49] Hongbing Fan, Jiping Liu, Yu-Liang Wu, and Chak-Chung Cheung. The Exact Channel Density and Compound Design for Generic Universal Switch Blocks. *ACM Trans. Design Autom. Electr. Syst.*, 12(2), 2007.

[50] Hongbing Fan, Yu-Liang Wu, and Chak-Chung Cheung. Design Automation for Reconfigurable Interconnection Networks. In Phaophak Sirisuk, Fearghal Morgan, Tarek A. El-Ghazawi, and Hideharu Amano, editors, *ARC*, volume 5992 of *Lecture Notes in Computer Science*, pages 244–256. Springer, 2010.

[51] Sebastian Faust, Krzysztof Pietrzak, and Joachim Schipper. Practical leakage-resilient symmetric cryptography. In Emmanuel Prouff and Patrick Schaumont, editors, *CHES*, volume 7428 of *Lecture Notes in Computer Science*, pages 213–232. Springer, 2012.

[52] Martin Feldhofer. Comparison of low-power implementations of trivium and grain. In *The State of the Art of Stream Ciphers, Workshop Record*, pages 236 – 246, 2007.

[53] Neils Ferguson. Authentication weaknesses in GCM. National Institute of Standards and Technologies (NIST), 2005. `http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/CWC-GCM/Ferguson2.pdf`.

[54] Niels Ferguson. AES-CBC + Elephant diffuser: A Disk Encryption Algorithm for Windows Vista. Microsoft white paper, 2006. `http://download.microsoft.com/download/0/2/3/0238acaf-d3bf-4a6d-b3d6-0a0be4bbb36e/BitLockerCipher200608.pdf`.

[55] Y. Fu, L. Hao, and X. Zhang. Design of an Extremely High Performance Counter Mode AES Reconfigurable Processor. In *Proceedings of the Second International Conference on Embedded Software and Systems (ICESS'05)*, pages 262–268. IEEE Computer Society, 2005.

[56] Kris Gaj and Pawel Chodowiec. FPGA and ASIC Implementations of AES. In Cetin Kaya Koc, editor, *Cryptographic Engineering*, pages 235–294. Springer, 2009.

[57] Andreas Gerstlauer, Christian Haubelt, Andy D. Pimentel, Todor Stefanov, Daniel D. Gajski, and Jürgen Teich. Electronic System-Level Synthesis Methodologies. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 28(10):1517–1530, 2009.

[58] Zheng Gong, Svetla Nikova, and Yee Wei Law. KLEIN: A new family of lightweight block ciphers. In Ari Juels and Christof Paar, editors, *RFIDSec*, volume 7055 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2011.

[59] T. Good and M. Benaissa. Hardware Results for Selected Stream Cipher Candidates. In *The State of the Art of Stream Ciphers, Workshop Record*, pages 191–204, 2007.

[60] Tim Good and Mohammed Benaissa. AES on FPGA from the Fastest to the Smallest. In Josyula R. Rao and Berk Sunar, editors, *CHES*, volume 3659 of *Lecture Notes in Computer Science*, pages 427–440. Springer, 2005.

[61] Jian Guo, Thomas Peyrin, and Axel Poschmann. The photon family of lightweight hash functions. In Phillip Rogaway, editor, *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 222–239. Springer, 2011.

[62] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw. The LED block cipher. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES*, volume 6917 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2011.

[63] Shai Halevi. EME*: Extending EME to Handle Arbitrary-Length Messages with Associated Data. In Canteaut and Viswanathan [22], pages 315–327.

[64] Shai Halevi. Invertible Universal Hashing and the TET Encryption Mode. In Alfred Menezes, editor, *CRYPTO*, volume 4622 of *Lecture Notes in Computer Science*, pages 412–429. Springer, 2007.

[65] Shai Halevi and Phillip Rogaway. A Tweakable Enciphering Mode. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 482–499. Springer, 2003.

[66] Shai Halevi and Phillip Rogaway. A Parallelizable Enciphering Mode. In Tatsuaki Okamoto, editor, *CT-RSA*, volume 2964 of *Lecture Notes in Computer Science*, pages 292–304. Springer, 2004.

[67] Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. The Grain Family of Stream Ciphers. In Robshaw and Billet [115], pages 179–190.

[68] I.N. Hersten. *Topics in Algebra*. Wiley India Pvt. Limited, 2006.

[69] S. F. Hsiao and M. C. Chen. Efficient Substructure Sharing Methods for Optimising the Inner-Product Operations in Rijndael Advanced Encryption Standard. *IEE Proceedings on Computer and Digital Technology*, 152(5):653–665, September 2005.

[70] David Hwang, Mark Chaney, Shashi Karanam, Nick Ton, and Kris Gaj. Comparison of FPGA-targeted hardware implementations of eSTREAM stream cipher candidates. In *State of the Art of Stream Ciphers Workshop, SASC 2008, Lausanne, Switzerland*, pages 151–162, Feb 2008.

[71] IEEE Security in Storage Working Group (SISWG). PRP Modes Comparison, November 2007. `http://siswg.org/`. IEEE p1619.2.

[72] Tetsu Iwata. New blockcipher modes of operation with beyond the birthday bound security. In Robshaw [114], pages 310–327.

[73] Tetsu Iwata. Authenticated encryption mode for beyond the birthday bound security. In Vaudenay [140], pages 125–142.

[74] Tetsu Iwata and Kan Yasuda. BTM: A Single-Key, Inverse-Cipher-Free Mode for Deterministic Authenticated Encryption. In Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography*, volume 5867 of *Lecture Notes in Computer Science*, pages 313–330. Springer, 2009.

[75] Tetsu Iwata and Kan Yasuda. HBS: A Single-Key Mode of Operation for Deterministic Authenticated Encryption. In Orr Dunkelman, editor, *FSE*, volume 5665 of *Lecture Notes in Computer Science*, pages 394–415. Springer, 2009.

[76] K. Järvinen, M. Tommiska, and J. Skyttä. Comparative Survey of High-Performance Cryptographic Algorithm Implementations on FPGAs. *Information Security, IEE Proceedings*, 152(1):3–12, October 2005.

[77] Antoine Joux and Pascal Delaunay. Galois LFSR, Embedded Devices and Side Channel Weaknesses. In Rana Barua and Tanja Lange, editors, *INDOCRYPT*, volume 4329 of *Lecture Notes in Computer Science*, pages 436–451. Springer, 2006.

[78] Marc Joye. Basics of Side-Channel Analysis. In Çetin Kaya Koç, editor, *Cryptographic Engineering*, pages 365–380. Springer US, 2009. 10.1007/978-0-387-71817-0_13.

[79] Charanjit S. Jutla. Encryption Modes with Almost Free Message Integrity. *J. Cryptology*, 21(4):547–578, 2008.

[80] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/ CRC, 2008.

[81] Lars R. Knudsen and Matthew Robshaw. *The Block Cipher Companion*. Information security and cryptography. Springer, 2011.

[82] Neal Koblitz, editor. *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*. Springer, 1996.

[83] Neal Koblitz and Alfred Menezes. Another Look at "Provable Security". II. *IACR Cryptology ePrint Archive*, 2006:229, 2006.

[84] Neal Koblitz and Alfred Menezes. Another Look at "Provable Security". *J. Cryptology*, 20(1):3–37, 2007.

[85] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Koblitz [82], pages 104–113.

[86] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Michael J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[87] Ted Krovetz and Phillip Rogaway. The software performance of authenticated-encryption modes. In Antoine Joux, editor, *FSE*, volume 6733 of *Lecture Notes in Computer Science*, pages 306–327. Springer, 2011.

[88] Yen-Tai Lai and Ping-Tsung Wang. Hierarchical Interconnection Structures for Field Programmable Gate Arrays. *IEEE Trans. VLSI Syst.*, 5(2):186–196, 1997.

[89] Lattice, Inc. *iCE40 Family Handbook Ultra Low-Power mobile FPGA LP, HX*, March 2012.

[90] Pil Joong Lee, editor. *Advances in Cryptology - ASIACRYPT 2004, 10th International Conference on the Theory and Application of Cryptology and Information Security, Jeju Island, Korea, December 5-9, 2004, Proceedings*, volume 3329 of *Lecture Notes in Computer Science*. Springer, 2004.

[91] Rudolf Lidl and Harald Niederreiter. *Introduction to Finite Fields and their Applications*. Cambridge University Press, 1984.

[92] Moses Liskov and Kazuhiko Minematsu. Comments on XTS-AES. Comments On The Proposal To Approve XTS-AES. Technical report, 2008.

[93] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable Block Ciphers. In Yung [154], pages 31–46.

[94] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable Block Ciphers. *J. Cryptology*, 24(3):588–613, 2011.

[95] Yiyuan Luo, Qi Chai, Guang Gong, and Xuejia Lai. A lightweight stream cipher wg-7 for rfid encryption and authentication. In *GLOBECOM*, pages 1–6. IEEE, 2010.

[96] Cuauhtemoc Mancillas-López, Debrup Chakraborty, and Francisco Rodríguez-Henríquez. On some weaknesses in the disk encryption schemes eme and eme2. In

Atul Prakash and Indranil Gupta, editors, *ICISS*, volume 5905 of *Lecture Notes in Computer Science*, pages 265–279. Springer, 2009.

[97] Cuauhtemoc Mancillas-López, Debrup Chakraborty, and Francisco Rodríguez-Henríquez. Reconfigurable hardware implementations of tweakable enciphering schemes. *IEEE Trans. Computers*, 59(11):1547–1561, 2010.

[98] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks - Revealing the Secrets of Smart Cards*. Springer, 2007.

[99] D. McGrew and J. Viega. The Galois/Counter Mode of Operation (GCM), Submission to NIST Modes of Operation Process, January 2004. Available at: http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-revised-spec.pdf.

[100] David A. McGrew and Scott R. Fluhrer. Arbitrary Block Length Mode, 2004.

[101] David A. McGrew and Scott R. Fluhrer. The Extended Codebook (XCB) Mode of Operation. Cryptology ePrint Archive, Report 2004/278, 2004. `http://eprint.iacr.org/`.

[102] David A. McGrew and Scott R. Fluhrer. The Security of the Extended Codebook (XCB) Mode of Operation. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *Selected Areas in Cryptography*, volume 4876 of *Lecture Notes in Computer Science*, pages 311–327. Springer, 2007.

[103] David A. McGrew and John Viega. The Security and Performance of the Galois/Counter Mode (GCM) of Operation. In Canteaut and Viswanathan [22], pages 343–355.

[104] Kazuhiko Minematsu. Improved security analysis of XEX and LRW modes. In Eli Biham and Amr M. Youssef, editors, *Selected Areas in Cryptography*, volume 4356 of *Lecture Notes in Computer Science*, pages 96–113. Springer, 2006.

[105] Kazuhiko Minematsu and Toshiyasu Matsushima. Tweakable Enciphering Schemes from Hash-Sum-Expansion. In Srinathan et al. [134], pages 252–267.

[106] E. Monmasson, L. Idkhajine, M.N. Cirstea, I. Bahri, A. Tisan, and M.W. Naouar. FPGAs in Industrial Control Applications. *Industrial Informatics, IEEE Transactions on*, 7(2):224–243, may 2011.

[107] Moni Naor and Omer Reingold. A Pseudo-Random Encryption Mode. Technical report, UNPUBLISHED, 1997. http://www.wisdom.weizmann.ac.il/ naor/PAPERS/nr-mode.ps.gz.

[108] National Institute of Standards and Technology. *FIPS PUB 46-3: Data Encryption Standard (DES)*. pub-NIST, pub-NIST:adr, oct 1999. supersedes FIPS 46-2.

[109] Z. Navabi. *VHDL:Modular Design and Synthesis of Cores and Systems, Third Edition*. McGraw-Hill, 2007.

[110] Raphael Chung-Wei Phan and Bok-Min Goi. On the Security Bounds of CMC, EME, EME^+ and EME^* Modes of Operation. In Sihan Qing, Wenbo Mao, Javier Lopez, and Guilin Wang, editors, *ICICS*, volume 3783 of *Lecture Notes in Computer Science*, pages 136–146. Springer, 2005.

[111] Krzysztof Pietrzak. A leakage-resilient mode of operation. In Antoine Joux, editor, *EUROCRYPT*, volume 5479 of *Lecture Notes in Computer Science*, pages 462–482. Springer, 2009.

[112] Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In Isabelle Attali and Thomas P. Jensen, editors, *E-smart*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001.

[113] Michael O. Rabin and Shmuel Winograd. Fast Evaluation of Polynomials by Rational Preparation. *Communications on Pure and Applied Mathematics*, 25:433–458, 1972.

[114] Matthew J. B. Robshaw, editor. *Fast Software Encryption, 13th International Workshop, FSE 2006, Graz, Austria, March 15-17, 2006, Revised Selected Papers*, volume 4047 of *Lecture Notes in Computer Science*. Springer, 2006.

[115] Matthew J. B. Robshaw and Olivier Billet, editors. *New Stream Cipher Designs - The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*. Springer, 2008.

[116] F. Rodríguez-Henríquez and Ç. K. Koç. On Fully Parallel Karatsuba Multipliers for $\text{GF}(2^m)$. In *International Conference on Computer Science and Technology CST 2003, May 19-21 2003, Cancún, México*, Lecture Notes in Computer Science, pages 405–410. Acta Press, May 2003.

[117] Rodríguez-Henríquez, Francisco and Saqib, N. A. and Díaz-Pèrez, A. and Koc, Cetin Kaya. *Cryptographic Algorithms on Reconfigurable Hardware (Signals and Communication Technology)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[118] Phillip Rogaway. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In Lee [90], pages 16–31.

[119] Phillip Rogaway and Thomas Shrimpton. A Provable-Security Treatment of the Key-Wrap Problem. In Vaudenay [139], pages 373–390.

[120] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli. Architecture of Field-Programmable Gate Arrays. *Proceedings of the IEEE*, 81(7):1013 –1029, jul 1993.

[121] Gaël Rouvroy, François-Xavier Standaert, Jean-Jacques Quisquater, and Jean-Didier Legat. Compact and efficient encryption/decryption module for fpga implementation of the aes rijndael very well suited for small embedded applications. In *ITCC (2)*, pages 583–587. IEEE Computer Society, 2004.

[122] Palash Sarkar. Improving Upon the TET Mode of Operation. In Kil-Hyun Nam and Gwangsoo Rhee, editors, *ICISC*, volume 4817 of *Lecture Notes in Computer Science*, pages 180–192. Springer, 2007.

[123] Palash Sarkar. A General Mixing Strategy for the ECB-Mix-ECB Mode of Operation. *Inf. Process. Lett.*, 109(2):121–123, 2008.

[124] Palash Sarkar. A New Multi-Linear Universal Hash Family. *IACR Cryptology ePrint Archive*, 2008:216, 2008.

[125] Palash Sarkar. Efficient Tweakable Enciphering Schemes from (Block-Wise) Universal Hash Functions. *IEEE Transactions on Information Theory.*, 55(10):4749–4760, 2009.

[126] Palash Sarkar. Pseudo-Random Functions and Parallelizable Modes of Operations of a Block Cipher. Cryptology ePrint Archive, Report 2009/217, 2009. `http://eprint.iacr.org/`.

[127] Palash Sarkar. Tweakable Enciphering Schemes From Stream Ciphers With IV. Cryptology ePrint Archive, Report 2009/321, 2009. `http://eprint.iacr.org/`.

[128] Palash Sarkar. Tweakable Enciphering Schemes Using Only the Encryption Function of a Block Cipher. *Inf. Process. Lett.*, 111(19):945–955, 2011.

[129] Akashi Satoh, Takeshi Sugawara, and Takafumi Aoki. High-Performance Hardware Architectures for Galois Counter Mode. *IEEE Trans. Computers*, 58(7):917–930, 2009.

[130] Seagate Technology. Internal 3.5-inch (SATA) Data Sheet. Available at:http://www.seagate.com/docs/pdf/datasheet/disc/ds_internal_sata.pdf.

[131] Security in Storage Workgroup of the IEEE Computer Society. Draft standard architecture for wide-block encryption for shared storage media. Institute of Electrical and Electronics Engineers, 2008. `http://siswg.net/index2.php?option=com_docman&task=doc_view&gid=84&Itemid=41`.

[132] Victor Shoup. On Fast and Provably Secure Message Authentication Based on Universal Hashing. In Koblitz [82], pages 313–328.

[133] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. *IACR Cryptology ePrint Archive*, 2004:332, 2004.

[134] K. Srinathan, C. Pandu Rangan, and Moti Yung, editors. *Progress in Cryptology - INDOCRYPT 2007, 8th International Conference on Cryptology in India, Chennai, India, December 9-13, 2007, Proceedings*, volume 4859 of *Lecture Notes in Computer Science*. Springer, 2007.

[135] Martin Stanek. Threshold Encryption into Multiple Ciphertexts. In Joaquín García-Alfaro and Pascal Lafourcade, editors, *FPS*, volume 6888 of *Lecture Notes in Computer Science*, pages 62–72. Springer, 2011.

[136] Seagate Technology. Comments on XTS-AES. Comments On The Proposal To Approve XTS-AES. Technical report, 2008.

[137] Texas Instruments. *Low-Power High-Performance Impact TM PAL Circuits*, December 2010.

[138] Serge Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ... In Lars R. Knudsen, editor, *EUROCRYPT*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–546. Springer, 2002.

[139] Serge Vaudenay, editor. *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, volume 4004 of *Lecture Notes in Computer Science*. Springer, 2006.

[140] Serge Vaudenay, editor. *Progress in Cryptology - AFRICACRYPT 2008, First International Conference on Cryptology in Africa, Casablanca, Morocco, June 11-14, 2008. Proceedings*, volume 5023 of *Lecture Notes in Computer Science*. Springer, 2008.

[141] Peng Wang, Dengguo Feng, and Wenling Wu. HCTR: A Variable-Input-Length Enciphering Mode. In Dengguo Feng, Dongdai Lin, and Moti Yung, editors, *CISC*, volume 3822 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2005.

[142] Xilinx, Inc. *Achieving Higher System Performance with the Virtex-5 Family of FPGAs*, July 2006.

[143] Xilinx, Inc. *CoolRunner-II CPLD Family*, September 2008.

[144] Xilinx, Inc. *Synthesis and Simulation Design Guide* , 2008.

[145] Xilinx, Inc. *Spartan-3 FPGA Family Data Sheet*, December 2009.

[146] Xilinx, Inc. *Virtex-5 Family Overview*, February 2009.

[147] Xilinx, Inc. *Virtex-4 Family Overview*, August 2010.

[148] Xilinx, Inc. *Spartan3 Generation FPGA User Guide*, June 2011.

[149] Xilinx, Inc. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, June 2011.

[150] Xilinx, Inc. *Virtex-5 FPGA User Guide*, March 2012.

[151] Xilinx, Inc. *Virtex-5 FPGA XtremeDSP Design Considerations*, January 2012.

[152] Xilinx, Inc. *Virtex-6 Family Overview*, January 2012.

[153] Kan Yasuda. A parallelizable PRF-based MAC algorithm: Well beyond the birthday bound. *IEICE Transactions*, 96-A(1):237–241, 2013.

[154] Moti Yung, editor. *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, volume 2442 of *Lecture Notes in Computer Science*. Springer, 2002.