Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

UNIDAD ZACATENCO
Departamento de Computación

Implementaciones Eficientes en Software de Esquemas de Cifrado de Discos usando AES-NI

Tesis que presenta

Nallely Itzel Guadalupe Trejo García

para obtener el Grado de

Maestro en Ciencias

en la Especialidad de

Computación

Director de Tesis:

Dr. Debrup Chakraborty

México D.F. **Febrero, 2012**

Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

UNIDAD ZACATENCO
Departamento de Computación

Efficient Software Implementations of Disk Encryption Schemes Using AES-NI Support

by

Nallely Itzel Guadalupe Trejo García

Thesis Advisor:

Dr. Debrup Chakraborty

México D.F. **February, 2012**

Acknowledgements

This thesis would not have been possible without the guidance and the help of several individuals who in one way or another contributed and extended their valuable assistance in the preparation and completion of this work.

First of all I would like to express my gratitude to Dr. Debrup Chakraborty for being an outstanding advisor and an excellent professor. His constant encouragement, support, and suggestions made this work succeded. I would like to thank also to my revisors for their time and effort in reviewing this document and their comments.

A very special thanks goes out to Prof. Palash Sarkar who adviced us to include other disk encryption schemes: XTS and BitLocker, and also adviced us to look for a concrete attack on XTS.

I would also acknowledge CONACYT which enabled me to undertake a master program at CINVESTAV. I am also grateful to this institution. In particular to our department secretaries, specially Sofy, whose assistance helped me along the way.

I wish to thank to all my colleagues and friends for helping me get through the difficult times, and for all the emotional support, entertainment, and caring they provided. I will never forget all the time we shared.

Finally, I would like to thank the people to whom I owe the most, my family. Specially I want to thank to the Hernandez-Trejo who provided me a home, far from home. And most importantly to my parents, Rogelio and Lupita and my sister, Jimena, which have given me their unequivocal support throughout, as always, for which my mere expression of thanks likewise does not suffice.

Resumen

Los TES (Tweakable Enciphering Schemes) son una clase de modos de operación que se ajustan al cifrado de información en dispositivos de almacenamiento como discos duros y memorias *flash*. A la fecha hay aproximadamente diez propuestas diferentes para TES y existe un esfuerzo de estandarización por parte del grupo de trabajo de seguridad en almacenamiento del IEEE. En los últimos años ha habido actividad considerable en el diseño de estos esquemas, sin embargo la cantidad de información acerca del desempeño experimental reportada en la literatura es escasa. Existe información acerca del desempeño en hardware, pero no hay información disponible de ello en software. Esta tesis se enfoca en esa brecha.

En esta tesis presentamos implementaciones optimizadas en software de (casi) todos los TES y comparamos su desempeño en terminos de velocidad. Nuestras implementaciones utilizan el nuevo conjunto de instrucciones AES-NI que se encuentran disponibles en algunos de los procesadores Intel y AMD actuales. También presentamos implementaciones de los esquemas en paralelo enfocadas a procesadores múlti-nucleo.

Nuestro estudio de desempeño también incluye dos esquemas que no son TES pero han sido propuestos para su uso en cifrado de discos: XTS es un esquema que recientemente fue estandarizado por el NIST para ser usado como algoritmo de cifrado de información en dispositivos de almacenamiento que guardan la información en bloques; y el esquema llamado BitLocker que ha sido usado ampliamente como algoritmo de cifrado de discos en el sistema operativo de Windows Vista. Presentamos información de desempeño tanto para XTS como para BitLocker y describimos una debilidad de seguridad que se encuentra en XTS.

Finalmente, se proponen dos nuevos TES llamados HCTR$*$ y HMCH2 los cuales son modificaciones de dos esquemas ya existentes. Nuestros datos experimentales sugieren que tanto HCTR$*$ como HMCH2 son más eficientes que sus predecesores, además se demuestra que ambos esquemas son seguros.

Abstract

Tweakable enciphering schemes (TES) are a class of block cipher modes of operation which are suitable for encryption of block oriented storage media like hard disks, flash memories etc. To date there are about ten different proposals for TES, and there is an active standardization effort by the IEEE working group on security in storage. In the last few years there have been an considerable activity involving designing of such schemes, but there are very little experimental performance data of these schemes reported in the literature. There are some performance data in hardware available, but no software performance data has yet been reported. This thesis closes this important gap.

In this thesis we present optimized implementations of (almost) all tweakable enciphering schemes in software and compare their performance in terms of speed. Our implementations utilizes a new instruction set extension called AES-NI which is available in some current Intel and AMD processors. We also present parallel implementations of the schemes directed towards multi-core processors.

Our performance study also includes two schemes which are not TES but has been proposed for disk encryption. XTS is a scheme which has recently been standardized by NIST for use as an encryption algorithm for block oriented storage devices, and another scheme called BitLocker has seen wide deployment as a disk encryption algorithm in the Windows Vista operating system. We present performance data for both XTS and BitLocker and also point out a security weakness in XTS.

Furthermore, we propose two new tweakable enciphering schemes named HCTR* and HMCH2 which are modifications of two existing schemes. The modifications were done by keeping an eye on the efficiency. Our experimental results suggests that both HCTR* and HMCH2 are more efficient than their predecessors. Through standard reductionist arguments we also prove that HCTR* and HMCH2 are secure TES.

Contents

1 Introduction **1**

1.1 Some Notions of Disk Encryption 2

1.2 Contribution and Scope of the Thesis 3

2 Exploiting Parallelism in Modern Processors **7**

2.1 Instruction Set Architecture 7

2.2 Instruction Level Parallelism 8

2.3 Data Level Parallelism and SIMD Instructions 11

2.3.1 SIMD instructions in modern processors 12

2.4 Multicore Processors and Thread Level Parallelism 15

2.5 Notes on Programming Languages and Tools 15

2.5.1 Basics of parallel programming 17

3 Schemes for Disk Encryption **23**

3.1 Activities of IEEE SISWG 23

3.2 Some Notations 25

3.3 Tweakable Enciphering Schemes 26

3.3.1 A brief history of the known constructions of TES 27

3.3.2 The schemes considered for this study 28

3.3.3 Description of some schemes 29

3.4 Other Schemes 34

3.4.1 Wide Block Block Ciphers 34

3.4.2 XTS 34

3.4.3 LRW 37

3.4.4 BitLocker 38

4 Two new Tweakable Enciphering Schemes **41**
4.1 Description of HCTR* and HMCH2 41
4.2 Security of the constructions 43
4.2.1 Definitions and notation 43
4.2.2 Statement of the results 45
4.2.3 Proofs 46

5 Implementing the Basic Building Blocks **57**
5.1 Binary Field Operations 57
5.1.1 Multiplication 58
5.1.2 Xtimes 59
5.1.3 Squaring 61
5.2 The Advanced Encryption Standard (AES) 62
5.2.1 Intel AES-NI architecture 66

6 Experimental Results **71**
6.1 Basic Implementation Strategies 71
6.2 System Information 73
6.3 Testing methodology 73
6.4 Results 75
6.4.1 Comparisons 81

7 Conclusion **89**
7.1 Future Work 92

A Source Codes of Some Basic Blocks **103**

B Experimental Results **105**

List of Algorithms

3.1 Encryption using EME2 30
3.2 Encryption using XCB 31
3.3 Encryption using HCTR 32
3.4 Encryption using HEH 33
3.5 Encryption using HMCH 33
3.6 Encryption and Decryption XTS 35
3.7 Encryption LRW 38
3.8 Encryption BitLocker 39

4.9 Encryption and Decryption using HCTR* 42
4.10 Encryption and Decryption using HMCH2 43

5.11 Karatsuba Multiplier 59
5.12 Fast Reduction modulo $q(x)$ 60
5.13 *xtimes* 61
5.14 Squaring in $\mathbb{F}_{2^{128}}$ 62

List of Tables

3.1 List of existing TES 29

6.1 Total clock cycles for computing the basic binary field operations over $\mathbb{F}_{2^{128}}$ in M2 using the ICC compiler. 75

6.2 Counter mode variants for a 4KB buffer using ICC compiler in machine M2. 75

6.3 Encryption and decryption implementation results **in clock cycles per byte** (ccpb), for a **4KB** buffer and a **128-bit tweak** using both machines M1 and M2. (a) includes Key Expansion procedure and (b) does not include it. 77

6.4 Encryption and decryption performance results of other disk encryption schemes: XTS, BitLocker **in clock cycles per byte** (ccpb), for a **4KB** buffer using both machines M1 and M2. 77

6.5 Hardware implementation results presented in [10, 47] which show the number of clock cycles and its corresponding throughput for various TES constructions to encrypt a whole disk sector of 512 bytes. 82

6.6 Encryption implementation throughput **GBit/Sec**, for a **4KB** buffer and a **128-bit tweak** using machine M2 and ICC compiler, including Key Expansion procedure 82

6.7 Encryption implementation throughput **GBit/Sec**, for a **512-bytes** buffer and a **128-bit tweak** using machine M2 and ICC compiler, including Key Expansion procedure 83

6.8 Parallel encryption and decryption implementation results with **2 threads in clock cycles per byte** (ccpb), for (a) **200** and (b) **1000** sectors of **4KB** and a **128-bit tweak** using both machines M1 and M2. 85

6.9 Parallel encryption and decryption implementation results with **4 threads in clock cycles per byte** (ccpb), for **1000** sectors of **4KB** and a **128-bit tweak** using machine M2. 85

B.1 Encryption and decryption implementation throughput results in **Gbit/sec**, for a **4KB** buffer and a **128-bit tweak** using both machines M1 and M2. (a) includes Key Expansion procedure and (b) does not include it. 105

List of Figures

3.1 The decryption of the ciphertext generated by adversary $\mathcal{A}$ 36

4.1 Games HCTR*1 and RAND1 54
4.2 Game RAND2 55
4.3 Game G2 56

5.1 The xtimes operation 60
5.2 AES encryption algorithm using FIPS-197 notation 66
5.3 AES encryption algorithm using AES-NI instructions set 68

6.1 Encryption (ENC) and decryption (DEC) implementation results **in clock cycles per byte** (ccpb), for a **4KB** buffer and a **128-bit tweak** using both machines M1, figure (a) includes Key Expansion procedure and (b) does not include it. 78
6.2 Encryption (ENC) and decryption (DEC) implementation results **in clock cycles per byte** (ccpb), for a **4KB** buffer and a **128-bit tweak** using both machines M2, figure (a) includes Key Expansion procedure and (b) does not include it. 79
6.3 Encryption and decryption comparison of other disk encryption schemes with some TES modes **in clock cycles per byte** (ccpb), for a **4KB** buffer using both machines M1 and M2. Results include key schedule. 80
6.4 Comparison of parallel encryption and decryption implementation results with **2 threads in clock cycles per byte** (ccpb), for (a) **200** and (b) **1000** sectors of **4KB** and a **128-bit tweak** using machine M1. 86
6.5 Comparison of parallel encryption and decryption implementation results with **2 threads in clock cycles per byte** (ccpb), for (a) **200** and (b) **1000** sectors of **4KB** and a **128-bit tweak** using machine M2. 87

6.6 Comparison of parallel encryption and decryption implementation results with **2 and 4 threads in clock cycles per byte** (ccpb), for (a) **1000** sectors of **4KB** and a **128-bit tweak** using machine M2. . . . 88

Chapter 1

Introduction

Nowadays, it is very difficult to imagine a life without computers. Today, every company uses computers to store its data and to make different kinds of operations. When we go to a store and use our credit card, many computers process our information and perform different kinds of transactions and store a record of them. When we need to get some cash, we use automatic teller machines (ATMs) that are computerized too. Also, personal computers and similar devices have become ubiquitous, and even a common man depends heavily on personally owned computational devices and technologies. An ordinary citizen today, stores many sensitive information in a personal desktop or laptop computer or in other specialized digital storage devices.

The loss of sensitive data stored in storage devices is a grave problem of today. Recent statistics show that an organization with 100,000 laptops may lose on average several of them per day [19]. 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 may fall in the wrong hands which could potentially cause much greater damage than the one incurred by the mere physical loss of the hardware and the data.

To ensure the confidentiality of stored data, one can use cryptographic techniques. Cryptography was traditionally used for securing data in transit, but now the security of stored information has also gained the same importance as that of the security of data in transit. But, the cryptographic techniques used for storage security are in some sense different from other cryptographic techniques. In this thesis we deal with a class of cryptographic techniques which are suitable for providing privacy of data stored in hard disks. In the following sections we try to motivate the problem of disk encryption and finally discuss the contents and scope of the thesis.

1.1 Some Notions of Disk Encryption

All computers store data on disks, and the value of the data on disk is not always measurable. In order to protect confidentiality of the data stored on a computer disk a computer security technique called disk encryption is used. With the advent of more powerful processors in the last decade, the data throughput of ciphers surpassed that of the data rates in hard disks. Hence, encryption is no longer a bottleneck, and the interest in the topic of disk encryption has increased.

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. A hard disk is partitioned into fixed-length sectors, usually 512 bytes or 4096 bytes for the new big format. Such a block wise (or sector wise) organization is also true for other storage devices like NAND flash memories, etc. It has been argued that the best way to achieve encryption of hard disks is to encrypt individual sectors. The encryption/decryption algorithm is not required to have knowledge of the high level organization of the data in the disk in terms of the file systems, etc. The only organization visible to the encryption algorithm is that of the sectors, and a sector is encrypted before the disk controller writes it to the disk, and is decrypted after the disk controller reads it from the disk. This generic methodology has been termed as *low-level disk encryption* or *in-place disk encryption*.

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. Another important property to achieve is the following: if the same plaintext is stored in encrypted form in two different sectors, the cipher texts should look different. To achieve this, it is required to treat each sector in a different way for the purpose of encryption. The current solution to this problem is achieved by a special public parameter which is called a tweak. The sector address is considered to be a tweak, and the tweak is an input to the encryption algorithm, thus as an effect of different tweaks, two sectors which stores the same information would store different ciphertexts.

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 ciphertext 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. During the last few years, there has been an intense research addressing this problem, and it appears that both practitioners and researchers have come to the conclusion that a class of encryption algorithms called tweakable enciphering scheme (TES) offers the best solution to the in-place disk encryption problem [30].

An important activity regarding the problem of disk encryption was initiated by the IEEE security in storage working group (SISWG)[1]. This working group is still working on various aspects of storage security and they aim to standardize cryptographic algorithms for various storage media. The SISWG has also considered encryption schemes other than TES, we summarize in more details the activities of this working group later in Section 3.1 (in page 23).

1.2 Contribution and Scope of the Thesis

In spite of the active effort of the community in constructing new schemes for disk encryption and the standardizing activities, there have been a very few optimized implementations of such schemes reported in literature. In [10, 46, 47] efficient hardware implementations of some TES were reported, but no performance comparison in software is yet available. This thesis aims to bridge this important gap.

The main thrust of this thesis is to generate experimental performance data of the available Tweakable Enciphering Schemes in modern processors. Tweakable enciphering schemes are generally constructed using a block cipher. Some of the reported schemes also uses some finite field operations. In all previous works the efficiency of a scheme was argued based on the number of block-cipher calls and the number of finite field multiplications it required. Since these were the most expensive operations to be performed and the other overhead was negligible. The new generation of Intel processors have been equipped with the new instructions for AES encryption/decryption and carry free multiplication of two 64 bit operands. These instruction set extension is commonly known as AES NI, were first introduced in the 2010 Intel Core processor family, based on the 32 nm Intel microarchitecture codename “Westmere”

and are also available for AMD procesors starting from its CPU generation called "Bulldozer". These instructions are very important from the perspective of diverse kinds of cryptographic implementations. With this new hardware support the two most expensive operations of the existing TES (the block cipher calls and the finite field multiplications) can be now implemented much more efficiently than in older processors.

The cryptographic community has already started taking advantage of these new instructions. Like, in [26], the authors report an efficient method to implement the Galois Counter Mode of operation. This contribution adds motivation on using AES-GCM for high performance secure data networking. In [27] is contained all the information regarding the instructions, and its usage for computing the Galois Hash. It also provides code examples for the usage of the carry free multiplier, together with the new AES instructions. Recently, in [41], T. Krovetz and P. Rogaway, study the software performance of authenticated encryption (AE) modes like CCM [16], GCM [51], and OCB [57] across a variety of platforms using the AES-NI support. The AES-NI support, specially the carry-free multiplier has also been utilized in implementing other primitives useful for elliptic curve cryptosystems, pairing based cryptosystems etc. [62].

In spite of the active effort in validating the usefulness of the AES-NI support for diverse kinds of cryptographic implementations, there are no optimized software implementations of disk encryption schemes available in the literature. We present here optimized software implementations of all important TES considering various implementation scenarios and provide a performance comparison between them. In our implementations we use the new intel AES NI instructions and report performance in two different processors. It is expected that the performance figures provided in this study would scale to other processors with AES NI support. Keeping in mind the support of AES-NI we also modify two existing constructions giving rise to two new schemes named HCTR* and HMCH2. We show that these new proposals perform much better than their predecessors.

Though it has been argued that TES are the only modes which provide adequate security for the in-place disk encryption application, over the years other kinds of schemes have also been proposed for this purpose. We provide a brief discussion on those schemes which include XTS and BitLocker. XTS is a modified tweakable block cipher which is now a NIST standard and BitLocker is a full disk encryption feature included in some OS versions of Microsoft's Windows Vista and Windows 7 [19]. We

also analyze XTS [63], and give a concrete attack on this scheme.

Since the use of multi-core processor is an emerging research topic in the context of cryptographic implementations we consider the usage of multiple cores for the application of disk encryption. Considering the disk encryption application where the message length is fixed and multiple messages are needed to be encrypted/decrypted in a short period of time, we use a strategy for parallelization. We also report these performance results and give a comparison between both serial and multi-threaded implementations.

The rest of the document is organized into 6 chapters, we discuss in short the contents of these Chapters next:

- In **Chapter 2**, we discuss some architectural issues in modern processors. In particular, we explore the different ways in which parallelism can be achieved in a modern processor. We discuss in brief instruction level parallelism, data level parallelism and thread level parallelism. We also shortly describe the evolution of the instruction set extensions in Intel processors which supports the paradigm of single instruction multiple data (SIMD). Finally we give a brief overview of parallel programming tools required to exploit the power of multiple core CPUs which are available today.

- In **Chapter 3** we describe the existing disk encryption schemes. In particular, most of the chapter is dedicated to the description of tweakable enciphering schemes, which is considered to be a main paradigm for disk encryption. In addition, we discuss some schemes other than TES which have been proposed for the purpose of disk encryption. Among them, an interesting proposal is XTS, which has been recently standardized by the National Institute of Standards and Technology (NIST) [17] as an encryption algorithm for block oriented storage devices. We argue, why XTS does not provide adequate security for this application, and formulate a concrete attack on the scheme. Though several weaknesses of XTS have been acknowledged by the community, but to our knowledge a concrete attack has not appeared in the literature before. Hence, this attack can also be seen as a contribution of this thesis.

- In **Chapter 4** we describe two new tweakable enciphering schemes. The new schemes are called HCTR* and HMCH2 which are modifications of two existing schemes HCTR and HMCH[BRW] respectively. The modifications which yields

the new schemes are subtle, and have been done keeping in mind the AES-NI support. The experimental results presented in Chapter 6 validates that the modified schemes perform better than their predecessors. In general one can prove security of a tweakable enciphering scheme by a standard reductionist argument which reduces the security of the whole scheme to that of the underlying block cipher. Both HCTR and HMCH[BRW] have a proof for its security. Thus, it is required that the security of the modified schemes should also be proved. We adapt the original proofs of the original schemes and successfully demonstrate that the modifications are also provably secure.

- In **Chapter 5**, we provide some implementation details. In particular we describe how we implement the basic building blocks for the various schemes that we study in this thesis. We provide details of the implementations for the finite field operations required for our purpose and also the description of the AES blockcipher, which is used in all schemes.

- In **Chapter 6** we present the experimental results. We discuss about the characteristics of the processors used for this study, and finally present performance data of our implementations in various scenarios. We also provide some preliminary analysis of the results.

- We conclude the thesis in **Chapter 7** and discuss about some issues which we would like to take up for future study.

Chapter 2

Exploiting Parallelism in Modern Processors

The crux of this thesis is about efficient implementations of a class of cryptographic algorithms which are suitable for the application of disk encryption. The main goal of the implementations is to obtain high speed. To achieve this goal we exploit various features present in modern processors. In this Chapter we try to summarize some of the ways one can exploit parallelism in a modern processor. We begin with a brief description of an instruction set architecture in Section 2.1. In Sections 2.2, 2.3 and 2.4 we discuss the three common types of parallelism which a modern processor provide, namely, instruction level parallelism, data level parallelism and thread level parallelism. In Section 2.3.1, we also discuss about some of the instruction set extensions which implements the paradigm of Single Instruction Multiple Data (SIMD) computing in the state of the art Intel and AMD processors. Finally, in Section 2.5, we discuss some issues about the choice of the programming language and some others about parallel computing.

2.1 Instruction Set Architecture

In general, computer architecture can be defined as the practical art of selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals [33]. Computer architecture can be broadly classified into instruction set architecture and microarchitecture.

An instruction set, or instruction set architecture (ISA), is the part of the com-

puter architecture related to programming. An ISA includes the native data types, instructions, registers, addressing modes, memory architecture, interrupt exception handling, and external input/output. An ISA incorporates a specification of the set of *opcodes* (machine language), and the native commands implemented by a particular processor. On the other hand, microarchitecture is the set of processor design techniques used to implement the instruction set. Computers with different microarchitectures can share a common instruction set. For example, the Intel Pentium and the AMD Athlon implement nearly identical versions of the x86 instruction set, but have radically different internal designs [33].

The instruction set architecture of a modern processor enables parallel computing in various ways. In the following three sections we discuss in brief the way a modern processor enables parallel computing.

2.2 Instruction Level Parallelism

A computer program is a sequence of instructions. In a program there may exist certain dependencies among the various instructions it contains. If two or more instructions are independent of each other then these instructions can be executed in any order and even they can be executed in parallel without any effect on the results they produce. To understand how instruction level parallelism can be exploited in a program, we need to first be precise regarding what we mean by two instructions to be independent.

There can be generally three types of dependencies between instructions. These dependencies are summarized as below [33]:

1. **Data dependencies:** If the output of an instruction a is input to an instruction b, then we say that there is a data dependency between the instructions a and b. It is easy to see that data dependencies are transitive, i.e., if b depends on a and c depends on b, then surely c also depends on a. Data dependent instructions have always to be executed in an order which respects the dependency. So if $a \rightarrow b \rightarrow c$, i.e., b depends on a and c depends on b, then these instructions have to be executed in the order a, b, c, and there is no way they can be parallelized.

2. **Name dependencies:** It would be easier to explain this kind of dependency using an example. Consider the code below:

1. $A = X + 1;$
2. $B = A + 2;$
3. $A = Y + 3;$

In this piece of code the instructions 1 and 3 are dependent, in the sense that the order of their execution determines the value of B. But here, the instructions 1 and 3 has no data dependencies, i.e., the input of one is not dependent on the output of the other. But still they cannot be swapped and has to be performed in the specific order specified by the code. This type of dependencies are not true dependencies and can be easily removed by renaming of variables. Like for our example if we replace instruction 3 by $C = Y+3$, then the instructions 1 and 3 are no more dependent. This kind of dependencies, where two instructions write to the same location, are called named dependencies.

3. **Control dependencies:** In a program there are certain instructions which change the flow of execution abruptly, possibly based on some values of variables attained in run-time. Such instructions are called branch instructions or jump instructions. These instructions are used to implement loops and other conditional operations. Swapping or reordering an instruction which contains a branch variable or instructions which belongs to different branches of execution can lead to un-desirable effects. Such dependencies are called control (or sometimes branch) dependencies. We illustrate this with a simple example. Consider the piece of code below:

A1. if $(a == b)$
A2. $\quad a = a + b;$
A3. else $b = a + b;$

In the above example A2 is control dependent on A1, and thus cannot be parallelized.

If two instructions are independent then they can be suitably parallelized. Modern processors apply various techniques for parallel execution of multiple instructions. We briefly describe next some of these techniques.

Instruction pipelining: The main goal of instruction pipelining is to increase the instruction throughput, i.e., to increase the number of instructions executed per unit time. The fundamental idea is to split the processing of an instruction into a series of independent steps, with storage at the end of each step. For example, a typical instruction in a modern reduced-instruction-set (RISC) processor can be broken into the following five steps:

1. Instruction fetch
2. Instruction decode and register fetch
3. Execute
4. Memory access
5. Register write back

For independent instructions these steps can overlap. While one instruction is decoded, the next instruction can already be fetched from memory and during execution of one instruction the next instruction can be decoded and so on. This overlapping in the execution of independent instructions is called pipelining. Note that some stages in the pipeline can also be overlapped for dependent instructions as long as the dependency is not a control dependency. There can be several issues in the process of pipelining, an important one is when the code has a branch. In this case, the processor could waste some clock cycles whenever it feeds a branch into the pipeline and later discovers it has chosen the wrong branch. In order to solve this problem, processors use a method called *branch prediction* which uses specialized algorithms based on history of other nearby branches to predict which way a new branch will go. There are different branch prediction techniques used depending on the processor. However, a good programming advice is avoiding branches as far as possible.

It may be possible to break a single instruction into more finer steps than the five steps described above. With such finer sub-division, deeper pipelines are possible. More pipeline stages have the advantage that the CPU needs to do less work in each pipeline stage which allows an increase in the CPU frequency [33].

Superscalar processing: A superscalar processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundant functional units on the processor. Each functional unit is not a separate CPU core but an execution resource within a single CPU such as an arithmetic logic unit, a bit shifter, or a multiplier. A superscalar CPU is thus capable of handling

multiple instructions in the same pipeline stage by duplicating the various resources of a single CPU, and is thus independent of instruction pipelining. In a superscalar CPU the dispatcher reads instructions from memory and decides which ones can be run in parallel, dispatching them to redundant functional units contained inside a single CPU. Therefore a superscalar processor can be envisioned having multiple parallel pipelines, each of which is processing instructions simultaneously from a single instruction thread.

Specialized instruction scheduling: In traditional processors, instructions are scheduled in the same order in which the instructions appear in a program. Such a scheduling is called in order instruction scheduling. In processors where in order scheduling takes place, it is the programmers task to sequence the instructions in such a manner so that instruction pipelining and other parallelization techniques inherent to the processor can be utilized on the given sequence of instructions.

Many modern processors, in particular most Intel and AMD processors, execute the instructions in a program out of order, i.e., the processor dynamically schedules instructions at run time and may not respect the sequence in which the instructions appear in the program. The main advantage of out-of-order execution is that programs can be compiled once and achieve reasonable performance on different microarchitectures. Furthermore it releases some pressure from the compiler, it becomes easier and cheaper to develop compilers for new microarchitectures.

2.3 Data Level Parallelism and SIMD Instructions

When the same computations need to be carried out on multiple independent inputs then these computations can be parallelized. Such a parallelization is called data level parallelism. In modern desktop processors data level parallelism is achieved through specialized instructions called SIMD instructions which works on vector registers.

Flynn [22] classifies "very high speed computers" in the following classes:

1. Single Instruction Stream, Single Data Stream (SISD)
2. Single Instruction Stream, Multiple Data Stream (SIMD)
3. Multiple Instruction Stream, Single Data Stream (MISD)
4. Multiple Instruction Stream, Multiple Data Stream (MIMD).

Each of this class can be seen as a specific paradigm to achieve high performance computers. Modern processors may have in them more than one of the above paradigms implemented. In our context, the SIMD is particularly important as this is the way to implement data level parallelization. The two implementations of SIMD found in current processors are vector registers and single instruction, multiple threads (SIMT).

Vector registers: The idea of vector registers is to keep multiple values of the same type in one register. Arithmetic operations are then carried out on all of these values in parallel by vector instructions (sometimes called *SIMD instructions*). For example, suppose one requires to add four pairs of 32 bit integers. If these integer values are stored in consecutive memory locations then they can be loaded into two 128 bit registers by two loads, can be added by a single add instruction, and the result can be stored back into the memory using a single store instruction. It is evident that this leads to considerable savings in CPU cycles.

Single instruction multiple threads: Many modern graphics processing units (GPUs) implement SIMD by executing the same instruction in parallel by many hardware threads. The program is the same for all threads. Accessing different input data is realized by loads from addresses that depend on a thread identifier. The main difference compared to the concept of vector registers is in handling of memory loads: Unlike current implementations of vector registers, SIMT makes it possible to let all threads load values from arbitrary memory positions in the same instruction. Note that this corresponds to collecting values from arbitrary memory positions in a vector register in one instruction. But as stated earlier, such loads are not possible in vector registers.

2.3.1 SIMD instructions in modern processors

Though the SIMD paradigm was first utilized in designing of vector supercomputers like Cray-1 [58], recently this paradigm has seen a glorious come-back in desktop computers. Starting from the early nineties desktop processors became powerful enough to support real-time gaming and video processing, and demand grew for this particular type of computing power, and microprocessor vendors turned to SIMD to meet the demand. In the last decade in most of the processor families new extensions of the instruction sets have been incorporated and these instructions implement the SIMD

paradigm through vector registers. We summarize the evolution vector instruction set extensions in the Intel x86 family [37]:

MMX: MMX is a SIMD instruction set designed by Intel, introduced in 1996 with their P5-based Pentium line of microprocessors, designated as "Pentium with MMX Technology". The MMX instruction added eight 64 bit registers to the then existing Pentium architecture. The new registers were named MM0 to MM7 and they are capable of holding 8 bytes, 4 words (16 bits), 2 double words (32 bits) or a single quad word (64 bits). Along with the new registers 57 new instructions were added. The new instructions were all directed towards increasing efficiency of multimedia related applications.

SSE: Streaming SIMD Extensions (SSE) is a SIMD instruction set extension to the x86 architecture, designed by Intel and introduced in 1999 in their Pentium III series processors. It is said that Intel's SSE extension was proposed as a reply to AMD's 3DNow, which had debuted a year earlier. SSE contains 70 new instructions, most of which work on single precision floating point data. SSE originally added eight new 128-bit registers known as XMM0 through XMM7. The AMD64 extensions from AMD (originally called x86-64) added a further eight registers XMM8 through XMM15, and this extension is duplicated in the Intel 64 architecture. There is also a new 32-bit control/status register, MXCSR. The registers XMM8 through XMM15 are accessible only in 64-bit operating mode. SSE used only a single data type for XMM registers: four 32-bit single-precision floating point numbers

SSE2: Streaming SIMD Extensions 2 (SSE2) was introduced with the Pentium 4 and is a major enhancement to SSE. SSE2 adds new math instructions for double-precision (64-bit) floating point and also extends MMX integer instructions to operate on 128-bit XMM registers. SSE2 enables the programmer to perform SIMD math on any data type (from 8-bit integer to 64-bit float) entirely with the XMM vector-register file, without the need to use the legacy MMX or FPU registers. Many programmers consider SSE2 to be "everything SSE should have been", as SSE2 offers an orthogonal set of instructions for dealing with common data types.

SSE3: Streaming SIMD Extensions 3, also known by its Intel code name Prescott New Instructions (PNI), is the third iteration of the SSE instruction

set for the IA-32 (x86) architecture. Intel introduced SSE3 in early 2004 with the Prescott revision of their Pentium 4 CPU. In April 2005, AMD introduced a subset of SSE3 in revision E of their Athlon 64 CPUs. SSE3 is a minor extension of SSE2, it adds 13 new instructions mostly related to mathematical operations for digital signal processing and thread management.

SSE4: SSE4 (Streaming SIMD Extension 4) is a CPU instruction set used in the AMD K10 (K8L) and Intel Core microarchitecture. SSE4 is a significant upgrade from SSE3. SSE4 contains 54 new instructions: A subset consisting of 47 instructions referred to as SSE4.1 and the 7 remaining instructions as a second subset called SSE4.2. AMD also added four new SSE instructions, naming the group SSE4a.

AVX: Advanced Vector Extensions (AVX) is an extension to the x86 instruction set architecture for microprocessors from Intel and AMD proposed by Intel in March 2008 and first supported by Intel with the Sandy Bridge processor and by AMD with the BullDozer processor. AVX provides new features, new instructions and a new coding scheme. The width of the SIMD registers is increased from 128 bits to 256 bits, and renamed from XMM0 - XMM15 to YMM0 - YMM15. In processors with AVX support, the legacy SSE instructions (which previously operated on 128-bit XMM registers) now operate on the lower 128 bits of the YMM registers. Additionally, AVX introduces a three-operand SIMD instruction format, where the destination register is distinct from the two source operands. For example, an SSE instruction using the conventional two-operand form $a = a + b$ can now use a non-destructive three-operand form $c = a + b$, preserving both source operands. AVX's three-operand format is limited to the instructions with SIMD operands (YMM), and does not include instructions with general purpose registers (e.g. EAX). Such support will first appear in AVX2.

As stated earlier, the shift to SIMD paradigm in the desktop computers were mainly motivated by the multi-million dollar gaming industry which required very fast graphics and related multimedia operations. But, in 2010 there was an instruction set extension called **AES-NI** introduced in the x86 processor which is of particular interest to the the cryptographic community. The AES-NI consists of various instructions with which the Advanced Encryption Standard block-cipher can be easily

implemented. In addition it contains a special instruction called PCLMULQDQ with the help of which carry free multiplication of two 64 bit integers can be performed. The PCLMULQDQ instruction can be used for efficient implementation of multipliers in a finite binary extension field. More discussions about the AES-NI instructions are provided in Chapter 5.2.1 (in page 66), where we discuss about the implementation details of some of the modes considered in this study.

2.4 Multicore Processors and Thread Level Parallelism

The last decade has seen exponential rise in CPU frequencies which has given rise to very fast CPUs. But in the last few years it was realized by the computer architects that CPU frequencies have reached near saturation, and it is unlikely that they can be further increased. This lead to a new paradigm of CPU designs which utilizes more than one processor core in a single processor. This kind of processors are commonly known as multicore processors. Each core in a multicore processor completely implements an architecture with all registers and units. But it may be the case that the cores share some resources like the caches, but architectures where each core has its dedicated cache are also available. In the most typical model of multicore architecture, the main memory is shared by the cores.

The performance gain in multicore processors heavily depends on the extent to which the application can be parallelized. For a programmer there exist the option to develop software which runs in multiple threads, and the threads communicate between them by using the shared memory. Developing such software is a non trivial task, thus multi-threaded implementations of common software is not very common. But performance critical applications, which are parallelizable, can gain a lot from a multi-threaded implementation. We describe more about parallel programming issues in Section 2.5.1 (in page 17).

2.5 Notes on Programming Languages and Tools

Choosing a proper programming language depends hugely on the application goals. The most preferred programming language from the point of view of a programmer is a high level language. High level languages help to create functionalities using

special reserved words (the syntax) to define memory locations, jumps over instruction codes, or handle input output data. The syntax of a high level language is generally intuitive, as it may hide low-level details of the instructions that ultimately generate the machine readable codes. Thus, it is easier to program in these languages and the code written in such languages is also easy to maintain and debug.

A code written in a high level language is generally compiled [1] and the compiler translates the code into a machine readable format. Thus, a programmer who develops his/her code in a high level language does not have much control on the code that is produced by the compiler. This is a serious disadvantage of high level languages, as till now the state of the art compilers are unable to produce code which is fully optimized for performance. Hence, generally it is difficult to develop a very high performance code using a high level language.

The alternative to high level language is coding in assembly language. An assembly code is very near to the hardware, and a programmer has full control over which resource is used by the program and in which manner, and they can be tuned to a large extent to produce highly optimized code. But, programming in assembly is tedious, it requires serious knowledge of the instruction set architecture of the machine for which the code is being written. Moreover, assembly codes are not easily readable and thus very difficult to debug and maintain.

Inline assembly: A trade off between assembly language and high level language is using inline assembly in a code written in a high level language. Inline assembly language programming refers to the practice of writing some parts of the code (which is written in a high level language) in assembly. The general practice is to write those parts of the code which have critical performance goals in assembly language, where as the other parts are in a high level language. Most compilers supports this practice, and they generate code respecting the parts which are written in assembly, i.e., by using a few directives it is possible that in the final code produced by the compiler the inlined assembly parts remains un-touched by the compiler. With inlining, it is thus possible to gain some aspects of the performance gains in a assembly program without totally losing the flexibility of a high level language.

There can be other advantages of inlining. For example, one way to utilize various SIMD instruction set extensions for the x86 architecture, is to implement the relevant code, which uses this instructions, in assembly and inline it in the program written

[1] some high level languages like MATLAB, Perl, Python or PHP are interpreted (not compiled)

in a high level language.

Intrinsic functions: Though the most direct way to use specialized processor instructions is to inline the assembly language instructions into the source code. However, this process can be time-consuming and tedious. In addition, some compilers may not support inline assembly language programming. Some compilers enables easy implementation of these instructions through the use of API extension sets built into the compiler. These extension sets are referred to as intrinsic functions or intrinsics [36]. Thus, we can say that an intrinsic function is a function available for use in a given language whose implementation is handled specially by the compiler. Typically, it substitutes a sequence of automatically generated instructions for the original function call, similar to an inline function. Unlike an inline function though, the compiler has an intimate knowledge of the intrinsic function and can therefore better integrate it and optimize it for the situation. The following are some of the advantages of using intrinsics:

- The compiler optimizes intrinsic instruction scheduling so that executables run faster.
- Intrinsics enable the use of the syntax of the high level language, in particular variables etc. can be named as required by the high level language instead of assembly language or hardware registers.
- Intrinsics provide access to instructions that cannot be generated using the standard constructs of the high level languages.

The work that we report in this thesis is highly performance critical. Our primary objective here is to obtain fast implementations. For this reason we heavily use the SIMD instructions. We write our codes in the high level language C with the intrinsic functions where ever possible. At some places we use a little amount of inline assembly also. Our codes are directed towards multicore processors, and we report some results where our code utilizes thread level parallelism using multiple cores. We describe next some issues in programming for multicore processors.

2.5.1 Basics of parallel programming

Parallel computing was an evolution of serial computing and attempts to save time or even money, for solving many problems that are considered to be so large and/or

so complex that could be impossible to solve them with a serial algorithm on a single computer. When we talk about parallel computing there are some concepts involved that need to be cleared.

For computing in parallel, we first need to divide the work into different tasks. A task is a logically discrete section of computational work, which can be seen as a set of instructions that are executed by a processor. A parallel program consists of multiple tasks running on multiple processors. In some cases, tasks need to exchange data which can be stored either in a distributed or in a shared memory. A shared memory, is the portion of physical memory where all processors have direct common access. A distributed memory is a specific architecture where each processor has its own memory.

Other than sharing data, tasks may need to communicate with each other for maintaining consistency. Such communications are generally called a synchronization. Synchronization of parallel tasks is also sometimes performed via the shared memory.

For parallel execution of different tasks there may be some extra overheads. The amount of time required to manage a parallel program is called parallel overhead. To measure the parallel overhead, there may be many things that are to be taken into account, like synchronization, data communications, task start-up and termination time, etc. The amount of parallel overhead generally determines whether it would be profitable to parallelize certain tasks, in some cases, the parallel overhead may exceed the gain obtained by parallelization, thus making parallelization redundant.

For evaluating a parallel program it is important to consider the following characteristics:

- Scalability: The ability that demonstrates that parallel speedup is proportional to the number of processors.

- Granularity : The amount of computation in relation to communications, it is a qualitative measure of the ratio of computation to communication. It is often related to how balanced the work load is between tasks.

High scalability suggests a low parallel overhead. But, in most cases there is a limit to the parallelization that can be achieved by adding more processors, i.e., most practical tasks are not infinitely scalable. Parallel overhead can be reduced by increasing the granularity within each task.

Threads: As discussed earlier the modern processors provide instruction level parallelism (like through instruction pipelining) and data level parallelism (through SIMD instructions). These forms of parallelism are inherent to the processor, and to exploit them explicit division into tasks and applying methods of parallel programming is not required. But to utilize the support of multiple cores, the task at hand need to be carefully subdivided into subtasks for the different cores. Here, techniques and tools of parallel programming becomes important. Exploiting the multiple cores is achieved by multi-threaded programming, hence this parallelism is referred to as thread level parallelism.

A thread is an independent execution state, which have their own machine registers, a call stack and the ability to execute code. But a thread is not an independent process. The differences between a thread and a process are basically the following:

- Threads exist within a process, which means they do not have distinct copies of state values like user ID, file descriptors, etc.
- Threads also share a single address space of memory segment belonging to their process.
- Threads can be scheduled by code without kernel assistance. This could be much faster than process scheduling.

Multithreading is a widespread programming and execution model which allows multiple threads to exist within the context of a single process. These threads share the resources of the process but are able to execute independently. The threaded programming model provides developers with an useful abstraction of concurrent execution. However, perhaps the most interesting application of the technology is when it is applied to a single process to enable parallel execution on a multiprocessor system.

This advantage of a multi-threaded program allows it to operate faster on computer systems that have multiple CPUs, CPUs with multiple cores, or across a cluster of machines, because the threads of the program naturally lend themselves to truly concurrent execution. In such a case, the programmer needs to be careful to avoid race conditions, and other non-intuitive behaviors. In order for data to be correctly manipulated, threads will often need to rendezvous in time in order to process the data in the correct order. Threads may also require mutually-exclusive operations (often implemented using semaphores) in order to prevent common data from being

simultaneously modified, or read while in the process of being modified. Careless use of such primitives can lead to deadlocks.

Tools for multi-threaded implementations: Many high level programming languages provide in-built support for multi-threading, and some provide access to the native threading APIs provided by the operating system. For developing multi-threaded applications in C or C++, there are primarily two different ways to go: OpenMP and POSIX threads (Pthreads).

OpenMP: Open Multi-Processing (OpenMP) is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most processor architectures and operating systems. It consists of a set of compiler directives, library routines, and environment variables which can be suitably used for building parallel code. OpenMP is based on a fork-join execution model. Execution begins with a single thread called the master thread, and when parallel sections are encountered within then the master thread spawns a set of parallel threads. The parallel portion of the code gets executed, and at the end of the parallel region all threads terminate leaving only the master thread. Parallel regions and other OpenMP constructs are defined through compiler directives. The `#pragma omp parallel` is one of the main OpenMP constructs which specifies a piece of code to be run in multiple threads. Starting from the beginning of the region, the code is duplicated and all threads execute that code. While OpenMP does not guarantee a priori how many threads will be created, it usually chooses a number equivalent to the number of available execution pipelines. On standard multiprocessor environments, this number is the number of processors. On systems with processors endowed with Hyper-Threading Technology, the number of pipelines is twice the number of processors. An API function or environment variable can be used to override the default number of threads.

POSIX threads: Pthreads is an API which was first defined in the ANSI/IEEE POSIX 1003.1-1995 standard, and since then has continued to evolve and by now many compilers support it. The Pthreads library contains around 100 subroutines that are used for thread management which includes routines dealing with synchronization, communications between threads and routines for using read/write locks and barriers. The current standard is only defined for

C, but there also exist some Fortran compilers that provide a similar Pthread APIs. Pthreads provides extensive control over threading operations, it is an inherently low-level API that mostly requires multiple steps to perform simple threading tasks. For example, using a threaded loop to step through a large data block requires that threading structures be declared, that the threads be created individually the loop bounds for each thread be computed and assigned to the thread, and ultimately that the thread termination be handled, all this must be coded by the developer. If the loop does more than simply iterate, the amount of thread-specific code can increase substantially. Thus, Pthreads provide much less abstraction than OpenMP, making development much more faster in OpenMP than in Pthreads. The number of threads in Pthreads applications needs to be hard coded in the code and this makes codes less portable across platforms with variable number of CPUs.

Summary

As the main goal of the thesis is to obtain high speed efficient implementations of cryptographic algorithms suitable for the application of disk encryption. We described some of the features present in modern processors which we considered in order to achieve this goal. Although modern processors may have in them more than one of the Flynn paradigms implemented, we heavily use the SIMD paradigm and our codes are directed towards multicore processors. We also described some issues in programming for multicore processors as we report some results where our code utilizes thread level parallelism using multiple cores.

Chapter 3

Schemes for Disk Encryption

In this chapter we give an overview of the important disk encryption schemes which are available in the literature. Our primary focus is on a class of block cipher modes of operation called tweakable enciphering schemes (TES). It is widely accepted that TES are the most suitable methods applicable for low level disk encryption. Nevertheless there have been other proposals for disk encryption which are not TES. We also discuss some important candidates of those classes.

In what is to follow, we first give a brief description of the activities of the IEEE working group on security in storage (SISWG) [34]. SISWG has been working in the past few years for formulating standards for various tasks related to security in storage. The activities of this working group have been a guiding force towards development of new methods for secure storage. Later in Section 3.3, we discuss about tweakable enciphering schemes and also give algorithmic descriptions of the specific schemes that we chose for this study. In Section 3.4 we give descriptions of some other schemes which are not TES but have been proposed for the application.

3.1 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 [17]. XTS is derived from a previous construction of Rogaway [57] which was called XEX. XEX is a tweakable block cipher and in [57] 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 [42] that use 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 [42], 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 [63]. In spite of these criticisms XTS has been standardized by NIST [17].

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 [30]. 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. It has been shown in [47, 10], that some of the existing TES when implemented in FPGAs can outperform the data rates of the most modern disk controllers, which shows that these schemes possibly can be used in real

life disk encryption. 1612.2 has chosen two modes EME2 (a variant of EME [31]), and XCB [48] 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 in [47] show XCB to be the least efficient mode and both XCB and EME are covered by patent claims.

3.2 Some Notations

The set of all n bit strings would be denoted by $\{0,1\}^n$. Given a finite binary string X, we write $|X|$ to denote its length in bits. By $X||Y$ we mean the concatenation of two finite strings X and Y. For a positive integer n, 0_n represents the $n-$times repetition of 0's. If X is a finite set, then $|X|$ denotes its cardinality.

The set of strings $\{0,1\}^n$ can be treated in multiple ways. In most cases $\{0,1\}^n$ is treated as the finite field $\mathbb{F}_{2^n}$. Thus, for $X, Y \in \{0,1\}^n$, by $X \oplus Y$ and XY we denote the addition and multiplication in the field $\mathbb{F}_{2^n}$. The elements in $\{0,1\}^n$ can be treated as polynomials of degree at most $n-1$ over $\mathbb{F}_2$, and with such a representation $X \oplus Y$ is ordinary polynomial addition where the coefficients are added modulo 2, and can be achieved by the bitwise XOR of X and Y. XY can be obtained by multiplying the polynomials associated with X and Y modulo an irreducible polynomial of degree n representing the field $\mathbb{F}_{2^n}$. If the strings in $\{0,1\}^n$ are treated as polynomials, then the string 10 represents the monomial x. Thus for $X \in \{0,1\}^n$ by xX we would mean the multiplication of the polynomial representing X with the monomial x modulo the irreducible polynomial representing the field. Further this operation xX will be called xtimes.

A block cipher is a function $E : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ such that for any $K \in \mathcal{K}$, $E(K,\cdot) = E_K(\cdot)$ is a permutation on $\{0,1\}^n$. We shall usually write $E_K(X)$ instead of $E(K,X)$. The positive integer n is the block length, and an $n-$bit string is called a block.

3.3 Tweakable Enciphering Schemes

Tweakable enciphering schemes (TES), are block cipher modes of operation which provide security in the sense of a tweakable strong pseudo-random permutation (SPRP)[1]. TES are based on the notion of tweakable block ciphers introduced in [43] (this important piece of work was recently published in a much expanded form in [44]).

Given a message space $\mathcal{M} = \cup_{i>0}\{0,1\}^i$, a tweak space $\mathcal{T} \subset \{0,1\}^*$ and a key space $\mathcal{K} \subset \{0,1\}^*$, a tweakable enciphering scheme can be defined as a function $\mathbf{E} : \mathcal{K} \times \mathcal{T} \times \mathcal{M} \to \mathcal{M}$. This notion is very similar to a tweakable block cipher [43]. A tweakable block cipher $\tilde{E}$ is defined as $\tilde{E} : \mathcal{K} \times \mathcal{T} \times \{0,1\}^n \to \{0,1\}^n$. As in a block cipher, the message and the cipher space of a tweakable block cipher are also restricted to fixed length strings. But, in a TES the message space can contain variable length and arbitrarily long strings. The tweak aims to provide variability of the ciphertext, with the property that if the same plaintext is encrypted using the same key and different tweaks then different ciphertexts would be produced. The tweak is not a part of the key, it is a public quantity. The use of tweaks is fundamental for the purpose of disk encryption. Generally, the sector address of a disk sector is treated as a tweak, this ensures that the tweaks for two different sectors are always different. Thus, such a use of the tweak guarantees that if the same message is written in two different sectors then the ciphertexts would be different. This property prevents an adversary to know that two sectors contain the encryption of the same plaintext. Also, the SPRP property of a TES guarantees that each bit of ciphertext is dependent on all the bits of the plaintext and vice versa, a property which makes TES very secure for the disk encryption application. Though, the general definition of a TES considers messages and tweaks of arbitrary length but for the application of disk encryption such generality is not required, as the sector lengths are of fixed size and tweaks (which are sector addresses) can also be suitably coded to fit into fixed length strings. As TES are generally constructed using a block-cipher, for efficiency reasons tweaks can be considered to be of the same length of that of the block length of the block cipher and the message lengths to be a suitable multiple of the block cipher. Such length restrictions on the messages and tweaks in most cases result in more efficient constructions than a construction which is more general.

In this Section we will provide a brief history of the constructions of TES known

[1]For clarity we skip formal security definition of TES in this chapter, we provide such formal definitions and describe the notion of SPRP in 4.2.1 (in page 43)

till date and finally provide specifications of the schemes that were selected for this study.

3.3.1 A brief history of the known constructions of TES

The first work to present a scheme which is very near to a TES was by Naor-Reingold [53]. 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 [43].

The first construction of a tweakable SPRP was presented in [30] which was called the CMC mode of operation. Also, in [30] 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. In this application, the disk is encrypted sector-wise and the sector address corresponds to the tweak. 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 [31]. 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*[28].

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[48, 50]. In [48] the security of the construction was not proved, which was done much later in [50].

HCTR [64] is another construction which uses the same methodology of XCB. A serious drawback of HCTR was that the security proof provided in [64] only guar-

anteed 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 [13] 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 [52] another variant of HCTR was proposed which provides a quadratic security bound and later in [11] a quadratic security bound of the original HCTR construction (as proposed in [64]) was proved.

ABL [49] 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-Reingold construction has also been further used to construct TES and they are categorized as *hash-encrypt-hash* type. Examples of constructions on this category are PEP [12], TET [29], HEH [59]. 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 a recent work [60] significant refinements of constructions of *hash-counter-hash* and *hash-encrypt-hash* constructions were provided. The main idea in [60] 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 [60] 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].

3.3.2 The schemes considered for this study

In Table 3.1 we summarize almost all the existing TES. In our study we consider all known efficient candidates, a scheme is left out either because it is known to be very inefficient or there exist a more efficient successor of the scheme. Many of

Table 3.1: List of existing TES

Mode	Reference	In this study	Remark
CMC	[30]	No	Uses two layers of CBC encryption; cannot be parallelized;
EME	[31]	No	Uses two layers of ECB, cannot encrypt more than 2048 bytes if AES-128 is used
EME2	[35]	Yes	Derivative of EME, a candidate for the standard P1619.2
XCB	[48]	Yes	Candidate for standard P1619.2
HCTR	[64]	Yes	The original work proved a very weak security bound which was improved in [11]
HCH	[13]	No	Has been further improved in HMCH
HMCH	[60]	Yes	Has two versions HMCH[Poly] and HMCH[BRW]. We study both variants
PEP	[12]	No	Known to be inefficient
TET	[29]	No	Has been further improved in HEH
HEH	[60]	Yes	Has two variants HEH[Poly] and HEH[BRW]. We study both variants
HCTR*	Chapter 4	Yes	A modification of HCTR
HMCH2	Chapter 4	Yes	A small modification of HMCH

the schemes considered have several variants, but we consider only those variants which are suitable for disk encryption, as this is the only known application of TES. We consider those variants which are designed for fixed length messages and a fixed length tweak. For disk encryption, the sector address is considered as the tweak, thus we consider the tweak length to be same as the block length of the underlying block cipher. We use AES 128 as our block cipher in all the implementations. The algorithms used for the schemes are discussed in the next section.

3.3.3 Description of some schemes

In this section we describe the existing TES that we selected for this study. The two new schemes HCTR* and HMCH2 which are proposed in this work are described later in Chapter 4. The descriptions in this section assume that the tweak length is same as the block length of the underlying block-cipher and the message length is fixed and is a multiple of the block length of the block cipher.

Specification of EME2

As stated earlier, EME2 is a candidate for the standard P1619.2, it is a successor of EME. EME uses a block cipher $E : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ and three keys K_1, K_2 and K_3. Out of the three keys only one key K_1 is used for the underlying block cipher E and the two other n bit keys K_2 and K_3 are used for other purposes (this would be evident soon from the description in Algorithm 3.1). EME2 consists of two layers of masked ECB encryption with a layer called *intermediate mixing* in-between. The complete specification of EME2 is given in Algorithm 3.1 which is self explanatory.

Algorithm 3.1: Encryption using EME2

EME2.$\mathcal{E}_K^T(P_1 \ldots, P_m)$
// $K = K_1||K_2||K_3$
$T^* \leftarrow E_{K_1}(xK_3 \oplus T) \oplus xK_3$;
$L \leftarrow K_2$;
for $i \leftarrow 1$ **to** m **do**
    $PPP_i \leftarrow E_{K_1}(L \oplus P_i)$;
    $L \leftarrow xL$;
$MP_1 \leftarrow (PPP_1 \oplus PPP_2 \oplus \ldots \oplus PPP_m) \oplus T^*$;
$MC_1 \leftarrow E_{K_1}(MP_1)$;
$M_1 \leftarrow MP_1 \oplus MC_1$;
for $i \leftarrow 2$ **to** m **do**
    $j \leftarrow \lceil i/n \rceil$, $k \leftarrow (i-1) \bmod n$;
    if $k = 0$ **then**
        $MP_j \leftarrow PPP_i \oplus M_1$;
        $MC_j \leftarrow E_{K_1}(MP_j)$;
        $M_j \leftarrow MP_j \oplus MC_j$;
        $CCC_i \leftarrow MC_j \oplus M_1$;
    else
        $CCC_i \leftarrow PPP_i \oplus x^k M_j$
$CCC_1 \leftarrow MC_1 \oplus (CCC_2 \oplus \ldots \oplus CCC_m) \oplus T^*$;
$L \leftarrow K_2$;
for $i \leftarrow 1$ **to** m **do**
    $C_i \leftarrow E_{K_1}(CCC_i) \oplus L$;
    $L \leftarrow xL$;
return $(C_1, C_2, \ldots, C_m)$

Specification of XCB and HCTR

XCB and HCTR modes are a class of TES constructions that utilize a variant of the Carter-Wegman polynomial hash [9] combined with a counter mode of operation. For this reason, these modes are classified as Hash-Counter-Hash type modes. The encryption algorithms for XCB and HCTR are described in Algorithms 3.2 and 3.3 respectively.

Algorithm 3.2: Encryption using XCB

XCB.$\mathcal{E}_K^T(P_1 \ldots, P_m)$

1 $K_0 \leftarrow E_K(0^n)$ $K_1 \leftarrow E_K(0^{n-1}||1)$;
2 $K_2 \leftarrow E_K(0^{n-2}||1||0)$ $K_3 \leftarrow E_K(0^{n-2}||1^2)$;
3 $K_4 \leftarrow E_K(0^{n-3}||1||0^2)$;
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 $(C_2, \ldots, C_{m-1}, C_m) \leftarrow \boldsymbol{Ctr}_{K_2,H_1}(P_2, \ldots, P_{m-1}, P_m)$;
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** $(C_1, C_2, \ldots, C_m)$

$\boldsymbol{Ctr}_{K,S}(X_1, \ldots, X_\ell)$
// $S = S_l||S_r$, $|S_r| = 32$
1 **for** $i \leftarrow 1$ **to** ℓ **do**
2 $\quad Y_i \leftarrow X_i \oplus E_K(S_l||(S_r + i \mod 2^{32}))$;
3 **return** $(Y_1, \ldots, Y_\ell)$

The XCB scheme uses the hash $\mathbf{h}_h(X, T)$ where $X = X_1||X_2||\ldots||X_m$ for $X_i \in \{0,1\}^n$, and T is an n bit string. The hash is defined as:

$$\mathbf{h}_h(X, T) = X_1 h^{m+2} \oplus X_2 h^{m+1} \oplus \ldots \oplus X_m h^3 \oplus T h^2 \oplus (\mathsf{bin}_{\frac{n}{2}}(|P|)||\mathsf{bin}_{\frac{n}{2}}(|T|))h. \quad (3.1)$$

Where $\mathsf{bin}_{\frac{n}{2}}(|Y|)$ denotes the $\frac{n}{2}-$bit binary representation of $|Y|$.

For $X = X_1||X_2||\ldots||X_m$, where each $X_i \in \{0,1\}^n$, the polynomial hash $\mathbf{H}_h(X)$ used in HCTR is defined as:

$$\mathbf{H}_h(X) = X_1 h^{m+1} \oplus X_2 h^m \oplus \ldots \oplus (X_m) h^2 \oplus \mathsf{bin}_n(|X|)h \quad (3.2)$$

Specification of HEH and HMCH

Before describing the specifications of HEH and HMCH we would describe a special class of polynomials which were introduced in [56]. These polynomials were later

Algorithm 3.3: Encryption using HCTR

HCTR.$\mathcal{E}^T_{K,h}(P_1 \ldots, P_m)$
1 $MM \leftarrow P_1 \oplus \mathbf{H}_h(P_2 || \ldots || P_m || T)$;
2 $CC \leftarrow E_K(MM)$;
3 $S \leftarrow MM \oplus CC$;
4 $(C_2, \ldots, C_{m-1}, C_m) \leftarrow \mathbf{Ctr}_{K,S}(P_2, \ldots, P_{m-1}, P_m)$;
5 $C_1 \leftarrow CC \oplus \mathbf{H}_h(C_2 || C_3 || \ldots || C_m || T)$;
6 **return** $(C_1, C_2, \ldots, C_m)$

$\mathbf{Ctr}_{K,S}(X_1, \ldots, X_\ell)$
1 **for** $i \leftarrow 1$ **to** ℓ **do**
2 $\quad Y_i \leftarrow X_i \oplus E_K(S \oplus \mathsf{bin}_n(i))$;
3 **return** $(Y_1, \ldots, Y_\ell)$

improved in [6] and recently they were named as Bernstein Rabin Winograd (BRW) polynomials in [60]. In [60] these polynomials were successfully used to construct the tweakable enciphering schemes HEH and HMCH.

A BRW polynomial is defined recursively a follows

- $\mathsf{BRW}_h() = 0$
- $\mathsf{BRW}_h(X_1) = X_1$
- $\mathsf{BRW}_h(X_1, X_2) = X_1 h \oplus X_2$
- $\mathsf{BRW}_h(X_1, X_2, X_3) = (h \oplus X_1)(h^2 \oplus X_2) \oplus X_3$
- $\mathsf{BRW}_h(X_1, X_2, \ldots, X_m) = \mathsf{BRW}_h(X_1, \ldots, X_{t-1})(h^t \oplus X_t) \oplus \mathsf{BRW}_h(X_{t+1}, \ldots, X_m)$

  Where $t \in \{4, 8, 16, 32, \ldots\}$ and $t \leq m < 2t$.

Here we assume that $h, X_i \in \{0,1\}^n$. A BRW polynomial is interesting because it computes a polynomial hash of a message of m blocks using only $\lceil \frac{m}{2} \rceil$ multiplications and $\lceil \lg m \rceil$ squarings. A normal polynomial (as defined in Equation (3.3)) requires m multiplications. As computing squares in a binary extension field is much easier than computing multiplications, hence BRW polynomials provide significant computational advantage over a normal polynomial.

HEH is a TES mode that falls under the *hash-encrypt-hash* category while HMCH falls under *hash-counter-hash* type. The encryption algorithms for HEH and HMCH are described in Algorithms 3.4 and 3.5 respectively. As it is clear from the algorithms, HEH uses a polynomial hash $\psi_h()$ along with a electronic code book type mode and HMCH uses the hash $\psi_h()$ along with a counter mode. The hash function $\psi_h()$

in both HEH and HMCH can be either instantiated using an ordinary polynomial $\mathsf{poly}_h()$, defined as

$$\mathsf{poly}_h(X) = X_1 h^m \oplus X_2 h^{m-1} \oplus \ldots \oplus X_m h, \tag{3.3}$$

or by a BRW polynomial $\mathsf{BRW}()_h$. For these different instantiations we get four different modes, namely HEH[Poly], HEH[BRW], HMCH[Poly] and HMCH[BRW].

Algorithm 3.4: Encryption using HEH

HEH.$\mathcal{E}^T_{K,h}(P_1 \ldots, P_m)$

$\beta_1 \leftarrow E_K(T)$;
$\beta_2 \leftarrow x\beta_1$;
$U \leftarrow P_m \oplus \psi_h(P_1, \ldots, P_{m-1})$;
$PP_m \leftarrow U \oplus \beta_1$;
$CC_m \leftarrow E_K(PP_m)$;
$V \leftarrow CC_m \oplus \beta_2$;
for $i \leftarrow 1$ **to** $m-1$ **do**
    $PP_i \leftarrow P_i \oplus U \oplus x^i\beta_1$;
    $CC_i \leftarrow E_K(PP_i)$;
    $C_i \leftarrow CC_i \oplus x^i\beta_2 \oplus V$;
$C_m \leftarrow V \oplus \psi_h(C_1, \ldots, C_{m-1})$;
return $(C_1, C_2, \ldots, C_m)$

Algorithm 3.5: Encryption using HMCH

HMCH.$\mathcal{E}^T_{K,h}(P_1 \ldots, P_m)$

$\beta_1 \leftarrow E_K(T)$;
$\beta_2 \leftarrow x\beta_1$;
$M_1 \leftarrow P_1 \oplus \psi_h(P_2, \ldots, P_m)$;
$U_1 \leftarrow E_K(M_1)$;
$S \leftarrow M_1 \oplus U_1 \oplus \beta_1 \oplus \beta_2$;
for $i \leftarrow 2$ **to** m **do**
    $C_i \leftarrow P_i \oplus E_K(x^{i-2}\beta_1 \oplus S)$;
$C_1 \leftarrow U_1 \oplus \psi_h(C_2, \ldots, C_m)$;
return $(C_1, C_2, \ldots, C_m)$

3.4 Other Schemes

Though it has been argued that TES are the only modes which provide adequate security for the in place disk encryption application, over the years other kinds of schemes have also been proposed for this purpose, we provide a brief discussion on those schemes in this section.

3.4.1 Wide Block Block Ciphers

We said that a tweakable enciphering scheme provides security in the sense of a strong pseudorandom permutation, hence it can be seen as a block cipher which works on arbitrary long message blocks and is tweakable. The existing secure constructions of TES rely on the use of fixed block length block ciphers, thus to achieve the goal of an SPRP on long messages a TES uses multiple block cipher calls and the security of a TES can be suitably reduced to the security of the block cipher.

There have been attempts to build wide block-size block cipher from scratch, and such schemes have also been proposed for use in the disk encryption application. Important examples of these attempts are Bear and Lion, which were proposed by Ross Andersen and Eli Biham [2]. Both are very similar in construction except that Bear uses two keyed hash function rounds and one of a stream cipher whereas Lion uses only one keyed hash function round and two of stream ciphers. Current investigations have shown that they do not achieve adequate efficiencies required for the disk encryption application. A faster version of Bear called Beast[45] was another proposal, but it was still slow enough to be a real solution for disk encryption. Another block cipher designed for disk sector encryption is Mercy[14], unfortunately broken by Scott Fluhrer in 2001 [21].

Hence currently, to our knowledge, there are no standing proposals for wide block block ciphers constructed from scratch which can be considered suitable for disk encryption, hence, we do not consider such schemes in this study.

3.4.2 XTS

NIST has recently standardized a scheme called XTS for applications in secure storage [17]. XTS is a variant of the tweakable block cipher XEX proposed in [57]. XTS takes in two block-cipher keys, a tweak and a plaintext to output a ciphertext of the same length of that of the plaintext. For messages whose lengths are multiples of the block

length of the block cipher XTS works as follows shown in Algorithm 3.6. As evident from the description in Algorithm 3.6, XTS works as an electronic code book mode for a tweakable block cipher, and uses two keys K_1 and K_2.

Algorithm 3.6: Encryption and Decryption XTS

XTS-AES.$\mathcal{E}^T_{K_1,K_2}(P_1,\ldots,P_m)$

1 $\beta \leftarrow E_{K_2}(T)$
  for $i \leftarrow 1$ **to** m **do**
2 $\beta \leftarrow x\beta$
3 $PP \leftarrow P_i \oplus \beta$
4 $CC \leftarrow E_{K_1}(PP)$
5 $C_i \leftarrow CC \oplus \beta$
6 **return** $(C_1, C_2, \ldots, C_m)$

XTS-AES.$\mathcal{D}^T_{K_1,K_2}(C_1,\ldots,C_m)$

1 $\beta \leftarrow E_{K_2}(T)$
  for $i \leftarrow 1$ **to** m **do**
2 $\beta \leftarrow x\beta$
3 $CC \leftarrow C_i \oplus \beta$
4 $PP \leftarrow E_{K_1}(CC)$
5 $P_i \leftarrow PP \oplus \beta$
6 **return** $(P_1, P_2, \ldots, P_m)$

Though this is much simpler than any of the TES but it is doubtful whether it provides the required security for the disk encryption application. The XTS with AES as the underlying block cipher was first standardized by the IEEE working group on security in storage in the standard document P1619.1. But a public scrutiny of the standard resulted in numerous debates which voiced concerns about its security [42, 63]. It was acknowledged that simple mix and match type attacks could be mounted on XTS. The standard provides a security proof, which proves XTS to be a secure tweakable block cipher. But such a security guarantee may not be enough for the purpose of disk encryption. Though there have been some skepticism surrounding XTS but no concrete attack on XTS has been yet demonstrated. A trivial attack that can be mounted on XTS is described next. The attack described below is a mix and match attack.

An attack on XTS

Suppose an adversary knows about the encryption of two plain texts P, P', where $P = (P_0, P_1, \ldots, P_{m-1})$ and $P' = (P'_0, P'_1, \ldots, P'_{m-1})$ using the same tweak T. Let the ciphertexts corresponding to P and P' be $C = (C_0, C_1, \ldots, C_{m-1})$ and $C' = (C'_0, C'_1, \ldots, C'_{m-1})$ respectively.

Then the adversary can create a new cipher text $Y_1, Y_2, \ldots, Y_m$ where each $Y_i \in$

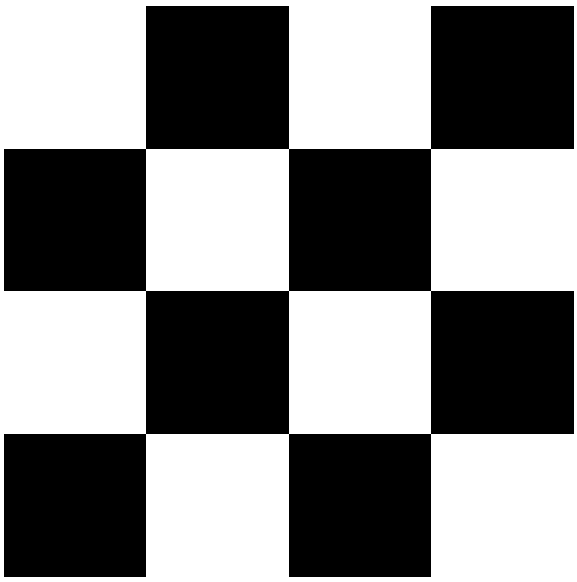


Figure 3.1: The decryption of the ciphertext generated by adversary $\mathcal{A}$

$\{C_i, C_i'\}$, which would get decrypted to $X_1, X_2, \ldots, X_m$ for the tweak T, where

$$X_i = \begin{cases} P_i & \text{if } Y_i = C_i \\ P_i' & \text{if } Y_i = C_i'. \end{cases}$$

This attack can be made more convincing with a concrete example. Consider the scenario that a raw image of size 64×64 of one byte per pixel is stored in a 4096 byte sector. The adversary have knowledge of the encryption of a white image, i.e., where each pixel has the value of 255 and a black image where each pixel have a value of 0, when stored in a specific sector. Thus, we are considering that $P = (P_0, P_1, \ldots, P_{m-1})$ and $P' = (P_0', P_1', \ldots, P_{m-1}')$ are the white and black image respectively and the adversary has knowledge of $C = (C_0, C_1, \ldots, C_{m-1})$ and $C' = (C_0', C_1', \ldots, C_{m-1}')$, where C and C' are the encryptions of P and P' using XTS with the same tweak T. If the block cipher of XTS has block length of 128 bits, then each P_i (and P_i') would contain sixteen pixels and the number of blocks m would be equal to 256. Now, the adversary $\mathcal{A}$ follows the following procedure to output Y.

Adversary $\mathcal{A}(C, C')$

1. **for** $k \leftarrow 1$ to $m - 1$,
2. $\quad j \leftarrow k \bmod 4$;
3. $\quad i \leftarrow \frac{k-j}{4}$;
4. $\quad p \leftarrow \lfloor \frac{i}{16} \rfloor$;

5. **if** $(j + p) \bmod 2 = 0$,
6. $Y_i \leftarrow C_i$;
7. **else** $Y_i \leftarrow C'_i$;
8. **end for**
9. **return** $Y \leftarrow (Y_0, \ldots, Y_{m-1})$

If Y is decrypted using the tweak T, the corresponding image would have a chess board pattern as shown in Figure 3.1 with alternate white and black blocks where each block would be 16×16 pixels. Thus, with the knowledge of two ciphertexts the adversary is able to construct a completely different ciphertext which decrypts to something meaningful. This is just one example, the adversary can be more artistic and generate ciphertexts corresponding to other images using C and C'. This kind of attack is not possible in TES as in a TES each block of ciphertext depends on each block of plaintext and vice versa.

This attack makes us skeptic about the security of XTS for disk encryption, but given that it is now part of a standard, we include this in our study and provide performance characteristics of this mode.

3.4.3 LRW

Another construction related to XTS is the LRW construction which is also a tweakable block cipher first proposed in [43]. P1619 was considering LRW as a candidate, but later rejected it. The draft in [40] gives a description of LRW-AES cipher. According to this draft, the encryption algorithm of LRW-AES takes as input a primary AES key, K_1 , a secondary key K_2 of 16 bytes, a $16-byte$ wide block P of plaintext and a $16-byte$ block T which represents the logical position or index of the block in relation to the beginning of the key scope (the value T acts as the tweak). The algorithm is shown in Algorithm 3.7. As shown in the Algorithm 3.7, in step 1 a finite field operation is required. The document [40] suggests an optimization in the computation in step 1, as it can be computed by reusing previously known information which can be stored in a pre-computed table. With this optimization, LRW would require only one field multiplication, 32 field additions and a few integer increments for computing 32 consecutive cipher blocks.

Though it seems that LRW is quite efficient, but like XTS, LRW is also not a TES, it is just a tweakable block cipher used in a ECB mode. Hence an attack similar to the one we described for XTS is applicable here. Additionally, it was observed that

Algorithm 3.7: Encryption LRW

LRW.$\mathcal{E}^T_{K_1,K_2}(P_1,\ldots,P_m)$
for $i \leftarrow 1$ **to** m **do**
1 $\beta \leftarrow K_2 I$;
2 $PP \leftarrow P_i \oplus \beta$;
3 $CC \leftarrow E_{K_1}(PP)$;
4 $T \leftarrow T + i$;
5 $C_i \leftarrow CC \oplus \beta$;
6 **return** $(C_1, C_2, \ldots, C_m)$

if a key is encrypted using LRW, then it leads to a complete break. This observation prompted its rejection from P1619.1. For these reasons we do not consider LRW in the current study.

3.4.4 BitLocker

BitLocker encrypts all the data on a specific operating system volume. The encryption offers only confidentiality of the data. In order to provide some kind of authentication of the data, BitLocker makes use of the TPM (Trusted Platform Module) chip mounted on the motherboard of most of the modern PCs. BitLocker also allows users to use a PIN or a cryptographic key contained in a USB device that the TPM checks, which serves as an authentication of a user. BitLocker uses AES-CBC mode for the primary encryption part plus a dedicated diffuser, named Elephant diffuser, to the plaintext side. The security of the Elephant diffuser is still unproven. The idea of the diffuser according to [19] is that even if the diffuser was broken, as the data is also encrypted using AES-CBC, the confidentiality will be ensured. The only task of the diffuser is to make manipulation attacks harder.

Encryption and decryption using BitLocker require four separate operations. Decryption applies them just in the other direction that encryption does. Next we describe the encryption process of BitLocker as is detailed in [19].

Specification of BitLoker Encryption

For encrypting, a plaintext is first xored with a sector key derived from the sector number, then run through two (unkeyed) diffusers and finally encrypted with AES in CBC mode. The encryption algorithm for BitLocker encryption is shown in Algorithm

3.8. Here, $E_K()$ represents the AES encryption function, and $e()$ an encoding function that maps a sector number s into a unique $16-byte$ value in which the first 8 bytes (in least significant byte first encoding) are the byte offset of the sector on the volume and the last 8 bytes are always zero. In the description of the Algorithm 3.8 first (in line 1) the sector key K_s is generated from the secret key K_sec as follows:

Algorithm 3.8: Encryption BitLocker

BitLocker.$\mathcal{E}^s_{K_{sec},K_{AES}}(S_1,\ldots,S_m)$

// Derive Sector Key K_s

1 $K_s \leftarrow E_{K_{sec}}(e(s))||E_{K_{sec}}(e'(s))||\ldots$;

2 $S \leftarrow (S_1||\ldots||S_m) \oplus K_s$;

3 $(s_1||\ldots||s_n) \leftarrow S$ // Each s_i is 32 bits

// Applying Diffuser A

for $i = n \cdot A_{cycles} - 1,\ldots,2,1,0$ **do**

4 $s_i \leftarrow s_i - (s_{i-2} \oplus (s_{i-5} <<< R^{(a)}_{i \bmod 4}))$;

// Applying Diffuser B

for $i = n \cdot B_{cycles} - 1,\ldots,2,1,0$ **do**

5 $s_i \leftarrow s_i - (s_{i+2} \oplus (s_{i+5} <<< R^{(b)}_{i \bmod 4}))$;

6 $S \leftarrow (s_i||\ldots||s_n)$;

// Each SS_i is 128 bits

7 $(SS_1||\ldots||S_m) \leftarrow S$;

// Finally AES-CBC

8 $C_0 \leftarrow IV_s$;

for $i \leftarrow 1$ **to** m **do**

9 $C_i \leftarrow E_{K_{AES}}(C_{i-1} \oplus SS_i)$;

10 **return** $(C_1, C_2, \ldots, C_m)$

$$K_s = E_{K_{sec}}(e(s))||E_{K_{sec}}(e'(s))$$

. $e'(s)$ is the same as $e(s)$ except that the last byte of the result has the value 128. K_s is repeated as many times as necessary to get a size equal to the size of the plaintext to be xored with it.

The xored plaintext is parsed through two diffusers as shown in lines 4 and 5. The diffusers uses two constants A_{cycles} and B_{cycles} which are fixed to 5 and 3 respectively. Additionally it uses some more constants stored in the arrays $R^{(a)}$ and $R^{(b)}$. Where $R^{(a)}$ and $R^{(b)}$ are fixed as $R^{(a)} = [9, 0, 13, 0]$ and $R^{(b)} = [0, 10, 0, 25]$. In lines 4 and 5 $a <<< b$ represents a rotated by b bits.

After parsing through the diffusers the transformed plaintext undergoes CBC encryption, with an initialization vector IV_s, which is computed as $IV_s = E_{K_{AES}}(e(s))$.

Though BitLocker has no provable guarantees for its security, it is still the only known scheme which has been widely deployed in commercial devices. Hence we do consider BitLocker in our performance study.

Summary

We gave a description of tweakable enciphering schemes, which is considered to be a main paradigm for disk encryption. Tweakable enciphering schemes for short TES, are block cipher modes of operation which provide security in the sense of a tweakable strong pseudo-random permutation (SPRP). Among all the several proposals for disk encryption schemes and many standardizing activities. In our study we considered all known TES efficient candidates.

We also described other kinds of schemes that have also been proposed for the purpose of disk encryption. XTS which has recently been standardized by NIST. LRW construction which was considered as a candidate also for standardization by P1619, and Bit-Locker which is part of Windows Vista and Windows 7 OS. Though several weaknesses of XTS we described a mix and match attack that can be mounted on XTS which has not appeared in the literature before.

Chapter 4

Two new Tweakable Enciphering Schemes

In this Chapter we propose modifications of two existing TES schemes HCTR and HMCH[BRW] and call the new schemes as HCTR* and HMCH2. The modifications leads to much efficient schemes. In both the constructions we would use a special type of polynomial called BRW polynomials [6]. The definition of a BRW polynomial was given in Section 3.3.3 (in page 29).

4.1 Description of HCTR* and HMCH2

HCTR* modifies HCTR in two different ways. Firstly, using the observations made in [60] we realized that the polynomial hash used in HCTR can be replaced by a BRW polynomial which can be computed using half the number of multiplications compared to an usual polynomial. Secondly, HCTR was designed for variable length messages and could support arbitrary (but fixed) length tweaks. As mentioned earlier, the functionality of variable length messages is not required for sector wise disk encryption, hence HCTR* only supports fixed length messages. For this restriction HCTR* unlike HCTR does not take in the length of the message as an input to the polynomial hash and can thus save some computation. The encryption and decryption algorithms for HCTR* are shown in Algorithm 4.9. These algorithms are self explanatory. For an m block message HCTR* requires $2(\lceil (m/2) \rceil + 1)$ multiplications, $\lfloor \lg m \rfloor$ squarings and m block cipher calls. This is a significant improvement over HCTR as the original construction required $(2m+2)$ multiplications and m block

cipher calls.

Algorithms 4.9: Encryption and Decryption using HCTR*

HCTR*.$\mathcal{E}^T_{K,h}(P_1 \ldots, P_m)$

1 $MM \leftarrow P_1 \oplus h\mathsf{BRW}_h(P_2, \ldots, P_m, T)$;
2 $CC \leftarrow E_K(MM)$;
3 $S \leftarrow MM \oplus CC$;
4 **for** $i \leftarrow 2$ **to** m **do**
5 $\quad C_i \leftarrow P_i \oplus E_K(S \oplus \mathsf{bin}_n(i))$;
6 $C_1 \leftarrow CC \oplus h\mathsf{BRW}_h(C_2, C_3, \ldots, C_m, T)$;
7 **return** $(C_1, \ldots, C_m)$

HCTR*.$\mathcal{D}^T_{K,h}(C_1 \ldots, C_m)$

1 $CC \leftarrow C_1 \oplus h\mathsf{BRW}_h(C_2, C_3, \ldots, C_m, T)$;
2 $MM \leftarrow E_K^{-1}(CC)$;
3 $S \leftarrow MM \oplus CC$;
4 **for** $i \leftarrow 2$ **to** m **do**
5 $\quad P_i \leftarrow C_i \oplus E_K(S \oplus \mathsf{bin}_n(i))$;
6 $P_1 \leftarrow MM \oplus h\mathsf{BRW}_h(P_2, \ldots, P_m, T)$;
7 **return** $(P_1, P_2, \ldots, P_m)$

The other construction HMCH2 is a modification of HMCH[BRW]. This modification is more subtle. The original HMCH[BRW] construction requires a series of *xtimes* operations (see Algorithm 3.5 in page 33). As stated earlier, if $A \in \mathbb{F}_{2^n}$ then A can be seen as a polynomial of degree less than n with coefficients in $\mathbb{F}_2$. We define $xtimes(A) = xA \bmod q(x)$, where $q(x)$ is the n degree irreducible polynomial representing the field $\mathbb{F}_{2^n}$. The operation *xtimes* can be performed very efficiently as this amounts to a shift and a conditional xor. For this efficiency it has been used in many other cryptographic constructions also, and previously it was considered that the computational cost of this operation is negligible compared to a block-cipher call or a multiplication. With the AES-NI support this consideration is no more true. Using 128 bit SIMD instructions, in a typical Intel machine *xtimes* requires about 6 cycles which is not negligible compared to an AES call. As with AES NI support, theoretically one AES round can be performed in one cycle if there are no dependencies between numerous calls. In our modification of HMCH[BRW] we eliminate the xtimes and thus come up with a new construction which enjoys the same security levels of that of HMCH[BRW].

The encryption and decryption algorithms for HMCH2 are shown Algorithm 4.10. The construction is similar to HCTR* but it handles the tweak in a different manner. The handling of tweak in HMCH2 is similar to that of HCH. Because of the typical way in which HMCH2 handles the tweak, it requires one more blockcipher call than HCTR* but the number of multiplications required is less.

Algorithms 4.10: Encryption and Decryption using HMCH2

HMCH2.$\mathcal{E}^T_{K,h}(P_1 \ldots, P_m)$

1 $\beta \leftarrow E_K(T)$;
2 $MM \leftarrow \beta \oplus P_1 \oplus h\mathsf{BRW}_h(P_2, \ldots, P_m)$;
3 $CC \leftarrow E_K(MM)$; $S \leftarrow MM \oplus CC$;
4 **for** $i \leftarrow 2$ **to** m **do**
5 $\quad C_i \leftarrow P_i \oplus E_K(S \oplus \mathsf{bin}_n(i-1))$;
6 $C_1 \leftarrow CC \oplus \beta \oplus h\mathsf{BRW}_h(C_2, \ldots, C_m)$;
7 **return** $(C_1, \ldots, C_m)$

HMCH2.$\mathcal{D}^T_{K,h}(C_1 \ldots, C_m)$

1 $\beta \leftarrow E_K(T)$;
2 $CC \leftarrow C_1 \oplus h\mathsf{BRW}_h(C_2, \ldots, C_m)$;
3 $MM \leftarrow E_K^{-1}(CC)$; $S \leftarrow MM \oplus CC$;
4 **for** $i \leftarrow 2$ **to** m **do**
5 $\quad P_i \leftarrow C_i \oplus E_K(S \oplus \mathsf{bin}_n(i-1))$;
6 $P_1 \leftarrow MM \oplus h\mathsf{BRW}_h(P_2, \ldots, P_m)$;
7 **return** $(P_1, P_2, \ldots, P_m)$

4.2 Security of the constructions

In this section we would prove that the minor modifications directed towards efficiency also give rise to secure schemes. To prove security of HCTR* and HMCH2, first we need to specify the security definition of a TES. In Section 4.2.1 we provide the formal description of an adversary which attacks a TES and then describe when a TES is considered to be secure. In Section 4.2.2 we state two theorems which specifies the security of HCTR* and HMCH2. In Section 4.2.3 we provide the proofs of these theorems.

4.2.1 Definitions and notation

An adversary A is a probabilistic algorithm which has access to some oracles and which outputs either 0 or 1. Oracles are written as superscripts. The notation $A^{\mathcal{O}_1,\mathcal{O}_2} \Rightarrow 1$ denotes the event that the adversary A, interacts with the oracles $\mathcal{O}_1, \mathcal{O}_2$, and finally outputs the bit 1. In what follows, by the notation $X \xleftarrow{\$} \mathcal{S}$, we will denote the event of choosing X uniformly at random from the finite set $\mathcal{S}$.

Let $\mathrm{Perm}(n)$ denote the set of all permutations on $\{0,1\}^n$. We require a block cipher $E : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ to be a strong pseudorandom permutation (SPRP). The advantage of an adversary A in breaking the strong pseudorandomness of $E(.,.)$ is defined in the following manner.

$$\begin{aligned}\mathbf{Adv}^{\pm\mathrm{prp}}_E(A) \;=\; &\Big|\Pr\Big[K \xleftarrow{\$} \mathcal{K} : A^{E_K(),E_K^{-1}()} \Rightarrow 1\Big] - \\ &\Pr\Big[\pi \xleftarrow{\$} \mathrm{Perm}(n) : A^{\pi(),\pi^{-1}()} \Rightarrow 1\Big]\Big|. \qquad (4.1)\end{aligned}$$

In Equation 4.1 above the advantage of an adversary A in breaking the strong pseu-

dorandomness of E is defined as a difference of probabilities of two different experiments. In the first experiment the adversary is provided with the oracles $E_K(\,), E_K^{-1}(\,)$ for a randomly chosen K in the key space. In the second experiment, A is provided with the oracles $\pi(\,), \pi^{-1}(\,)$ where π is chosen uniformly at random from $\mathrm{Perm}(n)$. The goal of the adversary is to distinguish between these two scenarios. Thus, if the difference between these two probabilities is very small then we say that with high probability A cannot distinguish between the block cipher and an uniform random permutation. Thus $E(.,.)$ is considered secure in the sense of an SPRP if for all efficient adversaries A, $\mathbf{Adv}_E^{\pm\mathrm{prp}}(A)$ is small.

As defined in Section 3.3 (in page 26), a tweakable enciphering scheme is a function $\mathbf{E} : \mathcal{K} \times \mathcal{T} \times \mathcal{M} \to \mathcal{M}$, 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 $\mathcal{M}$. For both HCTR* and HMCH2 we have $\mathcal{M} = \{0,1\}^{nm}$, where m is a fixed integer and n the block length of the underlying block cipher. We shall 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 $\mathbf{E}_K^T(X) = Y$.

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

$$\begin{aligned}\mathbf{Adv}_{\mathbf{E}}^{\pm\widetilde{\mathrm{prp}}}(A) \;=\; & \Big|\Pr\left[K \xleftarrow{\$} \mathcal{K} : A^{\mathbf{E}_K(.,.),\mathbf{E}_K^{-1}(.,.)} \Rightarrow 1\right] \\ & - \Pr\left[\boldsymbol{\pi} \xleftarrow{\$} \mathrm{Perm}^{\mathcal{T}}(\mathcal{M}) : A^{\boldsymbol{\pi}(.,.),\boldsymbol{\pi}^{-1}(.,.)} \Rightarrow 1\right]\Big|. \qquad (4.2)\end{aligned}$$

We define $\mathbf{Adv}_{\mathbf{E}}^{\pm\widetilde{\mathrm{prp}}}(q, \sigma)$ by $\max_A \mathbf{Adv}_{\mathbf{E}}^{\pm\widetilde{\mathrm{prp}}}(A)$ where maximum is taken over all adversaries which makes at most q queries having at most σ many blocks. For a computational advantage we define $\mathbf{Adv}_{\mathbf{E}}^{\pm\widetilde{\mathrm{prp}}}(q, \sigma, t)$ by $\max_A \mathbf{Adv}_{\mathbf{E}}^{\pm\widetilde{\mathrm{prp}}}(A)$. In addition to the previous restrictions on A, he can run in time at most t.

Pointless queries: Let T, P and C represent tweak, plaintext and ciphertext respectively. 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.

The notation $\mathbf{E}[E]$ denotes a tweakable enciphering scheme $\mathbf{E}$, where the n-bit block cipher E is used in the manner specified in $\mathbf{E}$. We will use the notation $\mathbf{E}_\pi$ as a shorthand for $\mathbf{E}[\mathrm{Perm}(n)]$ and $\mathbf{D}_\pi$ will denote the inverse of $\mathbf{E}_\pi$. Thus, the notation $A^{\mathbf{E}_\pi, \mathbf{D}_\pi}$ will denote an adversary interacting with the oracles $\mathbf{E}_\pi$ and $\mathbf{D}_\pi$. Note, in our case $\mathbf{E}$ is either HCTR* or HMCH2.

4.2.2 Statement of the results

The following theorem specifies the security of HCTR* and HMCH2.

Theorem 1. *Fix n, σ to be positive integers and an n-bit block cipher $E : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$. Then*

$$\mathbf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathrm{HCTR}^*[\mathrm{Perm}(n)]}(\sigma) \leq \frac{7.5\sigma^2}{2^n}. \tag{4.3}$$

$$\mathbf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathrm{HCTR}^*[E]}(\sigma, t) \leq \frac{7.5\sigma^2}{2^n} + \mathbf{Adv}^{\pm\mathrm{prp}}_{E}(\sigma, t') \tag{4.4}$$

where $t' = t + O(\sigma)$.

Theorem 2. *Fix n, σ to be positive integers and an n-bit block cipher $E : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$. Then*

$$\mathbf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathrm{HMCH2}[\mathrm{Perm}(n)]}(\sigma) \leq \frac{11.5\sigma^2}{2^n}. \tag{4.5}$$

$$\mathbf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathrm{HMCH2}[E]}(\sigma, t) \leq \frac{11.5\sigma^2}{2^n} + \mathbf{Adv}^{\pm\mathrm{prp}}_{E}(\sigma, t') \tag{4.6}$$

where $t' = t + O(\sigma)$.

The transitions from (4.3) to (4.4) and from (4.5) to (4.6) are standard, we provide proofs of (4.3) and (4.5) in the next section. The proofs use the standard technique of sequence of games, and are similar to the proof in [11].

4.2.3 Proofs

For proving (4.3) and (4.5), it would be easier if we consider an adversary's advantage in distinguishing a tweakable enciphering scheme $\mathbf{E}$ from an oracle which simply returns random bit strings. This advantage is defined in the following manner.

$$\begin{aligned}\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathbf{E}[\mathrm{Perm}(n)]}(A) &= \Big|\Pr\left[\pi \xleftarrow{\$} \mathrm{Perm}(n) : A^{\mathbf{E}_\pi,\mathbf{D}_\pi} \Rightarrow 1\right] \\ &\quad - \Pr\left[A^{\$(.,.),\$(.,.)} \Rightarrow 1\right]\Big| \qquad (4.7)\end{aligned}$$

where $\$(.,M)$ or $\$(.,C)$ returns independently distributed random bits of length $|M|$ or $|C|$ respectively. Thus, we have

$$\begin{aligned}\mathbf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathbf{E}[\mathrm{Perm}(n)]}(A) &= \Big(\Pr\left[\pi \xleftarrow{\$} \mathrm{Perm}(n) : A^{\mathbf{E}_\pi,\mathbf{D}_\pi} \Rightarrow 1\right] \\ &\quad - \Pr\left[\boldsymbol{\pi} \xleftarrow{\$} \mathrm{Perm}^{\mathcal{T}}(\mathcal{M}) : A^{\boldsymbol{\pi}(.,.),\boldsymbol{\pi}^{-1}(.,.)} \Rightarrow 1\right]\Big) \\ &= \Big(\Pr\left[\pi \xleftarrow{\$} \mathrm{Perm}(n) : A^{\mathbf{E}_\pi,\mathbf{D}_\pi} \Rightarrow 1\right] \\ &\quad - \Pr\left[A^{\$(.,.),\$(.,.)} \Rightarrow 1\right]\Big) \\ &\quad + \Big(\Pr\left[A^{\$(.,.),\$(.,.)} \Rightarrow 1\right] \\ &\quad - \Pr\left[\boldsymbol{\pi} \xleftarrow{\$} \mathrm{Perm}^{\mathcal{T}}(\mathcal{M}) : A^{\boldsymbol{\pi}(.,.),\boldsymbol{\pi}^{-1}(.,.)} \Rightarrow 1\right]\Big) \\ &\leq \mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathbf{E}[\mathrm{Perm}(n)]}(A) + \binom{q}{2}\frac{1}{2^n} \qquad (4.8)\end{aligned}$$

where q is the number of queries made by the adversary. The last inequality follows from the fact that the probability with which an adversary (which makes at most q valid queries) can distinguish between a random permutation and an oracle which just returns random strings would be the same as the collition probability of q random strings. As the collition probability of q random n-bit strings is $\binom{q}{2}\frac{1}{2^n}$ hence the inequality follows. For a more detailed proof see [30].

Thus, the main task for proving Theorems 1 and 2 reduces to obtaining an upper bound on $\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathrm{HCTR}^*[\mathrm{Perm}(n)]}(\sigma)$ and $\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathrm{HMCH2}[\mathrm{Perm}(n)]}(\sigma)$ respectively. In the next two sections we bound these advantages. The Theorems follows from these computed bounds and Equation (4.8).

Upper bound on $\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathrm{HCTR}^*[\mathbf{Perm}(n)]}(\sigma)$

To derive an upper bound on $\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathrm{HCTR}^*[\mathrm{Perm}(n)]}(\sigma)$ we follow the technique of sequence of games.

Game HCTR*1: In HCTR*1, the adversary interacts with $\mathbf{E}_\pi$ and $\mathbf{D}_\pi$ where π is a randomly chosen permutation from Perm(n). Instead of initially choosing π, we build up π in the following manner.

Initially π it is assumed that for all $X \in \{0,1\}^n$, $\pi(X)$ is undefined. When $\pi(X)$ is needed, but the value of π is not yet defined at X, then a random value is chosen among the available range values. Similarly when $\pi^{-1}(Y)$ is required and there is no X yet defined for which $\pi(X) = Y$, we choose a random value for $\pi^{-1}(Y)$ from the available domain values.

The domain and range of π are maintained in two sets *Domain* and *Range*, and $\overline{Domain}$ and $\overline{Range}$ are the complements of *Domain* and *Range* relative to $\{0,1\}^n$. The game HCTR*1 is shown in Figure 4.1.

The game HCTR*1 accurately represents the attack scenario, and by our choice of notation, we can write

$$\Pr[A^{\mathbf{E}_\pi,\mathbf{D}_\pi} \Rightarrow 1] = \Pr[A^{\text{HCTR}^*1} \Rightarrow 1]. \tag{4.9}$$

Game RAND1: We modify HCTR*1 by deleting the boxed entries in HCTR*1 and call the modified game as RAND1. By deleting the boxed entries it cannot be guaranteed that π is a permutation as though we do the consistency checks but we do not reset the values of Y (in Ch-π) and X (in Ch-π^{-1}). Thus, the games HCTR*1 and RAND1 are identical apart from what happens when the bad flag is set. By using the result from [5], we obtain

$$|\Pr[A^{\text{HCTR}^*1} \Rightarrow 1] - \Pr[A^{\text{RAND1}} \Rightarrow 1]| \leq \Pr[A^{\text{RAND1}} \text{ sets } \mathsf{bad}] \tag{4.10}$$

Another important thing to note is that in RAND1 in line 103, for a encryption query CC^s (and MM^s for a decryption query) gets set to a random n bit string. Similarly 105 and 108 Z_i^s gets set to random values. Thus the adversary gets random strings in response to both her encryption and decryption queries. Hence,

$$\Pr[A^{\text{RAND1}} \Rightarrow 1] = \Pr[A^{\$(.,.),\$(.,.)} \Rightarrow 1] \tag{4.11}$$

So using Equations (4.7), (4.10) and (4.11) we get

$$\begin{aligned}
\mathbf{Adv}^{\pm\text{rnd}}_{\text{HCTR}^*[\text{Perm}(n)]}(A) &= |\Pr[A^{\mathbf{E}_\pi,\mathbf{D}_\pi} \Rightarrow 1] - \Pr[A^{\$(.,.),\$(.,.)} \Rightarrow 1]| && (4.12)\\
&= |\Pr[A^{\text{HCTR}^*1} \Rightarrow 1] - \Pr[A^{\text{RAND1}} \Rightarrow 1]| \\
&\leq \Pr[A^{\text{RAND1}} \text{ sets } \mathsf{bad}] && (4.13)
\end{aligned}$$

Game RAND2: Now we make some subtle changes in the game RAND1 to get a new game RAND2 which is described in Figure 4.2. In game RAND1 the permutation was not maintained and a call to the permutation was responded by returning random strings, so in Game RAND2 we no more use the subroutines Ch-π and Ch-π^{-1}. Here we immediately return random strings to the adversary in response to his encryption or decryption queries. Later in the finalization step we adjust variables and maintain multi sets $\mathcal{D}$ and $\mathcal{R}$ where we list the elements that were supposed to be inputs and outputs of the permutation. In the second phase of the finalization step, we check for collisions in the sets $\mathcal{D}$ and $\mathcal{R}$, and in the event of a collision we set the bad flag to true.

Game RAND1 and Game RAND2 are indistinguishable to the adversary, as in both cases he gets random strings in response to his queries. Also, the probability with which RAND1 sets bad is same as the probability with which RAND2 sets bad. Thus we get:

$$\Pr[A^{\mathrm{RAND1}} \text{ sets } \mathsf{bad}] = \Pr[A^{\mathrm{RAND2}} \text{ sets } \mathsf{bad}] \tag{4.14}$$

Thus from Equations (4.13) and (4.14) we obtain

$$\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathrm{HCTR}^*[\mathrm{Perm}(n)]}(A) \leq \Pr[A^{\mathrm{RAND2}} \text{ sets } \mathsf{bad}] \tag{4.15}$$

Now our goal would be to bound $\Pr[A^{\mathrm{RAND2}} \text{ sets } \mathsf{bad}]$. We notice that in Game RAND2 the bad flag is set when there is a collision in either of the sets $\mathcal{D}$ or $\mathcal{R}$. So if COLLD and COLLR denote the events of a collision in $\mathcal{D}$ and $\mathcal{R}$ respectively then we have

$$\Pr[A^{\mathrm{RAND2}} \text{ sets } \mathsf{bad}] \leq \Pr[\mathsf{COLLR}] + \Pr[\mathsf{COLLD}]$$

In many previously reported game based proofs for strong pseudorandom permutations as in [64, 31, 13], the final collision analysis is done on a non-interactive game. The non-interactive game is generally obtained by eliminating the randomness present in the distribution of the queries presented by the adversary. To achieve this, the final non-interactive game runs on a fixed transcript which maximizes the probability of bad being set to true. In our case as we will soon see, such a de-randomization is not required. Because of the specific structure of the game RAND2 the probability COLLR and COLLD would be independent of the distribution of the queries supplied by the adversary, hence a final collision analysis can be done on the game RAND2 itself.

Our goal is to bound the probability that two formal variables in the sets $\mathcal{D}$ and $\mathcal{R}$ take the same value. After q queries of the adversary where the s^{th} query has m^s blocks of plaintext or ciphertext and t block of tweak, then the sets $\mathcal{D}$ and $\mathcal{R}$ can be written as follows:

Elements in $\mathcal{D}$:
$$MM^s = P_1^s \oplus Q^s,$$
$$S_j^s = S^s \oplus \mathsf{bin}_n(j) = (P_1^s \oplus C_1^s) \oplus (Q^s \oplus B^s \oplus \mathsf{bin}_n(j)),$$
where $Q^s = h\mathsf{BRW}_h(P_2^s \parallel \cdots \parallel P_m^s \parallel T^s)$ and
$B^s = h\mathsf{BRW}_h(C_2^s \parallel \cdots \parallel C_m^s \parallel T^s)$,
$1 \leq s \leq q$, $1 \leq j \leq m-1$,

Elements in $\mathcal{R}$:
$$CC^s = C_1^s \oplus B^s,$$
$$Y_i^s = C_i^s \oplus P_i^s,$$
$2 \leq i \leq m$, $1 \leq s \leq q$.

Before we present the collision analysis let us identify the random variables based on which the probability of collision would be computed. In game RAND2 the hash key h is selected uniformly from the set $\{0,1\}^n$. The outputs that the adversary receives are also uniformly distributed, and are independent of the previous queries supplied by the adversary and the outputs obtained by the adversary. The i^{th} query supplied by the adversary may depend on the previous outputs obtained by the adversary, but as the output of game RAND2 is not dependent in any way on the hash key h thus the queries supplied by the adversary are independent of h.

Also we would require two important properties of BRW polynomial which are easy to prove from the definition of BRW polynomials, which we state next.

Lemma 1. *For $m \geq 3$, and $X = (X_1, X_2, \ldots, X_m)$, $\mathsf{BRW}_h(X_1, X_2, \ldots, X_m)$ is a monic polynomial of degree at most $2m$ over $GF(2^n)$*

Lemma 2. *Let $m \geq 3$, and $X = (X_1, X_2, \ldots, X_m), X' = (X'_1, X'_2, \ldots, X'_m) \in [GF(2^n)]^m$ such that $X \neq X'$.*

Let $Y = h\mathsf{BRW}_h(X_1, X_2, \ldots, X_m)$ and $Y' = h\mathsf{BRW}_h(X'_1, X'_2, \ldots, X'_m)$. Then for every fixed $a \in GF(2^n)$

$$\Pr[Y \oplus Y' = a] \leq \frac{(2m+1)}{2^n}.$$

The probability is taken over the random choice of h.

See [60] for more discussion on the above stated properties of BRW polynomials.

Now, we are ready to provide the collision analysis. We first provide the collision analysis for $\mathcal{R}$:

- First let us consider the collision between CC^s. Let $s \neq s'$, then as $CC^s \oplus CC^{s'}$ is a nonzero polynomial of degree at most $2m+1$, hence, $\Pr[CC^s = CC^{s'}] \leq (2m+1)/2^n$, where the probability is taken over the uniform choice of h from $\{0,1\}^n$. Thus,

$$\Pr[CC^s = CC^{s'} : \text{for some } 1 \leq s < s' \leq q] \;\leq\; \binom{q}{2}\frac{2m+1}{2^n}. \quad (4.16)$$

- Similarly we can compute collision probability between Y_i^s and $CC^{s'}$. For each s', there are $(m-1)q$ many Y_i^s's. For each such choice, $\Pr[CC^{s'} = Y_i^s] \leq (2m-1)/2^n$. Thus,

$$\begin{aligned}\Pr[CC^{s'} = Y_i^s :\ \text{for some } 1 \leq s \;&\neq\; s' \leq q, 2 \leq i \leq m] \\ &\leq\; \frac{q(mq-q)(2m-1)}{2^n} \\ &\leq\; 2m^2q^2/2^n. \end{aligned} \quad (4.17)$$

- Now we consider collision among Y_i^s, $2 \leq i \leq m$, $1 \leq s \leq q$. For the pairs $(Y_i^s, Y_{i'}^{s'})$ with $s' \leq s$ and $(s,i) \neq (s',i')$, the collision probability is $1/2^n$, since either P^s or C^s is chosen uniformly and independently from the rest of the variables. There are $\binom{mq-q}{2}$ pairs of this form. Thus,

$$\begin{aligned}\Pr[Y_i^s = Y_{i'}^{s'} :\ \text{for some } 1 \leq s \leq s' \leq q, 1 \;&\leq\; i, i' \leq q, (s,i) \neq (s',i')] \\ &\leq\; \binom{mq-q}{2}/2^n. \end{aligned} \quad (4.18)$$

Combining equation (4.16), (4.17) and (4.18) we obtain

$$\Pr[\text{COLLR}] \leq \frac{3.5m^2q^2}{2^n}. \quad (4.19)$$

Now we consider collision in domain $\mathcal{D}$.

- Similar to equations (4.16) and (4.17), we have

$$\Pr[MM^s = MM^{s'} :\ \text{for some } 1 \leq s < s' \leq q] \;\leq\; \binom{q}{2}\frac{2m+1}{2^n}. \quad (4.20)$$

$$\Pr[MM^{s'} = S_i^s :\ \text{for some } 1 \leq s \neq s' \leq q, 2 \leq i \leq m^s] \leq 2m^2q^2/2^n. \quad (4.21)$$

- Now we consider collision among $S_i^s = S^s \oplus \mathsf{bin}_n(i)$, $2 \leq i \leq m^s$, $1 \leq s \leq q$. Note that, $S_i^s = S_{i'}^{s'}$ implies that $(P_1^s \oplus C_1^s) \oplus (Q^s \oplus B^s \oplus \mathsf{bin}_n(i)) = (P_1^{s'} \oplus C_1^{s'}) \oplus (Q^{s'} \oplus B^{s'} \oplus \mathsf{bin}_n(i'))$. Let $s' \leq s$ and $(s,i) \neq (s',i')$. Thus, either C_1^s (in case s^{th} query is encryption) or P_1^s (in case s^{th} query is decryption) is uniformly and independently distributed with all other variables stated in the above equality. Thus, the collision probability is $1/2^n$. Since there are $\binom{mq-q}{2}$ pairs of this form, we have

$$\begin{aligned} \Pr[S_i^s = S_{i'}^{s'} : \text{ for some } 1 \leq s \leq s' \leq q, 1 &\leq i, i' \leq q, (s,i) \neq (s',i')] \\ &\leq \binom{mq-q}{2}/2^n. \end{aligned} \tag{4.22}$$

The equations (4.20), (4.21) and (4.22) imply the following similar bound for domain collision probability.

$$\Pr[\text{COLLD}] \leq \frac{3.5m^2q^2}{2^n}. \tag{4.23}$$

Combining the domain and range collision probabilities, we obtain the probability of bad being set to true in RAND2 to be at most $7m^2q^2/2^n$. Thus, by using equations (4.19) and (4.23), we have

$$\mathbf{Adv}^{\pm\text{rnd}}_{\text{HCTR}^*[\text{Perm}(n)]}(A) \leq \frac{7m^2q^2}{2^n}. \tag{4.24}$$

Note, for an adversary making q queries each of a m block message an one block tweak, the query complexity $\sigma = mq + q$. Hence,

$$\mathbf{Adv}^{\pm\text{rnd}}_{\text{HCTR}^*[\text{Perm}(n)]}(A) \leq \frac{7\sigma^2}{2^n}. \tag{4.25}$$

Upper bound on $\mathbf{Adv}^{\pm\text{rnd}}_{\text{HMCH2}[\mathbf{Perm}(n)]}(\sigma)$

The derivation of the bound for $\mathbf{Adv}^{\pm\text{rnd}}_{\text{HMCH2}[\text{Perm}(n)]}(\sigma)$ is very similar to the derivation that we presented in the previous section. Using the same technique of sequence of games we can obtain the final game G2 as described in Figure 4.3. Which is very similar to the game in Figure 4.2.

Following the arguments as in the previous proof, we can say that

$$\begin{aligned} \mathbf{Adv}^{\pm\text{rnd}}_{\text{HMCH2}[\text{Perm}(n)]}(A) &\leq \Pr[A^{G2} \text{ sets } \mathsf{bad}] \\ &\leq \Pr[\mathsf{COLLD1}] + \Pr[\mathsf{COLLR1}], \end{aligned} \tag{4.26}$$

where $\mathsf{COLLD1}$ and $\mathsf{COLLR1}$ are the events of collision in the multisets $\mathcal{D}_1$ and $\mathcal{R}_1$ respectively as described in Figure 4.3.

The elements in the multisets $\mathcal{D}_1$ and $\mathcal{R}_1$ after q queries by the adversary would be as

Elements in $\mathcal{D}_1$: T^s, $MM^s = P_1^s \oplus Q^s \oplus \beta^s$,
$S_j^s = S^s \oplus \mathsf{bin}_n(j) = (P_1^s \oplus C_1^s) \oplus (Q^s \oplus B^s \oplus \mathsf{bin}_n(j))$,
where $Q^s = h\mathsf{BRW}_h(P_2^s \parallel \cdots \parallel P_m^s)$ and
$B^s = h\mathsf{BRW}_h(C_2^s \parallel \cdots \parallel C_m^s)$,
$1 \leq s \leq q$, $1 \leq j \leq m-1$,

Elements in $\mathcal{R}_1$: β^s, $CC^s = C_1^s \oplus B^s \oplus \beta^s$,
$Y_i^s = C_i^s \oplus P_i^s$,
$2 \leq i \leq m$, $1 \leq s \leq q$.

We observe that $\mathcal{D}_1 = \mathcal{D} \cup \{T^s : 1 \leq s \leq q\}$ and $\mathcal{R}_1 = \mathcal{R} \cup \{\beta^s : 1 \leq s \leq q\}$, where $\mathcal{D}$ and $\mathcal{R}$ are the multisets described in Figure 4.2. But the definition of MM^s in $\mathcal{D}_1$ is a bit different from that in $\mathcal{D}$, similarly CC^s in $\mathcal{R}_1$ and $\mathcal{R}$ are different. But this difference does not increase the collision probabilities involving these quantities that were presented in the previous section. So,

$$\Pr[\mathsf{COLLD1}] \leq \Pr[\mathsf{COLLD}] + \Pr[\mathsf{ED}] \tag{4.27}$$

and

$$\Pr[\mathsf{COLLR1}] \leq \Pr[\mathsf{COLLR}] + \Pr[\mathsf{ER}] + \Pr[\mathsf{EB}], \tag{4.28}$$

where ED is the event of a collision between T^s and the elements of $\mathcal{D}$, ER is the event of a collision between β^s and the elements of $\mathcal{R}$ and EB is the event of collision between the β^ss themselves. Note that in $\mathcal{D}_1$ there is never a collision between T^s and $T^{s'}$ because of the way the game G2 runs. Noting that the degree of the polynomial $h\mathsf{BRW}_h(P_2, \ldots, P_m)$ is at most $2m-1$ and following same arguments in the previous section we get

$$\Pr[MM^s = T^{s'} : \text{ for some } 1 \leq s, s' \leq q] \;\leq\; \frac{q^2(2m-1)}{2^n}. \tag{4.29}$$

$$\begin{aligned}\Pr[T^s = S_i^{s'} : \text{ for some } 1 \le s, s' \le q, 1 &\le i \le m-1] \\ &\le \frac{q(mq-q)}{2^n}. \end{aligned} \tag{4.30}$$

Thus, from equations (4.29) and (4.30) we have

$$\Pr[\mathsf{ED}] \le \frac{q^2(2m-1)}{2^n} + \frac{q(mq-q)}{2^n}. \tag{4.31}$$

As β^s is a uniform random element in $\{0,1\}^n$ and there are at most q many β^s in $\mathcal{R}_1$, so

$$\Pr[\mathsf{EB}] \le \binom{q}{2}\frac{1}{2^n}. \tag{4.32}$$

And as there are $(m-1)q+q$ many elements in $\mathcal{R}$, hence

$$\Pr[\mathsf{ER}] \le \frac{mq^2}{2^n}. \tag{4.33}$$

Thus, putting all together we have

$$\mathbf{Adv}^{\pm\text{rnd}}_{\text{HMCH2[Perm}(n)]}(A) \le \frac{11m^2q^2}{2^n} \le \frac{11\sigma^2}{2^n} \tag{4.34}$$

as desired.

Summary

We described the two new schemes HCTR*, HCTR*. HCTR* modifies HCTR in two different ways. Firstly, the polynomial hash used in HCTR can be replaced by a BRW polynomial which can be computed using half the number of multiplications compared to an usual polynomial. Secondly, HCTR was designed for variable length messages and could support arbitrary (but fixed) length tweaks. As mentioned earlier, the functionality of variable length messages is not required for sector wise disc encryption.

The other construction HMCH2 is a modification of HMCH[BRW]. This modification is more subtle. The original HMCH[BRW] construction requires a series of *xtimes* operations, For this efficiency we eliminate the xtimes and thus come up with a new construction which enjoys the same security levels of that of HMCH[BRW].

We proved that these minor modifications directed towards efficiency also give rise to secure schemes. After adapt the original proofs we successfully demonstrate that these modifications are also provably secure.

Figure 4.1: Games HCTR*1 and RAND1

Subroutine Ch-$\pi(X)$

11. $Y \xleftarrow{\$} \{0,1\}^n$; **if** $Y \in \mathit{Range}$ **then** bad $\leftarrow$ true; $\boxed{Y \xleftarrow{\$} \overline{\mathit{Range}}}$; **endif**;
12. **if** $X \in \mathit{Domain}$ **then** bad $\leftarrow$ true; $\boxed{Y \leftarrow \pi(X)}$; **endif**
13. $\pi(X) \leftarrow Y$; $\mathit{Domain} \leftarrow \mathit{Domain} \cup \{X\}$; $\mathit{Range} \leftarrow \mathit{Range} \cup \{Y\}$; **return**$(Y)$;

Subroutine Ch-$\pi^{-1}(Y)$

14. $X \xleftarrow{\$} \{0,1\}^n$; **if** $X \in \mathit{Domain}$, bad $\leftarrow$ true; $\boxed{X \xleftarrow{\$} \overline{\mathit{Domain}}}$; **endif**;
15. **if** $Y \in \mathit{Range}$ **then** bad $\leftarrow$ true; $\boxed{X \leftarrow \pi^{-1}(Y)}$; **endif**;
16. $\pi(X) \leftarrow Y$; $\mathit{Domain} \leftarrow \mathit{Domain} \cup \{X\}$; $\mathit{Range} \leftarrow \mathit{Range} \cup \{Y\}$; **return**$(X)$;

Initialization:

17. **for** all $X \in \{0,1\}^n$ $\pi(X) = \mathsf{undef}$ **endfor**
18. bad = false

Respond to the s^{th} query as follows: (Assume all queries are m blocks long

Encipher query: $\mathsf{Enc}(T^s; P_1^s, P_2^s, \ldots P_m^s)$

101. $MM^s \leftarrow P_1^s \oplus h\mathsf{BRW}_h(P_2^s || \ldots || P_m^s || T^s)$;
102. $CC^s \leftarrow$ Ch-$\pi(MM^s)$;
103. $S^s \leftarrow MM^s \oplus CC^s$;
104. **for** $i = 1$ to $m-1$,
105. $\quad Z_i^s \leftarrow$ Ch-$\pi(S^s \oplus \mathsf{bin}_n(i))$;
106. $\quad C_{i+1}^s \leftarrow P_{i+1}^s \oplus Z_i^s$;
107. **end for**
108. $C_1^s \leftarrow CC^s \oplus h\mathsf{BRW}_h(C_2^s || \ldots || C_m^s || T^s)$;
109. **return** $C_1^s || C_2^s || \ldots || C_m^s$

Decipher query: $\mathsf{Dec}(C_1^s, C_2^s, \ldots, C_m^s, T^s)$

$CC^s \leftarrow C_1^s \oplus h\mathsf{BRW}_h(C_2^s || \ldots || C_m^s || T^s)$;
$MM^s \leftarrow$ Ch-$\pi^{-1}(CC^s)$
$S^s \leftarrow MM^s \oplus CC^s$;
for $i = 1$ to $m-1$,
$\quad Z_i^s \leftarrow$ Ch-$\pi(S^s \oplus \mathsf{bin}_n(i))$;
$\quad P_{i+1}^s \leftarrow C_{i+1}^s \oplus Z_i^s$;
end for
$P_1^s \leftarrow MM^s \oplus h\mathsf{BRW}_h(P_2^s || \ldots || P_m^s || T^s)$;

return $P_2^s || \ldots || P_{m^s}^s$

Figure 4.2: Game RAND2

Respond to the s^{th} adversary query as follows:

ENCIPHER QUERY $\mathsf{Enc}(T^s; P_1^s, P_2^s, \ldots, P_m^s)$

$ty^s = \mathsf{Enc}$; $C_1^s||C_2^s||\ldots||C_m^s \xleftarrow{\$} \{0,1\}^{nm}$;

return $C_1^s||C_2^s||\ldots||C_m^s$;

DECIPHER QUERY $\mathsf{Dec}(T^s; C_1^s, C_2^s, \ldots, C_m^s)$

$ty^s = \mathsf{Dec}$; $P_1^s||P_2^s||\ldots||P_m^s \xleftarrow{\$} \{0,1\}^{nm}$;

return $P_1^s||P_2^s||\ldots||P_m^s$;

Finalization:

Case $ty^s = \mathsf{Enc}$:

$MM^s \leftarrow P_1^s \oplus h\mathsf{BRW}_h(P_2^s||\ldots||P_m^s||T^s)$;
$CC^s \leftarrow C_1^s \oplus h\mathsf{BRW}_h(C_2^s||\ldots||C_m^s||T^s)$;
$S^s \leftarrow MM^s \oplus CC^s$;
$\mathcal{D} \leftarrow \mathcal{D} \cup \{MM^s\}$; $\mathcal{R} \leftarrow \mathcal{R} \cup \{CC^s\}$;
for $i = 2$ to m,
  $Y_i^s \leftarrow C_i^s \oplus P_i^s$;
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{S^s \oplus \mathsf{bin}_n(i-1)\}$;
  $\mathcal{R} \leftarrow \mathcal{R} \cup \{Y_i^s\}$;
end for

Case $ty^s = \mathsf{Dec}$:

$MM^s \leftarrow P_1^s \oplus h\mathsf{BRW}_h(P_2^s||\ldots||P_m^s||T^s)$;
$CC^s \leftarrow C_1^s \oplus h\mathsf{BRW}_h(C_2^s||\ldots||C_m^s||T^s)$;
$S^s \leftarrow MM^s \oplus CC^s$;
$\mathcal{D} \leftarrow \mathcal{D} \cup \{MM^s\}$; $\mathcal{R} \leftarrow \mathcal{R} \cup \{CC^s\}$;
for $i = 2$ to m,
  $Y_i^s \leftarrow C_i^s \oplus P_i^s$;
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{S^s \oplus \mathsf{bin}_n(i-1)\}$;
  $\mathcal{R} \leftarrow \mathcal{R} \cup \{Y_i^s\}$;
end for

SECOND PHASE

$\mathsf{bad} \leftarrow \mathsf{false}$;
if (some value occurs more than once in $\mathcal{D}$) **then** $\mathsf{bad} \leftarrow \mathsf{true}$ **endif**;
if (some value occurs more than once in $\mathcal{R}$) **then** $\mathsf{bad} \leftarrow \mathsf{true}$ **endif**.

Figure 4.3: Game G2

Respond to the s^{th} adversary query as follows:

ENCIPHER QUERY $\mathsf{Enc}(T^s; P_1^s, P_2^s, \ldots, P_m^s)$
$ty^s = \mathsf{Enc}$; $C_1^s||C_2^s||\ldots||C_m^s \xleftarrow{\$} \{0,1\}^{nm}$;
return $C_1^s||C_2^s||\ldots||C_m^s$;

DECIPHER QUERY $\mathsf{Dec}(T^s; C_1^s, C_2^s, \ldots, C_m^s)$
$ty^s = \mathsf{Dec}$; $P_1^s||P_2^s||\ldots||P_m^s \xleftarrow{\$} \{0,1\}^{nm}$;
return $P_1^s||P_2^s||\ldots||P_m^s$;

Finalization:

Case $ty^s = \mathsf{Enc}$:

if $T^s = T^t$ for some $t < s$,
then $\beta^s \leftarrow \beta^t$;
else
$\beta^s \xleftarrow{\$} \{0,1\}^n$;
$\mathcal{D}_1 \leftarrow \mathcal{D}_1 \cup \{T^s\}$; $\mathcal{R}_1 \leftarrow \mathcal{R}_1 \cup \{\beta^s\}$;
endif
$MM^s \leftarrow \beta^s \oplus P_1^s \oplus h\mathsf{BRW}_h(P_2^s||\ldots||P_m^s)$;
$CC^s \leftarrow \beta^s \oplus C_1^s \oplus h\mathsf{BRW}_h(C_2^s||\ldots||C_m^s)$;
$S^s \leftarrow MM^s \oplus CC^s$;
$\mathcal{D}_1 \leftarrow \mathcal{D}_1 \cup \{MM^s\}$; $\mathcal{R}_1 \leftarrow \mathcal{R}_1 \cup \{CC^s\}$;
for $i = 2$ to m,
$Y_i^s \leftarrow C_i^s \oplus P_i^s$;
$\mathcal{D}_1 \leftarrow \mathcal{D}_1 \cup \{S^s \oplus \mathsf{bin}_n(i-1)\}$;
$\mathcal{R}_1 \leftarrow \mathcal{R}_1 \cup \{Y_i^s\}$;
end for

Case $ty^s = \mathsf{Dec}$:

if $T^s = T^t$ for some $t < s$,
then $\beta^s \leftarrow \beta^t$;
else
$\beta^s \xleftarrow{\$} \{0,1\}^n$;
$\mathcal{D}_1 \leftarrow \mathcal{D}_1 \cup \{T^s\}$; $\mathcal{R}_1 \leftarrow \mathcal{R}_1 \cup \{\beta^s\}$;
endif
$MM^s \leftarrow \beta^s \oplus P_1^s \oplus h\mathsf{BRW}_h(P_2^s||\ldots||P_m^s)$;
$CC^s \leftarrow \beta^s \oplus C_1^s \oplus h\mathsf{BRW}_h(C_2^s||\ldots||C_m^s)$;
$S^s \leftarrow MM^s \oplus CC^s$;
$\mathcal{D}_1 \leftarrow \mathcal{D}_1 \cup \{MM^s\}$; $\mathcal{R}_1 \leftarrow \mathcal{R}_1 \cup \{CC^s\}$;
for $i = 2$ to m,
$Y_i^s \leftarrow C_i^s \oplus P_i^s$;
$\mathcal{D}_1 \leftarrow \mathcal{D}_1 \cup \{S^s \oplus \mathsf{bin}_n(i-1)\}$;
$\mathcal{R}_1 \leftarrow \mathcal{R}_1 \cup \{Y_i^s\}$;
end for

SECOND PHASE
bad $\leftarrow$ false;
if (some value occurs more than once in $\mathcal{D}_1$) **then** bad $\leftarrow$ true **endif**;
if (some value occurs more than once in $\mathcal{R}_1$) **then** bad $\leftarrow$ true **endif**.

Chapter 5

Implementing the Basic Building Blocks

As described in Chapter 3 the basic building blocks for TES are block ciphers. However, all modes other than EME2 which are described in Chapter 3 require the computation of a polynomial hash. The computation of the hash in turn involves computation of multiplications in some finite field. Other than the block cipher calls and multiplications some modes requires some additional finite field operations like squaring and xtimes.

In this chapter we describe the implementational issues of the basic building blocks of the constructions presented in this thesis. In the subsequent sections we describe the basic strategies adopted in implementation of the AES and some basic finite field operations (multiplication, squaring and xtimes). For all the modes implemented we consider a block-size of 128 bits, hence we use AES-128 and the finite field of interest to us is the field $\mathbb{F}_{2^{128}}$.

5.1 Binary Field Operations

We denote a field of 2^n elements by $\mathbb{F}_{2^n}$. We shall often view an $n-$bit binary string $a = (a_{n-1}, \ldots, a_1, a_0)$ as an element in $\mathbb{F}_{2^n}$. Any n bit string a can be represented as a polynomial in one variable of degree at most $n-1$ with coefficients in $\mathbb{F}_2$. Then, the polynomial representation of a would be $a(x) = \sum_{i=0}^{n-1} a_i x^i$. In any software implementation, these coefficients are typically packed in different word sizes, depending on the word sizes available in the processor.

Addition in such a representation would amount to adding two polynomials where the coefficients are added modulo 2. Thus given two n bit strings, the sum of them can be realized by the bit wise exclusive or (xor) operation. For multiplication we fix an irreducible polynomial of degree n representing the field, and given two field elements in their polynomial representation we compute their product as the product of the polynomials modulo the irreducible polynomial.

5.1.1 Multiplication

We describe here how we perform multiplication in the field $\mathbb{F}_{2^{128}}$, where the elements of $\mathbb{F}_{2^{128}}$ are viewed as polynomials of degree at most 127 with coefficients in $\mathbb{F}_2$. For the field operations we choose $q(x) = x^{128} + x^7 + x^2 + x + 1$ as the irreducible polynomial.

Given two polynomials $A(x) = \sum_{i=0}^{127} a_i x^i$ and $B(x) = \sum_{i=0}^{127} b_i x^i$, we break them as

$$A(x) = A_0(x) + x^{64} A_1(x)$$
$$B(x) = B_0(x) + x^{64} B_1(x),$$

where each $A_0(x)$, $A_1(x)$, $B_0(x)$ and $B_1(x)$ are polynomials of degree at most 63 defined as

$$\begin{aligned} A_0(x) &= \sum_{i=0}^{63} a_i x^i \\ A_1(x) &= \sum_{i=0}^{63} a_{64+i} x^i \\ B_0(x) &= \sum_{i=0}^{63} b_i x^i \\ B_1(x) &= \sum_{i=0}^{63} b_{64+i} x^i. \end{aligned}$$

With such a partitioning of the polynomials and using the famous Karatsuba trick [38] the multiplication of $A(x)$ and $B(x)$ can be performed using 3 multiplication of polynomials each of degree at most 63. The method procedure is shown in Algorithm 5.11. This algorithm treats the polynomials $A(x)$ and $B(x)$ as 128 bit strings. The operation $*$ used in lines 1, 2 and 3 represents polynomial multiplication of two binary polynomials of degree at most 63. The output of the algorithm is a polynomial of degree at most 254, which is represented as a string of 256 bits.

Algorithm 5.11: Karatsuba Multiplier

input : $128 - bit$ strings $(A_1||A_0), (B_1||B_0)$
where A_1, A_0, B_1, B_0 are $64 - bit$ words
output: $256 - bit$ string S

1 $(C_1||C0) \leftarrow A_1 * B_1$
2 $(D_1||D_0) \leftarrow A_0 * B_0$
3 $(E_1||E_0) \leftarrow (A_0 \oplus A_1) * (B_0 \oplus B_1)$
4 $S \leftarrow C_1||(C_0 \oplus C_1 \oplus D_1 \oplus E_1)||(D_1 \oplus C_0 \oplus D_0 \oplus E_0)||D_0$

The reason behind this specific partitioning of the polynomials as used in the Algorithm 5.11 is because of the existence of a special AES NI instruction. AES NI provides a new instruction called PCLMULQDQ which does a carry-less multiplication of two 64 bit operands, which can be seen as multiplication of two polynomials in $\mathbb{F}_2[X]$ of degree at most 63. In a software implementation, this instruction can be suitably employed to implement the Algorithm 5.11.

As stated, the multiplication algorithm produces a 256 bit string which needs to be reduced using the irreducible polynomial $q(x)$. We do the reduction using a technique introduced in [26]. In [26], Gueron and Kounavis demonstrate a method to do reduction modulo the specific irreducible polynomial $q(x)$ using just shifts and xors. Their technique can be viewed as an extension of the Barrett modular reduction algorithm [4] to modulo-2 arithmetic, or as an extension of the Feldmeier cyclic redundancy check generation algorithm [18] to dividends and divisors of arbitrary size. The reduction algorithm described in [26] is depicted in Algorithm 5.12. It takes as input a 256 bit string representing a polynomial of degree at most 254 and performs the desired reduction modulo $q(x)$, and thus outputs a 128 bit string. For the correctness and other details of the algorithm we refer the reader to [26].

5.1.2 Xtimes

For a $A \in \mathbb{F}_{2^{128}}$, some modes require the computation of $xA \bmod q(x)$. This operation is called *xtimes*. *xtimes* can be easily computed by a left shift followed by a conditional xor as described in Figure 5.1. In this same figure, where `0x87` is the string representation of the polynomial $x^7 + x^2 + x + 1$. As there is no instruction available for shifting a 128 bit register by one position, so the xtimes turns out to be a bit costly. Let $X = A||B$, where both $A = [a_{63}....a_0]$ and $B = [b_{63}....b_0]$ are 64 bits

Algorithm 5.12: Fast Reduction modulo $q(x)$

input : $256 - bit$ strings $(A_3||A_2||A_1||A_0)$
where A_3, A_2, A_1, A_0 are $64 - bit$ words
output: $128 - bit$ string S

1 $X_0 \leftarrow A_3 >> 63$
2 $X_1 \leftarrow A_3 >> 62$
3 $X_2 \leftarrow A_3 >> 57$
4 $X_3 \leftarrow A_2 \oplus X_0 \oplus X_1 \oplus X_2$
5 $(B_1||B_0) \leftarrow (A_3||X_3) << 1$
6 $(C_1||C_0) \leftarrow (A_3||X_3) << 2$
7 $(D_1||D_0) \leftarrow (A_3||X_3) << 7$
8 $(E_1||E_0) \leftarrow (A_3 \oplus B_1 \oplus C_1 \oplus D_1)||(X_3 \oplus B_0 \oplus C_0 \oplus D_0)$
9 $S \leftarrow (A_1 \oplus E_1)||(A_0 \oplus E_0)$

xtimes(A)
1. $b \leftarrow \mathsf{msb}(A)$
2. $A \leftarrow A << 1$
3. **if** $b = 1$
4. $\quad A \leftarrow A \oplus 0\text{x}87$
5. **return** A

Figure 5.1: The xtimes operation

long, both A and B can be shifted right by one position using a single instruction, but this would lead to loss of the bits b_{63} and a_{63}, and some amount of book-keeping and data movement is necessary to keep track of these bits lost and ultimately computing *xtimes*. We use the instructions `PSRAD` and `PSHUFD` to achieve this goal. The pseudo-code for the operation is described in Algorithm 5.13.

Algorithm 5.13: *xtimes*

input : $128-bit$ String $A = (A_3||A_2||A_1||A_0)$
where A_3, A_2, A_1, A_0 are 32-bit words, and
$A_i = a_{i,31}a_{i,30}\ldots a_{i,0}$
output: $128-bit$ string S

1 $R \leftarrow \underline{a_{3,31}}||\underline{a_{2,31}}||\underline{a_{1,31}}||\underline{a_{3,31}}$
// where $\underline{a}$ means a repeated 32 times.
// R can be obtained from A by the instruction PSRAD
2 $S \leftarrow \underline{a_{1,31}}||\underline{a_{1,31}}||\underline{a_{1,31}}||\underline{a_{3,31}}$
// S can be obtained from R by using PSHUFD instruction
3 $S \leftarrow S \wedge (0\text{x}00||0\text{x}01||0\text{x}00||0\text{x}87)$
4 $S \leftarrow S \oplus [(A_3||A_2) << 1||(A_1||A_0) << 1]$

This operation has significant computational overhead. Our implementation shows that this operation takes around 6 cycles to be computed.

5.1.3 Squaring

For computation of the BRW polynomials squares are required to be computed in $\mathbb{F}_q$. Computing squares in binary fields are easier than a full multiplication, as if $A \in \mathbb{F}_{2^{128}}$ is represented as $A = \sum_{i=0}^{127} a_i x^i$ then we have $A^2 = \sum_{i=0}^{127} a_i x^{2i} \bmod q(x)$. To perform this operation we use the technique used in [3](Algorithm 1). The technique used in [3] involves using a lookup table for squares of four bit polynomials and byte interleaving.

The algorithm we used is described in Algorithm 5.14. The symbols $\wedge, \gg$ are used to refer to the bitwise `AND` and the bitwise right shift of 8-bit arrays, respectively.

For performing the *lookup* procedure the processor instruction `PSHUFB` is used. `PSHUFB` essentially does the following:

- It receives two 128-bit parameters A, B which can be seen as 8-bit arrays of 16 elements, $A := (a[0], \ldots, a[15])$, $B := (b[0], \ldots, b[15])$

- Returns a 128-bit value also seen as an array of 16 elements $R := (r[0], \ldots, r[15])$ which is obtained with the following:
  - `If` $(b[0]$ & `0x80`$)$ $r[0] \leftarrow$ `0x00` `else` $r[0] \leftarrow b[a[0]$ & `0x0F`$]$.
  - `If` $(b[1]$ & `0x80`$)$ $r[1] \leftarrow$ `0x00` `else` $r[1] \leftarrow b[a[1]$ & `0x0F`$]$.
    $\vdots$
  - `If` $(b[15]$ & `0x80`$)$ $r[15] \leftarrow$ `0x00` `else` $r[15] \leftarrow b[a[15]$ & `0x0F`$]$.

The byte interleaving step, denoted in Algorithm 5.14 as $interleaving_{-L}$ and $interleaving_{-H}$ uses the support of the instructions `PUNPCKLBW` and `PUNPCKHBW`. Both instructions interleave the four 8-bit values from the low/high half of its first parameter with the four values from the low/high half of its second parameter.

Algorithm 5.14: Squaring in $\mathbb{F}_{2^{128}}$

input : $128-bit$ strings $A := (A_15|| \ldots ||A_1||A_0)$
where $A_15, \ldots, A_1, A_0$ are $8-bit$ words
output: $256-bit$ string $S := (S_H||S_L)$
where S_H, S_L are $128-bit$ words

1 $mask_L \leftarrow$ (`0x0F,0x0F,0x0F,0x0F,0x0F,0x0F,0x0F,0x0F,0x0F,0x0F,0x0F,0x0F,0x0F,0x0F,0x0F,0x0F`)
2 $mask_H \leftarrow$ (`0xF0,0xF0,0xF0,0xF0,0xF0,0xF0,0xF0,0xF0,0xF0,0xF0,0xF0,0xF0,0xF0,0xF0,0xF0,0xF0`)
3 $table \leftarrow$ (`0x55,0x54,0x51,0x50,0x45,0x44,0x41,0x40,0x15,0x14,0x11,0x10,0x05,0x04,0x01,0x00`)
4 $S_L \leftarrow A \wedge mask_L$
5 $S_H \leftarrow A \wedge mask_H$
6 $S_H \leftarrow S_H \gg 4$
7 $S_L \leftarrow lookup(table, S_L)$
8 $S_H \leftarrow lookup(table, S_H)$
9 $S_L \leftarrow interleaving_{-L}(S_H, S_L)$
10 $S_H \leftarrow interleaving_{-H}(S_H, S_H)$

For more details of the implementation of the basic operations described above, we refer to the reader to Appendix A where the corresponding source code is shown.

5.2 The Advanced Encryption Standard (AES)

The Rijndael cipher [15] was developed by Joan Daemen and Vincent Rijmen and it was selected by the National Institute of Standards and Technology (NIST) in November 2001 to become the Advanced Encryption Standard (AES), the replacement for

the commonly used Data Encryption Standard (DES) [54]. Until now, AES is the Federal Information Processing Standard for symmetric encryption, and is defined by FIPS publication number 197 [20].

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. The operations defined in AES are at byte level. Each byte represents an element in the finite field $\mathbb{F}_{2^8}$ and some other operations are defined in terms of 4-byte words. Unlike DES, the AES cipher does not have a Feistel structure.

The AES cipher key is expanded into an array of 4-byte word elements where each one of them serves as a round key during one round. That means each round key is $128-bit$ long, and is derived from the cipher key by another algorithm called the *Key Expansion*.

The key expansion algorithm is independent of the input data and receives only the cipher key. For this reason it can be executed prior to the encryption/decryption phase. The algorithm consists of the combination of two transformations: `SubWord`, `RotWord`. These transformations are defined as follows:

- **SubWord**: This transformation uses a substitution table called *S-box*. The SubWord function takes a 4-byte input word and applies the S-Box to each byte of the input to produce an 4-byte output. The way this S-box is applied is analogous to the SubBytes operation.

- **RotWord**: Takes also a 4-byte word as input, and performs a left rotation of the word. For example, given $[a_0, a_1, a_2, a_3]$, where each a_i is 1 byte long, RotWord returns $[a_1, a_2, a_3, a_0]$ as output.

Besides the operations described above, AES Key Expansion procedure uses a constant array called *Rcon*. Each i-th Rcon value, denoted in [20] as `Rcon[i]`, is an element in $\mathbb{F}_{2^8}$ and corresponds to the i-th power of x, where x is the hexadecimal value 0x02.

AES encryption/decryption algorithm has four different stages: 1 of permutation and 3 of substitution. All the AES algorithm's operations are performed using an intermediate value called the *state* which can be seen as a array of 4 by 4-bytes words. Each round receives two 128-bit inputs: the *state* and a *round key*. The

following are the transformations applied to the input data during one round of encryption/decryption:

- **AddRoundKey**: Returns the bitwise XOR ($\bigoplus$) of the state and the corresponding round key. Lets denote the state as the 4 by 4-byte array A, and the round key i as the 4 by 4-byte array R.

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \leftarrow \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \bigoplus \begin{bmatrix} r_{00} & r_{01} & r_{02} & r_{03} \\ r_{10} & r_{11} & r_{12} & r_{13} \\ r_{20} & r_{21} & r_{22} & r_{23} \\ r_{30} & r_{31} & r_{32} & r_{33} \end{bmatrix}$$

- **SubBytes and InvSubBytes**. This operation substitutes each byte of the state by another byte using the S-box table. For each byte a of the state, the SubBytes transformation is performed as follows :

  - Compute the corresponding multiplicative inverse of a over $\mathbb{F}_{2^8}$. The element $\{00\}$ maps to itself.
  - Apply the following affine transformation, where a_i is the j-th bit of a. In order to get a'.

$$\begin{bmatrix} a'_0 \\ a'_1 \\ a'_2 \\ a'_3 \\ a'_4 \\ a'_5 \\ a'_6 \\ a'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

  This values can be precomputed and stored in a 256 lookup table. This structure is a substitution table called S-box.

- **ShiftRows and InvShiftRows**. It rotates the rows of the AES state by different distances. For the ShiftRows transformation:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{11} & a_{12} & a_{13} & a_{10} \\ a_{22} & a_{23} & a_{20} & a_{21} \\ a_{33} & a_{30} & a_{31} & a_{32} \end{bmatrix} \leftarrow \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

And for the InvShiftRows:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{13} & a_{10} & a_{11} & a_{12} \\ a_{22} & a_{23} & a_{20} & a_{21} \\ a_{31} & a_{32} & a_{33} & a_{30} \end{bmatrix} \leftarrow \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

- **MixColumns and InvMixColumns**. It operates on the state column by column treated as polynomials over F_{2^8} and multiplied modulo x^4+1 with a the fixed polynomial $c(x) = \{03\}x^3+\{01\}x^2+\{01\}x+\{02\}$ for the MixColumns step or by its inverse $c^{-1}(x) = \{0b\}x^3+\{0d\}x^2+\{09\}x+\{0e\}$ for the InvMixColumns transformation. This operation can be seen as multiplication between the state and a fixed circulant matrix. For each column j.

  MixColumns:

$$\begin{bmatrix} a_{0j} \\ a_{0j} \\ a_{0j} \\ a_{0j} \end{bmatrix} \leftarrow \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} a_{0j} \\ a_{0j} \\ a_{0j} \\ a_{0j} \end{bmatrix}$$

  InvMixColumns:

$$\begin{bmatrix} a_{0j} \\ a_{0j} \\ a_{0j} \\ a_{0j} \end{bmatrix} \leftarrow \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \cdot \begin{bmatrix} a_{0j} \\ a_{0j} \\ a_{0j} \\ a_{0j} \end{bmatrix}$$

The final round of both encryption and decryption consists of only 3 transformations. The same explained before without the MixColumns/InvMixColumns transformation.

Only the AddRoundKey transformation uses the key, other transformations are reversible without the knowledge of the key and together provide *confusion, diffusion* and *nonlinearity* [20].

Even though the order of the transformations is important, the SubBytes and ShiftRows transformations can be permuted. The same can be applied for InvSubBytes and InvShiftRows.

AES encryption algorithm is described in Figure 5.2. This algorithm and the rest detailed in this section use the notation given in [20].

Algorithm AES Encryption Algorithm
Input: `Data: 16` *bytes* `of plaintext`
`Round_Key_Encrypt: array of` $11-15$ $16-$ *bytes* `blocks.`

```
1. state = AddRoundKey (Data, Round_Key_Encrypt [0])
2. For round = 1-9 or 1-11 or 1-13:
3.    state = ShiftRows (state)
4.    state = SubBytes (state)
4.    state = MixColumns (state)
5.    state = AddRoundKey (state, Round_Key_Encrypt [round])
6. end loop
7. state = ShiftRows (state)
8. state = SubBytes (state)
9. state = AddRoundKey (state, Round_Key_Encrypt [10/12/14])
```

Output: `state` (16-bytes)

Figure 5.2: AES encryption algorithm using FIPS-197 notation

5.2.1 Intel AES-NI architecture

Since its standardization, designing fast implementations of AES has become an active research area. Examples of this research are bitsliced implementations for Intel core i7 architectures [8, 39] and some other implementations in a large variety of CPU architectures [32, 7]. Even a fast AES implementation targeting microcontrollers is described in [55].

After Intel introduced the streaming SIMD extensions (SSE) in 1999, which is a SMID instruction set extension to the x86 architecture, the use of this paradigm became very popular due to enhance the speed of many software implementations by using special units which operate on larger data types, improving overall throughput.

One of the recent additions to these SSE extension, is the AES instruction set (AES-NI) [24] available from the 2010 Intel Core processor family based on the 32nm micro-architecture named *Westmere*, and also supported by AMD since their CPU generation called *Bulldozer*.

In the AES-NI extension there are six instructions related to AES encryption and decryption. The following four instructions are provided in order to encrypt/decrypt the data: `AESENC, AESENCLAST, AESDEC, AESDECLAST`. While the key expansion is processed by two instructions: `AESKEYGENASSIST, AESIMC`.

Intel's AES-NI instructions operate on one or two 128−*bit* inputs, and the typical instruction format is `instruction xmm1 xmm2/m128`. Where `xmm` denotes a 128 bit register and `m128` a 128 bit memory location. The related instructions for AES encryption/decryption are described next.

- `AESENC xmm1, xmm2/m128`: Perform one round of an AES encryption call operating on a 128−*bit* data from xmm1 with a 128−*bit* round key from xmm2/m128. Thus, this single instruction performs all the relevant transformations of one AES round, namely, `ShiftRows, SubBytes, MixColumns` y `AddRoundKey`.

- `AESENCLAST xmm1, xmm2/m128`: Perform the last round of an AES encryption call operating on a 128−*bit* data from xmm1 with a 128−*bit* round key from xmm2/m128. Thus, this instructions performs the transformations `ShiftRows, SubBytes` y `AddRoundKey`.

- `AESDEC xmm1, xmm2/m128`. Perform one round of an AES decryption call, using the equivalent inverse cipher, operating on a 128−*bit* data from `xmm1` with a 128−*bit* round key from `xmm2/m128`.

- `AESDECLAST xmm1, xmm2/m128`. Perform the last round of an AES decryption call, using the equivalent inverse cipher, operating on a 128−*bit* data from `xmm1` with a 128−*bit* round key from `xmm2/m128`.

- `AESKEYGENASSIST xmm1, xmm2/m128, imm8`: Round key generation. Assist in AES round key generation using an 8−*bit* round constant (RCON) specified in the immediate byte, operating on 128 bits of data stored in `xmm2/m128` and save the result in `xmm1`.

- `AESIMC xmm1, xmm2/m128`: Perform the inverse MixColumn transformation on a 128−*bit* round key from `xmm2/m128` and store the result in `xmm1`.

Among the properties of AES algorithm, there exist a way to use the same sequences of transformations as in the cipher, each transformation replaced by its inverse, in order to perform AES decryption. This is possible with a small change in the key schedule. This equivalent way for performing AES decryption is called the *Equivalent Inverse Cipher*, and is used by Intel architecture. The other way to perform this operation is called *Inverse Cipher* but regarding the purpose of the section the description of this operation is omitted. Nevertheless it is important to mention

Algorithm AES Encryption Algorithm using AES-NI instructions set

Input: `Data: 16` *bytes* `of plaintext`
`Round_Key_Encrypt: array of` $11 - 15$ $16 -$ *bytes* `blocks.`

1. `state = XOR128 (Data, Round_Key_Encrypt[0])`
2. `For round = 1-9 or 1-11 or 1-13:`
3. `state = AESENC(state, Round_Key_Encrypt[round])`
4. `end loop`
5. `state = AESENCLAST(state, Round_Key_Encrypt[10/12/14])`

Output: `state` (16-bytes)

Figure 5.3: AES encryption algorithm using AES-NI instructions set

that the difference between them is that the order in which the the transformations are applied is not the same, which also changes the way that the decryption round keys are defined.

The round keys used for the *Equivalent Inverse Cipher* are created with the following steps:

1. Let `Round_Key_Encrypt[j]` be the round key used in round number `j` during the encryption flow. For `j=0,1,2,...,Nr`, where `Nr` takes the values of 10,12 or 14 depending on the selected key version of AES (AES-128, AES-192, AES-256 respectively).

2. Let `Round_Key_Decrypt[j]` be the decryption round key used in round number `j` during the decryption flow.

3. Derive the decryption round keys from the encryption round keys, in the following manner:

   (a) `Round_Key_Decrypt[0]` $\leftarrow$ `Round_Key_Encrypt[Nr]`

   (b) For every `round` $\leftarrow$ `1,2, to Nr - 1` do

   (c) `Round_Key_Decrypt[round]` $\leftarrow$ `InvMixColumns(Round_Key_Encrypt[Nr-round])`

   (d) Finally, `Round_Key_Decrypt[Nr]` $\leftarrow$ `Round_Key_Encrypt[0]`

AES encryption algorithm using AES-NI instructions is described in Algorithm 5.3.

Summary

We described the implementational issues of the AES and some basic finite field operations (multiplication, squaring and xtimes) used in this thesis. We use AES-128 and the finite field of interest to us is the field $\mathbb{F}_{2^{128}}$. Regarding the multiplication in the field we use the famous Karatsuba trick. In the case of the squaring operation we use a technique used in [3] which uses a lookup table for squares of four bit polynomials and byte interleaving. All operations were designed to take advantage of SMID instructions including the AES-NI instruction set available in the new Intel and AMD processors.

Chapter 6

Experimental Results

In this Chapter we discuss in details the experimental results and compare the performance of the various schemes studied in this thesis. In Section 6.1 we describe the basic implementation strategies with a thrust to the strategies employed for the multi-threaded implementations. In Section 6.3 we state the testing methodology adopted, and finally in Section 6.4 we give the detailed results in various scenarios.

6.1 Basic Implementation Strategies

We implemented the following modes: EME2, HEH[Poly], HMCH[Poly], XCB, HCTR, HEH[BRW], HMCH[BRW], HMCH2, HCTR*, XTS and BitLocker. All implementations were done for a message length of 4096 bytes, as this is the size of a sector for the currently available hard disks. The tweak was considered to be of 128-bit.

For all the implementations we use AES-128 as the underlying block cipher and the finite field operations are over the field $\mathbb{F}_{2^{128}}$ defined with the irreducible polynomial $x^{128}+x^7+x^2+x+1$. All implementations are in C using the AES-NI instruction set and the PCLMULQDQ instruction for the multiplication in the field using Karatsuba's method as explained in Chapter 5.

The new AES round instructions are pipelined and can be dispatched theoretically every 1-2 CPU clock cycles [25] if there is no data dependency between subsequent calls and data can be provided sufficiently fast. This turns out to be very convenient when we use the AES block cipher in parallel modes of operation like the CTR mode or the ECB mode which are employed for some of the TES layers. For these parallel modes, instead of waiting for the completion of the encryption/decryption of one

data block and then continuing with the subsequent blocks, a better throughput is obtained when we compute AES rounds on multiple blocks in parallel. In general, it is recommended to process 4-8 blocks in parallel in order to achieve high throughput [25]. For this work we process 4 blocks in parallel for AES whenever possible.

We have tried to vectorize the code and used the SSE instruction wherever possible. Thus, our code tries to utilize the instruction level parallelism and data level parallelism to the fullest extent. We also did some implementations using thread level parallelism by utilizing the multiple cores. Considering the application of disk sector encryption, where the message length is fixed, and multiple messages are needed to be encrypted/decrypted in a short period of time, there can be two strategies for parallelization utilizing multiple cores:

1. Given a scheme **E** finding possibilities of parallelism for encrypting a single message using **E** and running different parts of the scheme in different cores and ultimately combining the results to obtain the final cipher/plain text.

2. To run various instances of **E** in different cores in parallel and thus encryption/decrypting one message/cipher in one core.

Most TES are designed to support the parallelization strategy as in the strategy 1. Careful analysis of possible parallelization in some existing TES were done in [46, 10] in the context of hardware implementations. Those analysis are valid to a large extent for software implementations also and it is possible to divide work between different processors for processing a single message. But, this would involve creating and merging the threads and codes in critical sections. Creation of threads consumes lot of cycles and the wait required for critical sections to complete may not be always very predictable.

As the message sizes for the application of disk encryption are small, parallel overhead exceeds the real computation time. A parallel implementation will be efficient in terms of speed if the communication overhead between threads is less than the cost of the procedure/routine itself. Hence we decided to follow strategy 2, where each message gets encrypted in a different core in parallel. This strategy does not involve any parallelization within the algorithm, and thus careful division of tasks among cores is not required and the results shown later suggests that this give rise to the desired speedups.

6.2 System Information

All constructions have been implemented on two different machines with the following specifications:

Machine 1 (M1)

- **CPU**: Intel® Core™ i5-661 @ **3.33GHz**, 4M Cache. (**2 cores**, 4 logical threads)
- **Memory**: 4GB DDR3.
- **OS**: GNU/Linux Fedora release 16 (Verne)
- **Compilers**: `GCC 4.6.2` and `ICC 12.0.1`

Machine 2 (M2)

- **CPU**: Intel® Core™ i5-2400 CPU @ **3.10GHz**, 6M Cache. (**4 cores**, 4 logical threads)
- **Memory**: 4GB DDR3.
- **OS**: GNU/Linux Fedora release 16 (Verne)
- **Compilers**: `GNU Compiler 4.6.2` and `ICC 12.0.4`

Both machines M1 and M2 feature 64-bit instruction set, and SSE4.1/4.2 instruction set extension. In the case of machine M2, the Advanced Vector Extensions (AVX) are also included. For the ease of development, we only use the intrinsics for the C programming language provided by Intel with two different compilers: the GNU Compiler Collection (GCC) and the Intel Compiler (ICC), as both compilers support these. Finally, all tests were run with Intel Turbo Boost Technology and Enhanced Intel Speedstep Technology disabled. In the case of machine M1 Intel Hyper-Threading Technology was also disabled. The running time was measured when no X-server and no network daemon were running.

6.3 Testing methodology

For performance evaluation a simple benchmark procedure was developed. An example of the basic skeleton of the benchmark procedure is shown with the next C macro

```
1 #define BENCHMARK(x)  for (i=0; i< WARMUP; i++)          \
2             {x;}                          \
3             start_cycles=get_cycles();          \
4             for (i = 0; i < BENCH; i++)          \
5             {x;}                          \
6             end_cycles=get_cycles();          \
7             total=(double)(end_cycles-start_cycles)/BENCH;
```

Listing 6.1: Basic benchmarking procedure.

In the above listing `x` represents the set of instructions used for the implementation of a specific scheme. To obtain the cycle counts for a specific set of instructions we aim to reduce the cache effects. The effects of transition from memory to data cache and memory to instruction cache are generally called as cache effects. The first loop in the listing above handles these cache effects.

To eliminate cache effects, both the instructions and data must be contained in the L1 cache, which is the cache closest to the processor. The technique of storing memory into a cache before it is actually used is known as "cache warming". *Warm up* the cache simply requires "pass through" the entire data set which is going to be used, so that it will be moved into the cache. Warming the instruction cache requires making a first pass through all instructions before the timing begins. We accomplished this by putting the entire procedure which we want to test in a loop. After this warming, the cycle counts are computed as an averaged over 1,000,000 runs of the set of instructions.

The benchmarks were done with the help of the time stamp counter which is read using the `RDTSC` instruction. This instruction returns the number of ticks since reset in registers `EDX:EAX`. We used the following piece of ASM code for this purpose:

```
1 static __inline UINT64 get_cycles(void) {
2   UINT64 tmp;
3   __asm__ volatile(
4       "rdtsc\n\t\
5       mov %%eax,(%0)\n\t\
6       mov %%edx,4(%0)"::"rm"(&tmp):"eax","edx");
7   return tmp;
8 }
```

Listing 6.2: Code used to measure time.

6.4 Results

The results presented in this Section are an average over 1,000,000 calls to the corresponding functions as explained before. First we present the results of the basic building blocks. In Table 6.1 we report the time required for the operations of multiplication, squaring, xtimes and reduction in the field $\mathbb{F}_{2^{128}}$. Note that the time of multiplication reported includes the time for reduction. In Table 6.2 we show the performance of three versions of the counter mode. The three versions differ in the way the counter is updated. In the first version (with xtimes) the initial counter value is updated by performing an *xtimes* operation repeatedly as in the mode HMCH. In the second version (with xor) the counter value is updated by xor-ing a constant (as in the mode HCTR, HCTR* and HMCH2). The third version (addition modulo 2^{32}) the counter value is updated by incrementing the last 32 bits of the initial counter (as in XCB). The counter mode for all versions were implemented by grouping four AES calls together. As is evident from the results in Table 6.2, 32 bit increments performs the best whereas xtimes performs the worst.

	Total clock cycles
Multiplication	37.38
Squaring	11.54
xtimes	6.01
Reduction	9.01

Table 6.1: Total clock cycles for computing the basic binary field operations over $\mathbb{F}_{2^{128}}$ in M2 using the ICC compiler.

	Total clock cycles	Cycles per byte
with xtimes	5061.02	1.23
with xor	4851.67	1.18
with addition mod 2^{32}	3998.61	0.97

Table 6.2: Counter mode variants for a 4KB buffer using ICC compiler in machine M2.

Next, we present the performance comparison of the various modes in cycles per bytes. The number of CPU cycles needed to encrypt a message is divided by the length of the message to derive the cost per byte to encrypt messages of that length, for short **cpb** (*cycles per byte*). All results include all memory and loop overheads. The performance results for both compilers using encryption and decryption are shown in Tables 6.3, 6.4. Table 6.3 shows results for TES modes and consists of two parts

(a) and (b). In (a) the performance figures include the computation of the AES key expansion. But in (b) the key expansion procedure is not included. We show the performance of XTS and BitLocker separately in Table 6.4. In these Tables, we show the timings both for encryption and decryption in terms of cycles per byte for both machines M1 and M2 using two different compilers (ICC and GCC). The results described in the Tables are also shown as histograms in Figures 6.1, 6.2, 6.3 for easy visual comparisons.

In the sub-tables in Table 6.3 the TES are grouped into three groups. The first group contain the encrypt-mask-encrypt scheme EME2 which only uses block cipher calls and the second and third groups contains the schemes which also use finite field multiplications in addition to the block cipher calls. The second group contains the schemes which are constructed using normal polynomial hash functions and the schemes in the last group uses BRW polynomials. The best candidate in each group is marked in bold.

The following observations summarize the obtained results:

- **The best TES**
  EME2 has the best performance among the TESs. EME2 uses little more than 2 blockcipher calls for each block of message and all the block cipher calls are totally parallelizable. Thus EME can use the instruction level pipelining of the AES instructions to the full extent. Other than the block cipher calls EME2 uses some xtimes operations, but all other TES uses significant number of multiplications in $\mathbb{F}_{2^{128}}$ (roughly two per block for modes using normal polynomials and one per block for the modes using BRW polynomials). For one multiplication in $\mathbb{F}_{2^{128}}$, three calls to the PCLMULQDQ are required with some other operations. Each PCLMULQDQ has a latency of about 12 cycles [23], which makes it more expensive than the AES calls.

- **BRW polynomials vs usual polynomials:**
  HEH[BRW], HMCH[BRW], HMCH2, HCTR*, are the schemes that uses BRW polynomials and they are significantly faster than the schemes HCTR and HMCH[Poly], HEH[Poly] which use usual polynomials. The reason behind this is that BRW-based schemes utilize about half the number of multiplications compared to other schemes which require usual polynomials.

- **Schemes using usual polynomials:**
  In most scenarios XCB has the best performance results among the schemes

M1:

	Encryption		**Decryption**	
	gcc	icc	gcc	icc
EME2	3.22	3.27	3.23	3.29
HEH[p]	8.56	7.86	8.75	7.88
HMCH[p]	8.46	7.84	8.47	7.86
XCB	**8.38**	7.74	8.55	**7.74**
HCTR	8.52	**7.72**	**8.36**	7.75
HEH[B]	4.90	4.58	4.84	4.58
HMCH[B]	4.78	4.54	4.81	4.53
HMCH2	**4.58**	4.38	**4.59**	4.39
HCTR*	4.66	**4.33**	4.66	**4.33**

M2:

	Encryption		**Decryption**	
	gcc	icc	gcc	icc
EME2	2.83	2.65	2.90	2.77
HEH[p]	7.49	7.29	7.46	7.30
HMCH[p]	7.12	7.13	7.13	7.12
XCB	**6.97**	**6.99**	**6.97**	**7.00**
HCTR	7.01	7.12	7.01	7.12
HEH[B]	4.51	4.00	4.48	4.02
HMCH[B]	4.16	3.80	4.14	3.81
HMCH2	**3.92**	**3.75**	**3.92**	**3.75**
HCTR*	4.00	4.00	4.00	4.00

(a)

M1:

	Encryption		**Decryption**	
	gcc	icc	gcc	icc
EME2	3.16	3.27	3.16	3.26
HEH[p]	8.52	7.86	8.55	7.86
HMCH[p]	8.46	7.79	8.46	7.77
XCB	**8.27**	**7.63**	**8.27**	**7.63**
HCTR	8.34	7.68	8.34	7.68
HEH[B]	4.83	4.55	4.91	4.55
HMCH[B]	4.78	4.48	4.78	4.47
HMCH2	**4.55**	4.33	**4.56**	4.34
HCTR*	4.64	**4.32**	4.64	**4.29**

M2:

	Encryption		**Decryption**	
	gcc	icc	gcc	icc
EME2	2.77	2.61	2.78	2.61
HEH[p]	7.35	7.29	7.40	7.29
HMCH[p]	7.07	6.94	7.07	6.94
XCB	**6.83**	**6.85**	**6.83**	**6.88**
HCTR	6.94	6.99	6.93	6.99
HEH[B]	4.33	3.99	4.38	4.00
HMCH[B]	4.03	3.66	4.03	3.66
HMCH2	**3.92**	**3.64**	**3.91**	**3.64**
HCTR*	3.97	3.99	3.98	3.99

(b)

Table 6.3: Encryption and decryption implementation results **in clock cycles per byte** (ccpb), for a **4KB** buffer and a **128-bit tweak** using both machines M1 and M2. (a) includes Key Expansion procedure and (b) does not include it.

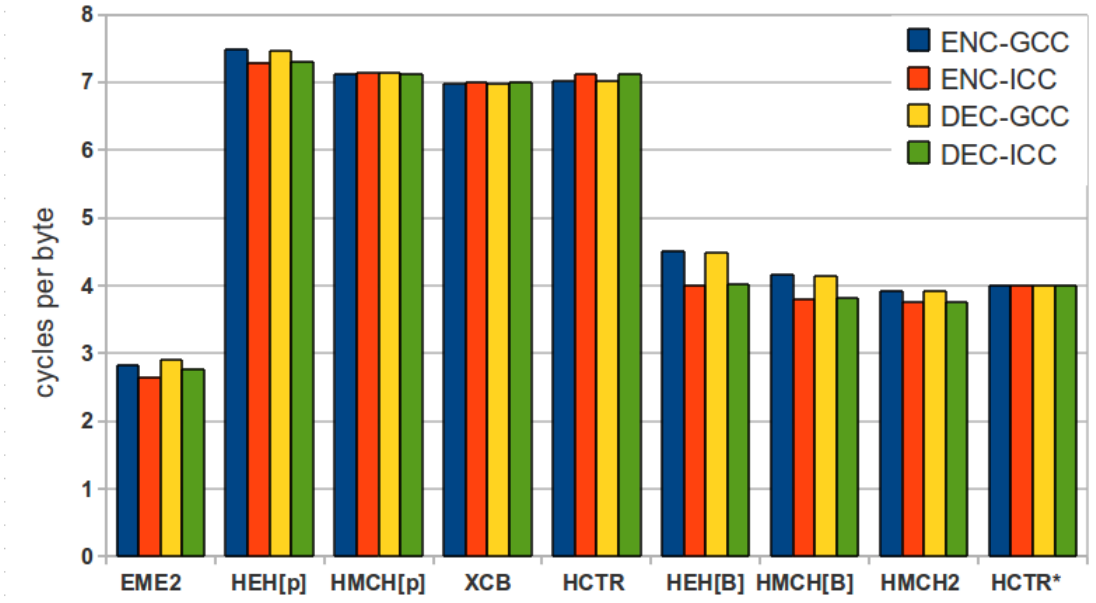
M1:

	Encryption		**Decryption**	
	gcc	icc	gcc	icc
XTS	1.58	1.65	1.57	1.65
BitLocker	18.35	12.58	15.50	10.18

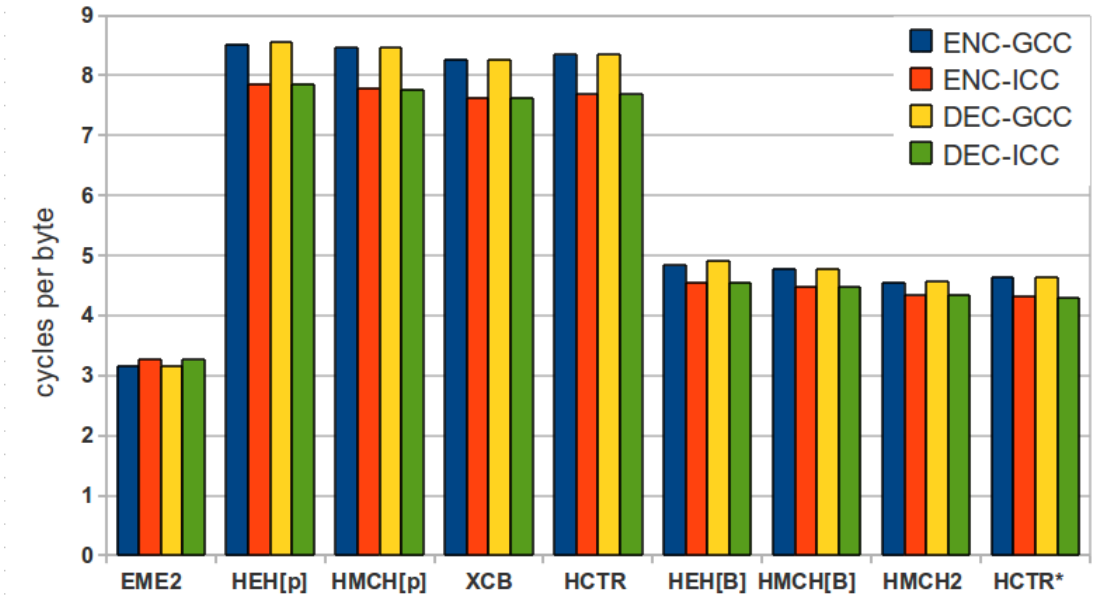
M2:

	Encryption		**Decryption**	
	gcc	icc	gcc	icc
XTS	1.36	1.47	1.45	1.48
BitLocker	17.32	13.90	13.27	10.28

Table 6.4: Encryption and decryption performance results of other disk encryption schemes: XTS, BitLocker **in clock cycles per byte** (ccpb), for a **4KB** buffer using both machines M1 and M2.

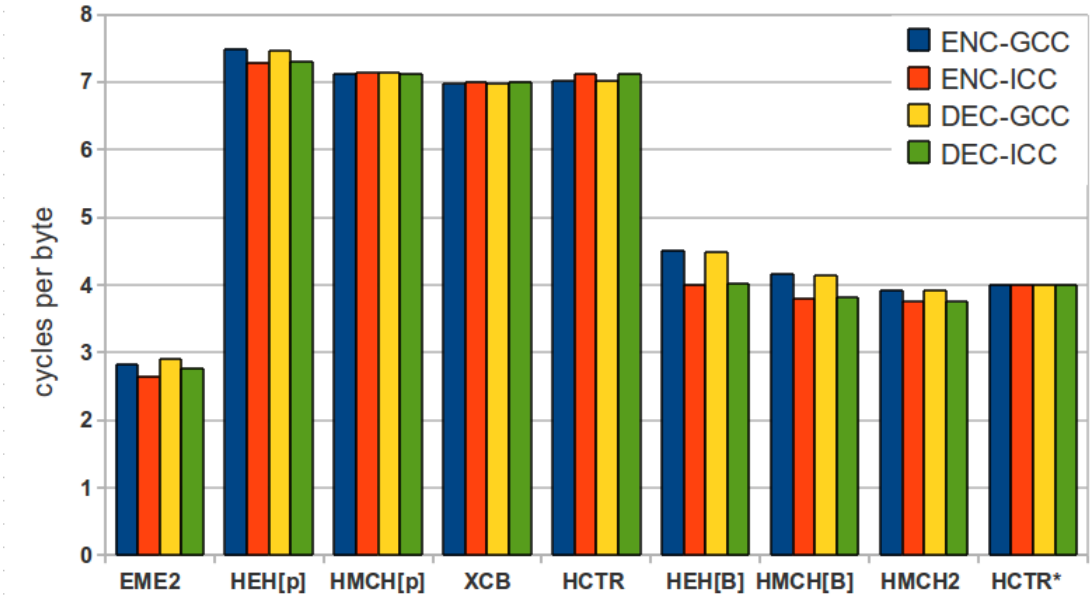


(a)

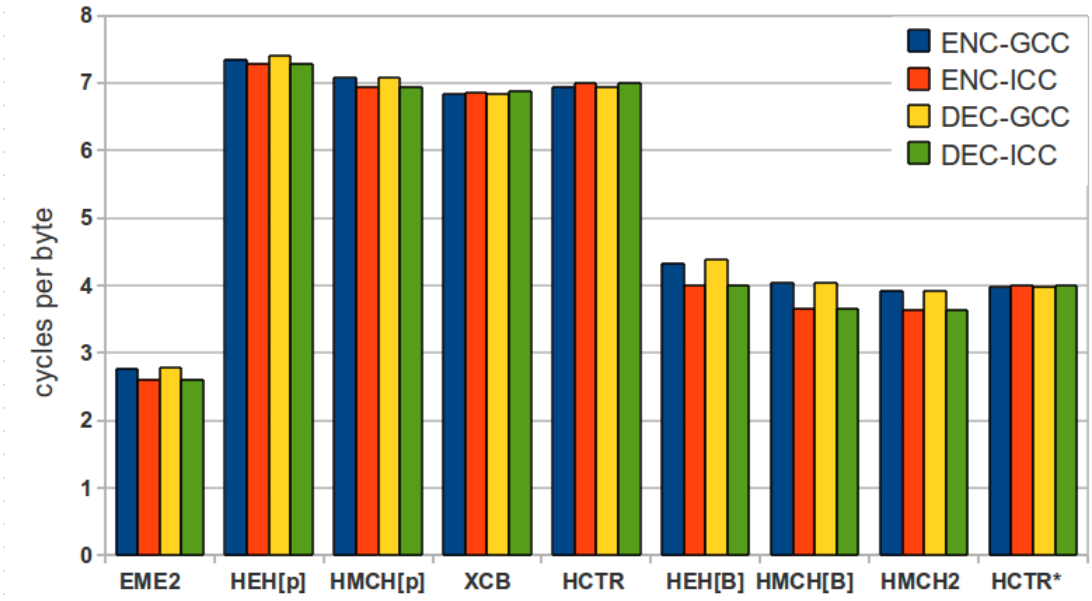


(b)

Figure 6.1: Encryption (ENC) and decryption (DEC) implementation results **in clock cycles per byte** (ccpb), for a **4KB** buffer and a **128-bit tweak** using both machines M1, figure (a) includes Key Expansion procedure and (b) does not include it.



(a)



(b)

Figure 6.2: Encryption (ENC) and decryption (DEC) implementation results **in clock cycles per byte** (ccpb), for a **4KB** buffer and a **128-bit tweak** using both machines M2, figure (a) includes Key Expansion procedure and (b) does not include it.

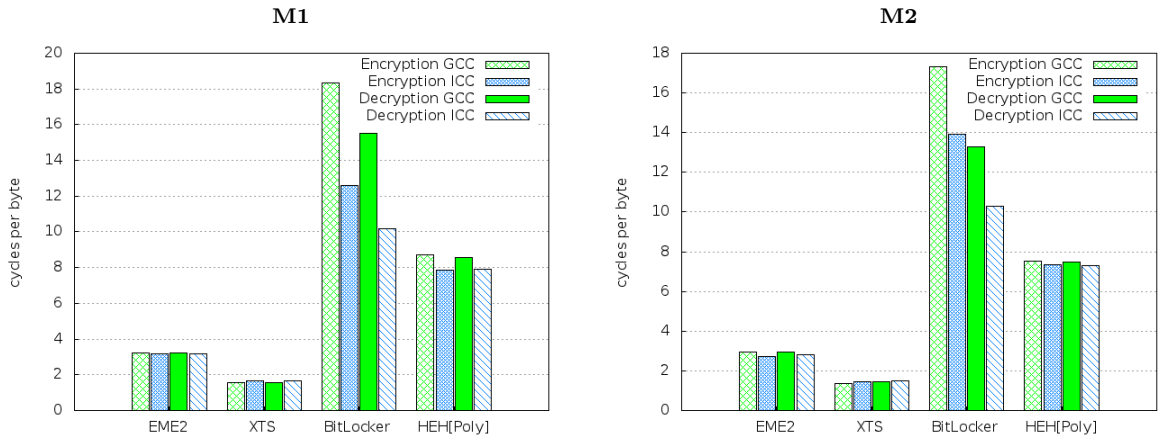


Figure 6.3: Encryption and decryption comparison of other disk encryption schemes with some TES modes **in clock cycles per byte** (ccpb), for a **4KB** buffer using both machines M1 and M2. Results include key schedule.

which use usual polynomials. Only in machine M1 HCTR encryption wins in this group when compiled using ICC and HCTR decryption wins when compiled using GCC, when the computation of the key schedules is also included for counting the cycles. XCB uses three different blockcipher keys and hence there is a requirement for running three different key schedules, whereas all other schemes in this category use only one AES key. Moreover XCB uses six AES calls more than HCTR. Thus, it is non-intuitive that XCB performs better than HCTR (or even other schemes in the category). A possible explanation of this is the way the counter mode is implemented in XCB. In XCB the counter uses an increment modulo 2^{32} of the last 32 bits of the counter value, which can be implemented by a single instruction on an 128 bit register. On the other hand, the CTR mode used for HCTR encryption/decryption, requires to perform an XOR of each input block with the binary representation of a counter value. To compute this value, it is needed to perform some extra operations which are more expensive than the addition modulo 2^{32} required by the XCB mode. This possibly explains the better performance of XCB.

The other modes in this category namely HEH[Poly] and HMCH[Poly] uses significant number of xtimes operations, which are not present in HCTR and XCB. The xtimes operations when implemented in 128 bit registers is particularly inefficient, according the estimates shown in [41] one xtime operation

costs approximately 6.8 cycles in a typical Westmere machine. This explains the worse performance of HEH[Poly] and HMCH[Poly] compared to HCTR and XCB. HEH[Poly] has the worst performance result as it requires twice the number of xtimes operations compared to HMCH[Poly].

- **Schemes using BRW polynomials:**
  HMCH2 is the fastest among BRW-based schemes followed by HCTR*. Compared to its predecessors, both schemes obtained a remarkable gain. The main reason of this results is due to the fact that neither HMCH2 nor HCTR* use any xtimes operation.

- **The schemes that are not TES:**
  XTS and BitLocker are the schemes which are not TES. Figure 6.3 shows its comparison between EME2 and HEH[Poly], the TES modes which according to our study are the most and least efficient. BitLocker is the least efficient disk encryption scheme as the diffusers are designed to exploit 32-bit architectures and data parallelization is not suitable in their design. XTS is the fastest disk encryption scheme, as like EME2 it uses only AES calls and thus gains a lot from the use of AES-NI. XTS performs much better than EME2 because it uses only half the number of AES calls compared to EME2. As stated before, though XTS is the fastest it has some glaring issues regarding its security, and we are skeptic whether it provides adequate security for the application.

6.4.1 Comparisons

As we stated at the beginning of this document there are no performance data in software of TES modes available in the literature. The only implementations reported, are the hardware implementations reported in [47] and [10]. In [47] optimized implementations of EME, HCTR, HEH, XCB and some other scheme were reported. No scheme using BRW polynomials were included in the study in [47]. The implementations were directed towards Virtex 4 family of FPGAs, and throughputs were reported for encrypting disk sectors of 512 bytes. In [10] an efficient methodology for computing BRW polynomials were proposed and the performance comparisons for HEH[Poly], HMCH[Poly], HEH[BRW] and HMCH[BRW] were provided. The implementations in [10] were directed towards Virtex 5 family of FPGAs and the throughputs were reported for message lengths of 512 bytes. Thus, strictly speaking, the results in [10, 47]

are not comparable, but as these are the only available results we compare our results with them.

In Table 6.5 we show the results reported in [10, 47] for encrypting a sector of 512 bytes. The performance is shown in terms of total clock cycles and the throughput in Gigabits per second. In Table 6.6 we show the throughput of the TES modes results obtained using ICC compiler, for encrypting a big sector (4096 bytes) using machine M2. And in 6.7 we show our results for encryption 512 bytes.

	Encryption Total clock cycles	**Throughput** GBits/Sec	Source
EME	107	2.597	[47]
HEH[Poly]	83	10.768	[10]
HMCH[Poly]	94	9.825	[10]
HEH	75	3.956	[47]
XCB	116	1.907	[47]
HCTR	89	3.665	[47]
HEH[BRW]	55	15.184	[10]
HMCH[BRW]	66	13.193	[10]

Table 6.5: Hardware implementation results presented in [10, 47] which show the number of clock cycles and its corresponding throughput for various TES constructions to encrypt a whole disk sector of 512 bytes.

	Encryption Total clock cycles	**Throughput** GBits/Sec
EME2	10,854	9.358
HEH[p]	29,860	3.402
HMCH[p]	29,205	3.478
XCB	28,631	3.548
HCTR	29,164	3.483
HEH[B]	16,384	6.200
HMCH[B]	15,565	6.526
HMCH2	15,360	6.613
HCTR*	16,384	6.200

Table 6.6: Encryption implementation throughput **GBit/Sec**, for a **4KB** buffer and a **128-bit tweak** using machine M2 and ICC compiler, including Key Expansion procedure

From Table 6.5 we see that the implementations in [10] perform much better than that reported in [47], as in [10] better strategies for parallelization were adopted and the results are in a Virtex 5 device which in general are able to operate at higher frequencies. Our results as reported in Table 6.6 are comparable to that of the results

	Encryption Total clock cycles	**Throughput** GBits/Sec
EME2	1664	7.630
HEH[p]	3796	3.345
HMCH[p]	3705	3.427
XCB	4142	3.066
HCTR	3856	3.293
HEH[B]	2258	5.623
HMCH[B]	2171	5.849
HMCH2	2173	5.843
HCTR*	2322	5.468

Table 6.7: Encryption implementation throughput **GBit/Sec**, for a **512-bytes** buffer and a **128-bit tweak** using machine M2 and ICC compiler, including Key Expansion procedure

reported in [47]. In fact our implementations of EME2 and XCB gives much better throughput than those reported in [47].

The fastest known AES implementation without AES-NI support is reported in [39]. In their implementation for encrypting a 4096 byte message using counter mode, it takes at least 6.92 cycles per byte in a Intel Core i7 920. For all our TES implementations which uses BRW and EME2 performs much better than this. It is to be noted that any TES uses much more operations than a single counter mode. So use of AES-NI gives significant speed ups.

The goal of our implementations is to obtain high speed in order to achieve an encryption/decryption speed which matches the speed of the current data rates of commercial disk controllers. With emerging technologies like serial ATA and Native Command Queuing (NCQ), modern day disk controllers can provide data rates around 3 Giga-bits per second [61]. Thus, it is clearly seen that the throughput results obtained by our software implementations are competitive enough with hardware designs and can also achieve the required speed for a low-level disk encryption.

Multi-threaded implementation results

As discussed earlier, for the multi-threaded implementations we implement procedures so that one sector gets encrypted/decrypted in each core, we do not attempt to parallelize the operations within a single sector. Such an implementation can be practically useful, as in general a disk controller can get read/write requests of several sectors almost simultaneously and thus individual sectors may be processed in differ-

ent CPU cores. Under this scenario we do not consider computing the key schedules for each sector. In a realistic scenario, the key schedules would be computed only once during the system startup and they would be used for subsequent encryption/decryption thus making the cost involved for key scheduling negligible compared to the total encryption/decryption overhead.

For our multi-threaded implementations we use the POSIX thread libraries. Since both machines have at least two cores, we run all tests for 2 threads, and in the case of machine M2 we also did experiments with 4 threads. We assure thread affinity, that is, we force a thread to run on a specific core. This is done using the following methodology: The system represents affinity with a bitmask called a processor affinity mask. The size of the affinity mask is the size of the maximum number of processors in the system, with bits set to identify a subset of processors. Initially, the system determines the subset of processors in the mask. A CPU affinity mask is represented by the CPU_SET_T structure, a "CPU set", pointed to by mask. Then, SCHED_SETAFFINITY() sets the CPU affinity mask of the process whose ID is pid to the value specified by mask. Thus, the following codes was used to reach this goal.

```
1 int set_thread_affinity(int thread_num){
2   cpu_set_t currentCPU;
3   CPU_ZERO(&currentCPU);
4   CPU_SET(thread_num, &currentCPU);
5   sched_setaffinity (0, sizeof(currentCPU), &currentCPU);
6 }
```

Listing 6.3: Code used to ensure thread affinity.

Table 6.8 show the parallel encryption and decryption implementation results with 2 threads in clock cycles per byte (ccpb), for 200 and 1000 sectors of 4096 bytes each with a 128-bit tweak. Both compilers were used (GCC and ICC), and for a better comparison we show its corresponding histograms in Figures 6.4 and 6.5.

As machine M2 has 4 cores, we show in Table 6.9 the results obtained for encrypting and decrypting 1000 sectors of 4096 bytes with 4 threads using both compilers (GCC and ICC). In Figure 6.6 we give a graphic comparison between these and the results obtained for 2 threads and 1000 sectors in the same machine.

The results of the multi-threaded implementations show that the increase in encryption/decryption speed is almost directly proportional to the number of cores. It is expected that processors with Hyper-threading technology which provide multiple logical cores per one physical core would provide similar proportional results with only a very small thread overhead. The comparative performance among the various

M1:

	Encryption		**Decryption**	
	gcc	icc	gcc	icc
EME2	1.58	1.67	1.58	1.66
HEH[p]	4.27	3.94	4.38	3.94
HMCH[p]	4.24	3.90	4.24	3.89
XCB	**4.14**	**3.82**	**4.14**	**3.83**
HCTR	4.17	3.84	4.17	3.84
HEH[B]	2.43	2.29	2.47	2.29
HMCH[B]	2.39	2.26	2.39	2.25
HMCH2	2.44	**2.17**	2.53	**2.18**
HCTR*	**2.33**	2.18	**2.33**	2.19

M2:

	Encryption		**Decryption**	
	gcc	icc	gcc	icc
EME2	1.42	1.36	1.41	1.36
HEH[p]	3.66	3.65	3.68	3.65
HMCH[p]	3.51	3.47	3.51	3.47
XCB	**3.42**	**3.44**	**3.42**	**3.44**
HCTR	3.47	3.50	3.47	3.51
HEH[B]	2.16	2.00	2.19	2.00
HMCH[B]	2.02	1.83	2.02	1.84
HMCH2	**1.96**	**1.83**	**1.95**	**1.84**
HCTR*	1.99	2.00	1.99	2.00

(a)

M1:

	Encryption		**Decryption**	
	gcc	icc	gcc	icc
EME2	1.67	1.76	1.67	1.75
HEH[p]	4.34	4.01	4.30	4.00
HMCH[p]	4.31	3.97	4.31	3.96
XCB	**4.21**	**3.89**	**4.21**	**3.89**
HCTR	4.25	3.91	4.25	3.91
HEH[B]	2.59	2.36	2.67	2.52
HMCH[B]	2.53	2.43	2.59	2.46
HMCH2	**2.30**	**2.25**	**2.31**	2.40
HCTR*	2.53	2.29	2.53	**2.18**

M2:

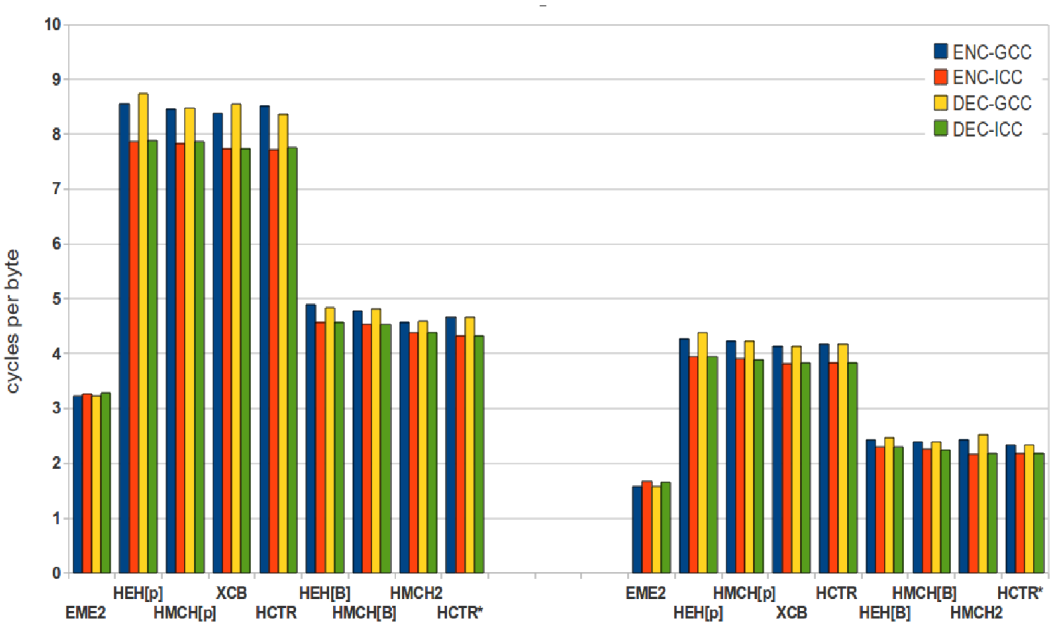
	Encryption		**Decryption**	
	gcc	icc	gcc	icc
EME2	1.45	1.39	1.45	1.39
HEH[p]	3.69	3.68	3.71	3.69
HMCH[p]	3.54	3.51	3.54	3.51
XCB	**3.46**	**3.48**	**3.46**	**3.48**
HCTR	3.50	3.54	3.50	3.54
HEH[B]	2.25	2.05	2.32	2.13
HMCH[B]	2.09	1.88	2.12	1.96
HMCH2	**2.02**	**1.87**	**2.06**	**1.95**
HCTR*	2.08	2.08	2.08	2.08

(b)

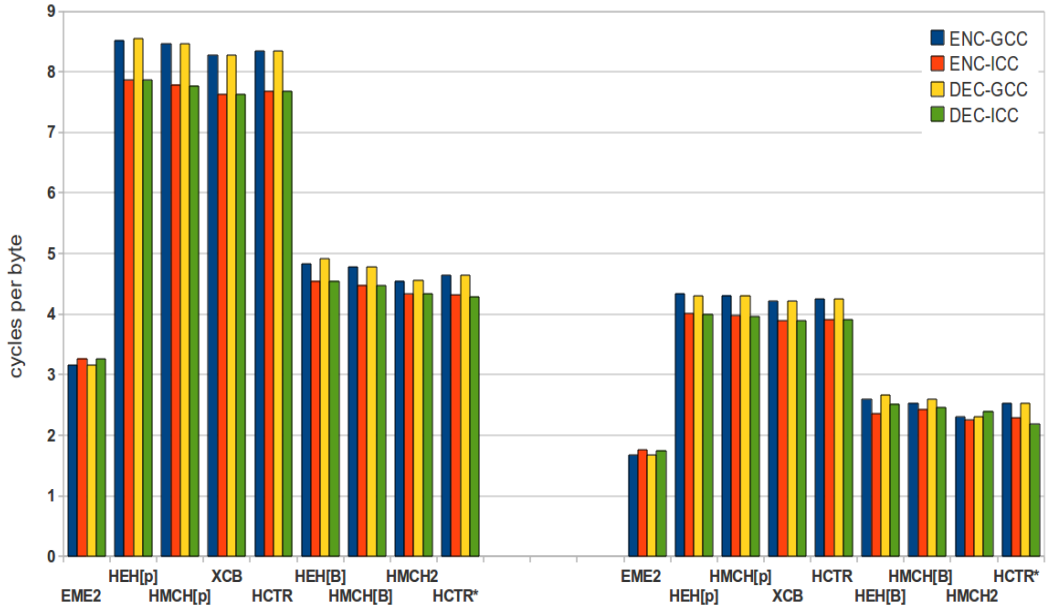
Table 6.8: Parallel encryption and decryption implementation results with **2 threads in clock cycles per byte** (ccpb), for (a) **200** and (b) **1000** sectors of **4KB** and a **128-bit tweak** using both machines M1 and M2.

	Encryption		**Decryption**	
	gcc	icc	gcc	icc
EME2	0.74	0.71	0.74	0.71
HEH[p]	1.85	1.85	1.87	1.85
HMCH[p]	1.78	1.76	1.78	1.77
XCB	**1.74**	**1.75**	**1.74**	**1.75**
HCTR	1.77	1.78	1.77	1.78
HEH[B]	1.14	1.04	1.18	1.08
HMCH[B]	1.06	0.95	1.08	0.99
HMCH2	**1.02**	**0.95**	**1.03**	**0.98**
HCTR*	1.06	1.06	1.06	1.06

Table 6.9: Parallel encryption and decryption implementation results with **4 threads in clock cycles per byte** (ccpb), for **1000** sectors of **4KB** and a **128-bit tweak** using machine M2.

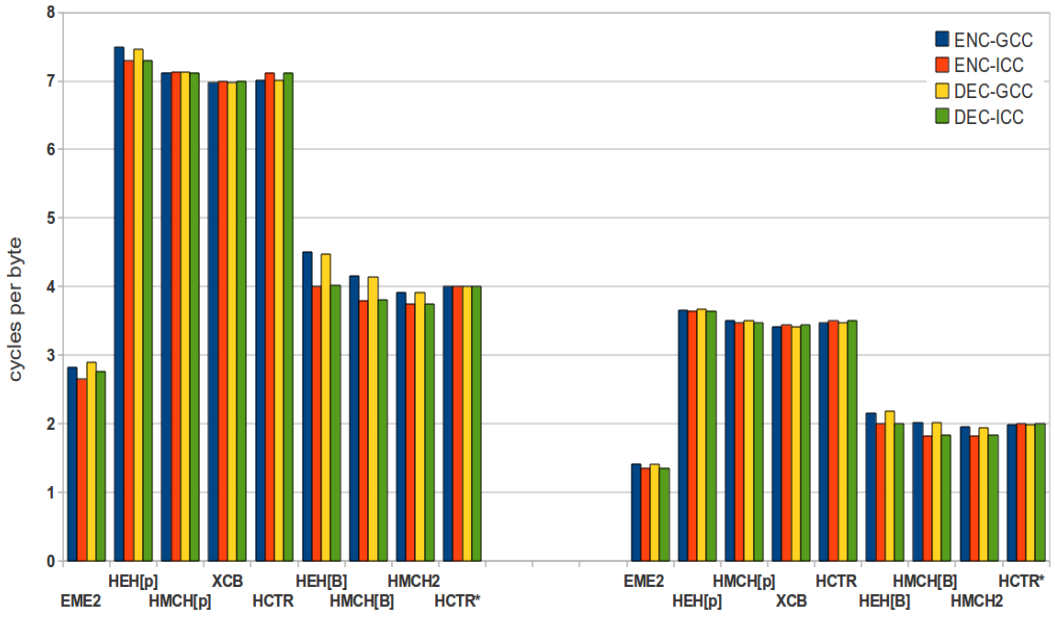


(a)

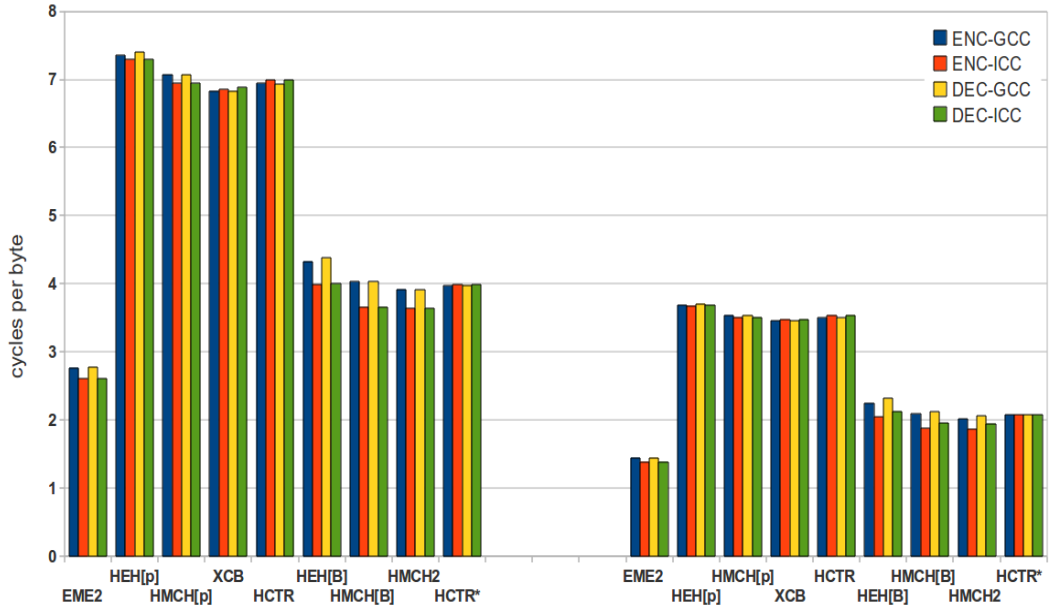


(b)

Figure 6.4: Comparison of parallel encryption and decryption implementation results with **2 threads in clock cycles per byte** (ccpb), for (a) **200** and (b) **1000** sectors of **4KB** and a **128-bit tweak** using machine M1.



(a)



(d)

Figure 6.5: Comparison of parallel encryption and decryption implementation results with **2 threads in clock cycles per byte** (ccpb), for (a) **200** and (b) **1000** sectors of **4KB** and a **128-bit tweak** using machine M2.

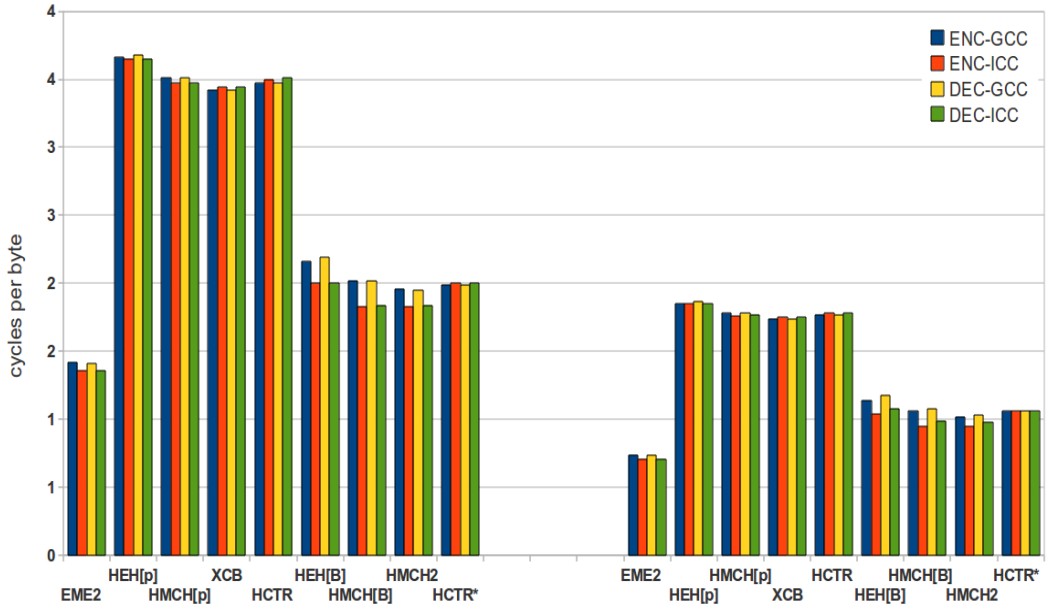


Figure 6.6: Comparison of parallel encryption and decryption implementation results with **2 and 4 threads in clock cycles per byte** (ccpb), for (a) **1000** sectors of **4KB** and a **128-bit tweak** using machine M2.

TES is also retained in the multi-threaded implementations.

Chapter 7

Conclusion

With the increased need to store digital information, security of data stored in bulk storage devices like hard disks of laptop and desktop computers, flash memories, and a variety of small mobile devices, has become an important issue of today. Disk encryption prevents unauthorized access to data storage. Tweakable enciphering schemes (TES), are considered to be a main paradigm for disk encryption.

We presented a new approach for implementing TES modes of operation using different parallelization techniques: data and thread parallelization under the SIMD (Single Instruction, Multiple Data) paradigm. The new generation of modern Intel processors have been equipped with the new instructions for AES encryption/decryption including an instruction for performing a carry free multiplication of two 64 bit operands. With the aid of these new instructions the two most expensive operations of the existing TES (the block cipher calls and the finite field multiplications) can be now implemented much more efficiently than in older processors. In this thesis we experimentally showed that use of these new instructions leads to efficient software implementations of tweakable enciphering schemes.

From the different experiments presented in Chapter 6 using two different machines (M1 and M2) supporting SSE4.1/SSE4.2 instructions, we conclude that EME2 is the fastest among all the TES modes. EME2 is the only TES which uses only block cipher calls, thus it gains the most from using the AES-NI instructions. The TES modes other than EME2 uses finite field multiplications, as all of them requires to compute certain polynomial hashes over the messages or other intermediate values. Among the modes that uses polynomial hashes there are two distinct divisions. One class of schemes uses the normal polynomials, which can hash a m block message using m multiplications and the other class uses a special class of polynomials called

BRW polynomials which can hash m message blocks using about $m/2$ multiplications. Thus, as evident, the class of modes using the BRW polynomials performs much better compared to the schemes which uses normal polynomials.

There exist no standard for TES till date. IEEE security in storage working group is working towards formulation of a standard for such modes. This body has selected two schemes EME2 and XCB as the candidates, but the final standard is not yet out. Our study suggest that EME2 is the fastest mode, and XCB is the fastest among all modes which uses the normal polynomial hashing. But, as stated earlier, there are other schemes available (like the ones that uses BRW polynomials) which are much more efficient than XCB, hence the choice of XCB for the standard is not clear. It would be worth mentioning here, that a previous performance study of TES in FPGA devices [47] suggested XCB to the least efficient mode. This was probably due to the more flexibility that a hardware designer has to exploit parallelism helped the other modes to gain more than XCB in a hardware implementation.

In Chapter 4 of this thesis we also proposed two new TES named HCTR* and HMCH2. Both these modes use BRW polynomials and our experiments suggests that they perform better than all existing modes which uses BRW polynomials. The new modes were designed as modifications of two existing modes (HCTR and HMCH[BRW]). The modifications were done keeping an eye to the efficiency gains that can be achieved by using SIMD instructions. In this context, the design of HMCH2 is particularly interesting. HMCH[BRW] which is the predecessor of HMCH2 uses a series of xtimes operations. The implementation of xtimes operations in 128 bit registers turned out to be particularly inefficient. Thus in HMCH2 we successfully remove the xtimes operations at the cost of just an increment and a 128 bit xor. Results in Chapter 6 clearly demonstrate that this subtle change do have a very positive impact on the efficiency.

To guarantee security of TES, one uses some reductionist arguments where one reduces the security of the whole TES to that of the block cipher. In other words, one argues that a break in the TES would imply a break of the underlying block cipher. Thus, such an argument guarantees that a TES would be secure as long as the underlying blockcipher is secure. Such a reductionist argument is generally called a security proof, and schemes for which such an argument is applicable are called provably secure. But, it should be noted that a security proof does not suggest absolute security, but is a statement of security in a specified model relative to the security of an underlying primitive. All existing TES studied in this thesis have

security proof attached to it. In fact, a new TES without a security proof is not acceptable. Thus in Chapter 4 we provide security proofs for HCTR* and HMCH2. Our proofs do not use any new idea, but closely follow proofs of other existing TES.

There are schemes other than TES which are used for disk encryption. In this thesis we included two schemes called XTS and BitLocker. XTS is a NIST standard for encrypting block oriented storage devices like hard disk, and BitLocker (to our knowledge) is the only scheme which has been widely deployed in general purpose devices through the Microsoft Windows Vista operating system. XTS is not a TES and does not enjoy security properties similar to TES, this is well known and also the NIST standard document acknowledges that XTS may be vulnerable to certain kinds of cryptanalytic attacks. In Chapter 3 we describe a convincing attack on XTS, which clearly show that XTS does not attain similar security as that of any TES. The attack, though is simple, but to our knowledge is the first concrete attack demonstrated on XTS. Experimental results in Chapter 6 shows XTS to be the most efficient scheme for disk encryption, which is not surprising as XTS only uses a single ECB type encryption layer, whereas all other TES are much more complex. The performance advantage of XTS should be interpreted in accordance to its security weakness.

BitLocker is also not a TES, and there is no security proof for BitLocker. The authors argue that BitLocker achieves security through a secure CBC mode along with some scrambling done by two diffusers. The exact properties of the diffusers used in BitLocker have not been adequately studied, thus one has every right to be skeptic about this scheme. But, there exist no concrete attack against bit locker. Our experiments show that BitLocker has a very poor performance when implemented with the 128 bit register support. This is due to the fact that BitLocker was designed for a 32 bit architecture, and efficient implementation of the diffusers within 128 bit registers is not possible. Additionally, BitLocker uses the CBC mode, which is inherently serial and thus instruction pipelining of the AES-NI instructions cannot be used in here.

In summary, in this thesis we present a comprehensive study of disk encryption schemes. Though the main direction of this study is in obtaining efficient implementation of such schemes in modern processors, but we also achieve significant other results like two new secure TES schemes which are most efficient in their category. Our performance studies shows that TES when implemented with AES-NI support can perform very good in terms of speed and can reach the data rates of modern disk

controllers, thus software implementations of TES may have practical applicability in the near future.

7.1 Future Work

As a future work, we would like to explore the following:

Implementation aspects

1. Although we took care on the performance and optimization of our C implementation, these numbers can certainly be improved if they were all implemented directly in assembly. As shown in [27, 25] where better results were obtained for the assembly versions. For a real world application this surely should be done.

2. Significant improvements may be achieved using AVX instructions in assembly directly. It is known that mixing 256-bit AVX instructions with legacy (non VEX-encoded) Intel SSE instructions may result in penalties that could impact performance. 256-bit AVX instructions operate on the 256-bit YMM registers which are 256-bit extensions of the existing 128-bit XMM registers. 128-bit AVX instructions operate on the lower 128 bits of the YMM registers and zero the upper 128 bits. However, legacy Intel SSE instructions operate on the XMM registers and have no knowledge of the upper 128 bits of the YMM registers. Because of this, the hardware saves the contents of the upper 128 bits of the YMM registers when transitioning from 256-bit AVX to legacy SSE, and then restores these values when transitioning back from SSE to AVX (256-bit or 128-bit). The save and restore operations both cause a penalty that amounts to several tens of clock cycles for each operation. There are several different situations where AVX-SSE transitions might occur, such as when 256-bit AVX intrinsic instructions or inline assembly are mixed with any of the following:

   - 128-bit intrinsic instructions
   - SSE inline assembly
   - C/C++ floating point code that is compiled to Intel SSE
   - Calls to functions or libraries that include any of the above

There are several methods to either remove AVX-SSE transitions or to remove the penalty from transitions. The easiest method to avoid the AVX-SSE transition penalty is to compile the relevant source files with the Intel Compiler using either the `-xavx` or `-mavx` for Intel compiler or `-msse2avx` or `-mavx` for GNU compiler. When these flags are used the compiler will automatically generate VEX-encoded instructions rather than legacy SSE instructions where appropriate, which removes the transition between AVX and SSE within those files. Although we use these flags in our experiments we do not know exactly how the compilers handles this, what or how many AVX instructions are used. Thus, the performance of the schemes in machine M2 (where AVX instructions are enabled) can be significantly improved with a redesign of the code to exploit the AVX instructions more.

3. Other than the AVX instructions, the latencies of the AES-NI instructions in machine M2 are quite different from that of machine M1. For example, we learned from a recent talk by Shay Gueron (Indocrypt 2011, Tutorial) that for the SandyBridge processors the optimal grouping of the AES encrypt/decrypt instructions should be done in groups of 8 instead of groups of 4 which was the optimal grouping in previous processors. Thus, a proper study of the latencies of the instructions in particular processors may lead to further optimized codes.

4. The main goal of our implementations was to achieve speed. We did not pay much attention to make our codes secure against known software side channel attacks like cache attacks and timing attacks. Designing code, keeping in mind these issues is something that we would like to focus in the near future.

Other aspects

1. Our experiments suggested that XCB gains a lot by using 32 bit increments in the counter value. There is no TES using BRW polynomials which uses this philosophy. We believe that such a scheme would result in much better performance.

2. The scheme BitLocker has not been adequately studied by the community. Finding a convincing security argument or an attack on BitLocker would be an interesting project.

3. Design of new TES or other cryptographic schemes which can take specific architectural advantage can be an interesting area of study. In the near future we would like to undertake this study and explore ways in which message authentication schemes, authenticated encryption schemes and other similar block cipher based schemes can be designed so that they would have high performance advantage when implemented in processors with AES-NI support.

Bibliography

[1] IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices. *IEEE Std 1619-2007*, pages c1–32, 2008.

[2] Ross J. Anderson and Eli Biham. Two Practical and Provably Secure Block Ciphers: BEARS and LION. In Dieter Gollmann, editor, *FSE*, volume 1039 of *Lecture Notes in Computer Science*, pages 113–120. Springer, 1996.

[3] Diego F. Aranha, Julio López, and Darrel Hankerson. Efficient Software Implementation of Binary Field Arithmetic Using Vector Instruction Sets. In Michel Abdalla and Paulo S. L. M. Barreto, editors, *LATINCRYPT*, volume 6212 of *Lecture Notes in Computer Science*, pages 144–161. Springer, 2010.

[4] Paul Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In Andrew M. Odlyzko, editor, *CRYPTO*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986.

[5] Mihir Bellare and Phillip Rogaway. Code-Based Game-Playing Proofs and the Security of Triple Encryption. *IACR Cryptology ePrint Archive*, 2004:331, 2004.

[6] Daniel J. Bernstein. Polynomial Evaluation and Message Authentication, 2007. `http://cr.yp.to/papers.html#pema`.

[7] Daniel J. Bernstein and Peter Schwabe. New AES software speed records. *IACR Cryptology ePrint Archive*, 2008:381, 2008.

[8] Eli Biham. A Fast New DES Implementation in Software. In Eli Biham, editor, *FSE*, volume 1267 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 1997.

[9] Larry Carter and Mark N. Wegman. Universal Classes of Hash Functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.

[10] Debrup Chakraborty, Cuauhtemoc Mancillas-Lopez, Francisco Rodriguez-Henriquez, and Palash Sarkar. Efficient Hardware Implementations of BRW Polynomials and Tweakable Enciphering Schemes. Cryptology ePrint Archive, Report 2011/161, 2011. `http://eprint.iacr.org/`.

[11] 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.

[12] Debrup Chakraborty and Palash Sarkar. A New Mode of Encryption Providing a Tweakable Strong Pseudo-random Permutation. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *Lecture Notes in Computer Science*, pages 293–309. Springer, 2006.

[13] 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.

[14] Paul Crowley. Mercy: A Fast Large Block Cipher for Disk Sector Encryption. In Bruce Schneier, editor, *FSE*, volume 1978 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2000.

[15] Joan Daemen and Vincent Rijmen. The Block Cipher Rijndael. In Jean-Jacques Quisquater and Bruce Schneier, editors, *CARDIS*, volume 1820 of *Lecture Notes in Computer Science*, pages 277–284. Springer, 1998.

[16] Morris Dworkin. Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality, 2004.

[17] Morris Dworkin. Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices. NIST Special Publication 800-38E, 2010. `http://csrc.nist.gov/publications/nistpubs/800-38E/nist-sp-800-38E.pdf`.

[18] David C. Feldmeier. Fast software implementation of error detection codes. *IEEE/ACM Trans. Netw.*, 3(6):640–651, 1995.

[19] N. Ferguson. AES-CBC+ Elephant diffuser: A disk encryption algorithm for Windows Vista. *Microsoft Corp*, 2006.

[20] FIPS. *Advanced Encryption Standard (AES)*. National Institute for Standards and Technology, pub-NIST:adr, November 2001.

[21] Scott R. Fluhrer. Cryptanalysis of the Mercy Block Cipher. In Mitsuru Matsui, editor, *FSE*, volume 2355 of *Lecture Notes in Computer Science*, pages 28–36. Springer, 2001.

[22] M. Flynn. Very High-Speed Computing Systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[23] Agner Fog. *Instruction tables : Lists of instruction latencies, throughputs and microoperation breakdowns for Intel, AMD and VIA CPUs*. Copenhagen University College of Engineering, 2011.

[24] S. Gueron. Advanced encryption standard (AES) instructions set. 25, 2008.

[25] Shay Gueron. Intel's New AES Instructions for Enhanced Performance and Security. In Orr Dunkelman, editor, *FSE*, volume 5665 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2009.

[26] Shay Gueron and Michael E. Kounavis. Efficient implementation of the Galois Counter Mode using a carry-less multiplier and a fast reduction algorithm. *Inf. Process. Lett.*, 110(14-15):549–553, 2010.

[27] Shay Gueron and Michael E. Kounavis. Intel Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode. `http://software.intel.com/en-us/articles/carry-less-multiplication-and-its-usage-for-computing-the-gcm-mode/`, January 2010.

[28] Shai Halevi. Eme*: Extending EME to Handle Arbitrary-Length Messages with Associated Data. In Anne Canteaut and Kapalee Viswanathan, editors, *INDOCRYPT*, volume 3348 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2004.

[29] Shai Halevi. Invertible Universal Hashing and the TET Encryption Mode. *IACR Cryptology ePrint Archive*, 2007:14, 2007.

[30] Shai Halevi and Phillip Rogaway. A Tweakable Enciphering Mode. *IACR Cryptology ePrint Archive*, 2003:148, 2003.

[31] 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.

[32] Mike Hamburg. Accelerating AES with Vector Permute Instructions. In Christophe Clavier and Kris Gaj, editors, *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2009.

[33] J.L. Hennessy, D.A. Patterson, and D. Goldberg. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2003.

[34] IEEE Security in Storage Working Group, (SISWG). PRP Modes Comparison. `http://siswg.org/`, November 2008. IEEE p1619.2.

[35] IEEE 1619 SISWG Security in Storage Working Group. Draft standard architecture for wide-block encryption for shared storage media. `https://siswg.net/index.php?option=com_content&task=view&id=36&Itemid=75`.

[36] Intel. *Intel C++ Compiler 12.0 User and Reference Guides*. INTEL Corporation.

[37] INTEL Corporation. *Intel 64 and IA-32 Architectures Software Developer Manuals*.

[38] A. Karatsuba and Yu. Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics—Doklady*, 7(7):595–596, January 1963.

[39] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant aes-gcm. In Christophe Clavier and Kris Gaj, editors, *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009.

[40] C. KENT. Draft Proposal for Tweakable Narrow-block Encryption. Draft. *IEEE Computer Society, August*, 2004.

[41] 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.

[42] Moses Liskov and Kazuhiko Minematsu. Comments on XTS-AES. Comments On The Proposal To Approve XTS-AES, 2008.

[43] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable Block Ciphers. In Moti Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2002.

[44] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable Block Ciphers. *J. Cryptology*, 24(3):588–613, 2011.

[45] Stefan Lucks. BEAST: A Fast Block Cipher for Arbitrary Blocksizes. In Patrick Horster, editor, *Communications and Multimedia Security*, volume 70 of *IFIP Conference Proceedings*, pages 144–153. Chapman & Hall, 1996.

[46] Cuauhtemoc Mancillas-López, Debrup Chakraborty, and Francisco Rodríguez-Henríquez. Efficient implementations of some tweakable enciphering schemes in reconfigurable hardware. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *INDOCRYPT*, volume 4859 of *Lecture Notes in Computer Science*, pages 414–424. Springer, 2007.

[47] Cuauhtemoc Mancillas-López, Debrup Chakraborty, and Francisco Rodríguez-Henríquez. Reconfigurable Hardware Implementations of Tweakable Enciphering Schemes. *IEEE Trans. Computers*, 59(11):1547–1561, 2010.

[48] D.A. McGrew and S.R. Fluhrer. The extended codebook (XCB) mode of operation. *Rep*, 278:2004, 2004.

[49] D.A. McGrew and J. Viega. Arbitrary block length mode. *Standards contribution*, 2004.

[50] 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.

[51] David A. McGrew and John Viega. The Security and Performance of the Galois/Counter Mode (GCM) of Operation. In Anne Canteaut and Kapalee Viswanathan, editors, *INDOCRYPT*, volume 3348 of *Lecture Notes in Computer Science*, pages 343–355. Springer, 2004.

[52] Kazuhiko Minematsu and Toshiyasu Matsushima. Tweakable Enciphering Schemes from Hash-Sum-Expansion. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *INDOCRYPT*, volume 4859 of *Lecture Notes in Computer Science*, pages 252–267. Springer, 2007.

[53] Moni Naor and Omer Reingold. A Pseudo-Random Encryption Mode, January 2002.

[54] National Bureau of Standards. *FIPS Publication 46-1: Data Encryption Standard*, January 1988.

[55] Dag Arne Osvik, Joppe W. Bos, Deian Stefan, and David Canright. Fast software aes encryption. In Seokhie Hong and Tetsu Iwata, editors, *FSE*, volume 6147 of *Lecture Notes in Computer Science*, pages 75–93. Springer, 2010.

[56] Michael O. Rabin and Shmuel Winograd. Fast evaluation of polynomials by rational preparation. *Communications on Pure and Applied Mathematics*, 25:433–458, 1972.

[57] Phillip Rogaway. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In Pil Joong Lee, editor, *ASIACRYPT*, volume 3329 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2004.

[58] Russel, R. M. The CRAY-1 Computer System. *Comm. ACM*, 21(1):63–72, 1978.

[59] 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.

[60] Palash Sarkar. Efficient tweakable enciphering schemes from (block-wise) universal hash functions. *IEEE Transactions on Information Theory*, 55(10):4749–4760, 2009.

[61] Seagate Technology. *Internal 3.5-Inch (SATA) Data Sheet*, 2010.

[62] Jonathan Taverne, Armando Faz-Hernández, Diego F. Aranha, Francisco Rodríguez-Henríquez, Darrel Hankerson, and Julio López. Speeding scalar multiplication over binary elliptic curves using the new carry-less multiplication instruction. *J. Cryptographic Engineering*, 1(3):187–199, 2011.

[63] Seagate Technology. Comments on XTS-AES. Comments On The Proposal To Approve XTS-AES, 2008.

[64] 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.

Appendix A

Source Codes of Some Basic Blocks

```
#define reduction()        \
 2 tmp5 = _mm_shuffle_epi32(tmp4,0x93);  \
 3 tmp5 = _mm_srli_epi32(tmp5,0x19); \
 4 tmp2 = _mm_srli_epi32(tmp5,0x05); \
 5 tmp3 = _mm_srli_epi32(tmp2,0x01); \
 6 tmp5 = _mm_xor_si128(tmp5,tmp2);  \
 7 tmp5 = _mm_xor_si128(tmp5,tmp3);  \
 8 tmp3 = _mm_and_si128(tmp5,tmp0);  \
 9 tmp5 = _mm_xor_si128(tmp5,tmp3);  \
10 tmp4 = _mm_xor_si128(tmp4,tmp3);  \
11 tmp0 = _mm_slli_epi32(tmp4,0x01); \
12 tmp2 = _mm_slli_epi32(tmp0,0x01); \
13 tmp3 = _mm_slli_epi32(tmp2,0x05); \
14 tmp4 = _mm_xor_si128(tmp4,tmp0);  \
15 tmp4 = _mm_xor_si128(tmp4,tmp2);  \
16 tmp4 = _mm_xor_si128(tmp4,tmp3);  \
17 tmp4 = _mm_xor_si128(tmp4,tmp1);  \
18 *res = _mm_xor_si128(tmp4,tmp5);  \
19
20static inline void gfmul_karatsuba(__m128i a, __m128i b, __m128i *res){
21 __m128i tmp0,tmp1,tmp2,tmp3,tmp4,tmp5;
22
23 tmp0 = _mm_set_epi32(0x00,0x00,0x00,0xFF);
24 tmp1 = _mm_shuffle_epi32(a,0x44);
25 tmp2 = _mm_shuffle_epi32(b,0x44);
26 tmp1 = _mm_xor_si128(tmp1,a);
27 tmp2 = _mm_xor_si128(tmp2,b);
28 tmp1 = _mm_clmulepi64_si128(tmp1,tmp2,0x11);
29 tmp2 = _mm_clmulepi64_si128(a,b,0x00);
30 tmp3 = _mm_clmulepi64_si128(a,b,0x11);
31
32
33 tmp1 = _mm_xor_si128(tmp1,tmp2);
34 tmp1 = _mm_xor_si128(tmp1,tmp3);
35 tmp4 = _mm_srli_si128(tmp1,0x08);
36 tmp1 = _mm_slli_si128(tmp1,0x08);
```

```
37
38
39 tmp4 = _mm_xor_si128(tmp4,tmp3);//
40 tmp1 = _mm_xor_si128(tmp1,tmp2);//
41
42 reduction();
43 }
44
45 static inline void gfsqr(__m128i a, __m128i *res){
46
47 __m128i tmp0,tmp1,tmp2,tmp3,tmp4,tmp5;
48 __m128i maskl, maskh, table;
49
50 tmp0 = _mm_set_epi32(0x00,0x00,0x00,0xFF);
51 table = _mm_set_epi32(0x55545150,0x45444140,0x15141110,0x05040100);
52 maskl = _mm_set1_epi32(0x0F0F0F0F);
53 maskh = _mm_set1_epi32(0xF0F0F0F0);
54 tmp1 = _mm_and_si128(a, maskh);
55 tmp2 = _mm_and_si128(a, maskl);
56 tmp1 = _mm_srli_epi64(tmp1, 0x04);
57 tmp1 = _mm_shuffle_epi8(table,tmp1);
58 tmp2 = _mm_shuffle_epi8(table,tmp2);
59 tmp4 = _mm_unpackhi_epi8(tmp2,tmp1);//hi
60 tmp1 = _mm_unpacklo_epi8(tmp2,tmp1);//low
61 reduction();
62 }
63
64 static inline void gfmulby2(__m128i a,__m128i* res){
65 *res = _mm_srai_epi32(a,31);
66 *res = _mm_shuffle_epi32(*res,0x57);
67 *res = _mm_and_si128(*res,_mm_set_epi32(0x00,0x01,0x00,0x87));
68 *res = _mm_xor_si128(*res,_mm_slli_epi64(a,0x01));
69 }
```

Listing A.1: Code of basic building blocks.

Appendix B

Experimental Results

M1:

	Encryption		**Decryption**	
	gcc	icc	gcc	icc
EME2	8.27	8.15	8.25	8.10
HEH[p]	3.11	3.39	3.04	3.38
HMCH[p]	3.15	3.40	3.15	3.39
XCB	**3.18**	3.44	3.12	**3.44**
HCTR	3.13	**3.45**	**3.19**	3.44
HEH[B]	5.44	5.82	5.50	5.82
HMCH[B]	5.57	5.87	5.54	5.88
HMCH2	**5.82**	6.08	**5.80**	6.07
HCTR*	5.72	**6.15**	5.72	**6.15**

M2:

	Encryption		**Decryption**	
	gcc	icc	gcc	icc
EME2	8.763	9.358	8.552	8.953
HEH[p]	3.311	3.402	3.324	3.397
HMCH[p]	3.483	3.478	3.478	3.483
XCB	**3.558**	**3.548**	**3.558**	**3.543**
HCTR	3.538	3.483	3.538	3.483
HEH[B]	5.499	6.200	5.536	6.169
HMCH[B]	5.962	6.526	5.990	6.509
HMCH2	**6.327**	**6.613**	**6.327**	**6.613**
HCTR*	6.200	6.200	6.200	6.200

(a)

M1:

	Encryption		**Decryption**	
	gcc	icc	gcc	icc
EME2	8.43	8.15	8.43	8.17
HEH[p]	3.13	3.39	3.12	3.39
HMCH[p]	3.15	3.42	3.15	3.43
XCB	**3.22**	**3.49**	**3.22**	**3.49**
HCTR	3.19	3.47	3.19	3.47
HEH[B]	5.52	5.85	5.43	5.85
HMCH[B]	5.57	5.95	5.57	5.96
HMCH2	**5.85**	6.15	**5.84**	6.14
HCTR*	5.74	**6.17**	5.74	**6.21**

M2:

	Encryption		**Decryption**	
	gcc	icc	gcc	icc
EME2	8.95	9.50	8.92	9.50
HEH[p]	3.37	3.40	3.35	3.40
HMCH[p]	3.51	3.57	3.51	3.57
XCB	**3.63**	**3.62**	**3.63**	**3.60**
HCTR	3.57	3.55	3.58	3.55
HEH[B]	5.73	6.22	5.66	6.20
HMCH[B]	6.15	6.78	6.15	6.78
HMCH2	**6.33**	**6.81**	**6.34**	**6.81**
HCTR*	6.25	6.22	6.23	6.22

(b)

Table B.1: Encryption and decryption implementation throughput results in **Gbit/sec**, for a **4KB** buffer and a **128-bit tweak** using both machines M1 and M2. (a) includes Key Expansion procedure and (b) does not include it.