EXPLOITING MODERN HARDWARE FOR SECURE DATA MANAGEMENT

by

Mustafa Canim

APPROVED BY SUPERVISORY COMMITTEE:

____

Dr. Murat Kantarcioglu, Chair

____

Dr. Bhavani Thuraisingham

____

Dr. Latifur Khan

____

Dr. Kevin W. Hamlen

Copyright 2011

Mustafa Canim

All Rights Reserved

To my family

EXPLOITING MODERN HARDWARE FOR SECURE DATA MANAGEMENT

by

MUSTAFA CANIM, B.S., M.S.

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

May, 2011

PREFACE

This dissertation was produced in accordance with guidelines which permit the inclusion as part of the dissertation the text of an original paper or papers submitted for publication. The dissertation must still conform to all other requirements explained in the "Guide for the Preparation of Master's Theses and Doctoral Dissertations at The University of Texas at Dallas". It must include a comprehensive abstract, a full introduction and literature review and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgement to be made of the importance and originality of the research reported.

It is acceptable for this dissertation to include as chapters authentic copies of papers already published[1], provided these meet type size, margin and legibility requirements. In such cases, connecting texts which provide logical bridges between different manuscripts are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the student's contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the dissertation attest to the accuracy of this statement.

[1] Partial or full content of chapters has been published in [16, 17, 15].

ACKNOWLEDGEMENTS

Throughout my four years at UT Dallas I have been blessed with the company of so many wonderful people that I find it difficult to thank and express my appreciation to every single one of them. Nevertheless, I would like to voice my deepest gratitude to some of them who have left indelible traces on my personality, my scholarly development, and my work. Firstly, I would like to give special thanks to my advisor, Dr. Murat Kantarcioglu, for his continuous guidance and support throughout my Ph.D. years. He has been inspirational and enlightening at all stages of my dissertation. I doubt that this work would have been brought to fruition absent his understanding and tolerance. Next, I would like to thank my committee members Dr. Bhavani Thuraisingham, Dr. Latifur Khan and Dr. Kevin W. Hamlen for their invaluable inputs. Their encouragement and positive criticism have been of utmost help throughout the entire dissertation process. I would like to extend my gratitude to my colleagues and co-authors at the IBM T.J. Watson research center. In particular I would like to thank George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, Christian A. Lang and Yuan-Chi Chang for their contribution to my development as a scholar and a researcher.

The challenges of living in a foreign country and pursuing a graduate degree in computer science would have been much harder to overcome without my friends. Luckily, I have befriended some of the finest and most valuable people on earth. Especially, I cannot express my gratitude to my long time friend Musa Subasi, with whom Dallas felt like home. It was an honor for me to share the same apartment with him for three years. Ali Inan has been the fatherly figure in Dallas

whose presence gave me confidence that I was safe in this foreign environment. I would like to thank Engin Tozal, Mehmet Kuzu, Selami Ciftci, Huseyin Ulusoy, and Dursun Baran for their friendship.

Thank also to my family, especially my mom Muyesser, my sister Fatma Tuba and my dad, Ridvan. Without them I would be a different person today and probably could not get this degree.

Last but not least, I would like to thank my wife, Huma Nurper, for her understanding and support. Without her it would be very hard for me to cope with the difficulties I had while I was getting closer to my graduation. Her love made me happy and her smile has given me the hope to finish my PhD.

January, 2011

EXPLOITING MODERN HARDWARE FOR SECURE DATA MANAGEMENT

Publication No. ____________________

Mustafa Canim, Ph.D.
The University of Texas at Dallas, 2011

Supervising Professor: Dr. M. Kantarcioglu

Databases and data warehouses facilitate the storage and retrieval of large amounts of information. These systems have been increasingly used by corporations to store data such as unit cost information, customers credit card numbers, or patients medical records that are confidential in nature. This trend has been accompanied by numerous incidents of data theft and has resulted in more corporations preferring to store their sensitive records in an encrypted format. Since traditional cryptographic operations entail significant amount of arithmetic calculations, latencies generally arise while recording or retrieving sensitive data. We propose novel and efficient techniques to minimize the cost of cryptographic operations while processing sensitive information. These techniques involve exploiting the parallel computation capability of multi-core CPUs and IO latencies.

We also present two studies in which we exploit the benefits of a tamper proof cryptographic hardware. In the first study, we discuss how attackers can capture valuable information such as encryption keys from the memories of database systems. We then propose a framework for

reducing the disclosure risk of sensitive information for these types of attacks. The experimental results show that the disclosure risk of such attacks can be reduced dramatically while incurring a small performance overhead.

In the second study, we demonstrate that tamper proof cryptographic hardware can be used for sharing and processing biomedical data. The proposed framework eliminates the need for delegating the storage and processing of sensitive biomedical data to third parties. Within this framework, we define a secure protocol to process genomic data and perform a series of experiments to demonstrate that such an approach can be run in an efficient manner for typical biomedical investigations.

TABLE OF CONTENTS

Preface v

Acknowledgements vi

Abstract viii

List of Tables xiv

List of Figures xv

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 RELATED WORK 7

2.1 Querying Encrypted Data 7

2.2 Authenticity of Query Results 9

2.3 Database Auditing and Forensic Analysis 10

2.4 Encrypted Data Storage Problem From File Systems Perspective 11

2.5 Virtualization Based Solutions 11

2.6 Secure Processing of Genomics Data 12

2.7 Background on Secure Coprocessors 13

CHAPTER 3 DESIGN AND ANALYSIS OF QUERYING ENCRYPTED DATA IN RELATIONAL DATABASES 16

3.1 Introduction 17

3.2 Block Cipher Modes Suitable for Databases 17

3.2.1 Overview of Block Cipher Modes 18

3.2.2 Evaluating the Performance of Encryption Modes under Different Encryption Granularity 20

3.2.3 Performance of Encryption Modes under Different Disk Access Patterns 24

3.2.4 Which Mode? 27

3.3 A New Approach for Storing Encrypted Data in Database Pages 28

3.3.1 Experiments and Analyses 32

3.4 Discussion 35
3.5 Conclusion 36

CHAPTER 4 QUERY OPTIMIZATION IN ENCRYPTED RELATIONAL DATABASES BY VERTICAL SCHEMA PARTITIONING 37

4.1 Introduction 37
4.1.1 Advantages of the Proposed Approach 41
4.2 Preventing CPU Bottleneck by Vertical Partitioning 42
4.2.1 Details of the CPU Bottleneck Experiments 43
4.2.2 Mitigating the Join Cost Due to the Partitioning 46
4.3 Partitioning a Single Relation 47
4.3.1 Formal Definition of the Problem 47
4.3.2 Experiment Results 48
4.4 Partitioning Multiple Table 50
4.4.1 Formal Definition of the Problem 50
4.4.2 One Step at a Time (OSAT) Heuristic 51
4.4.3 Experiment Results 52
4.5 Conclusion 54

CHAPTER 5 BUILDING DISCLOSURE RISK AWARE QUERY OPTIMIZERS FOR RELATIONAL DATABASES 56

5.1 Introduction 56
5.1.1 Threat Model and System Overview 59
5.2 Disclosure metrics for memory-based attacks 64
5.2.1 Disclosure Risk for Single-relation Queries 66
5.2.2 Disclosure Risk for Multi-relation Queries 70
5.3 Multi-Objective Query Optimization 72
5.3.1 Combining two cost measures 72
5.3.2 Modifications in MySQL Query Optimizer and InnoDB Storage Engine 73
5.3.3 An Alternative Implementation Method for Constraint Approach 76
5.4 Experiments 77
5.4.1 Experiments with TPC-H workload 77
5.4.2 Performance Overhead of Key Generation and Audit Log Generation Mechanism 81
5.4.3 Performance Overhead of Key Generation and audit log generation mechanism 81

5.4.4 Hardware & Software Specifications 83
5.4.5 Preparation of TPC-H database instance and query workload 83
5.5 Discussion 85
5.5.1 Motivating Example For Query Optimization 85
5.5.2 Granularity of Key Management 85
5.5.3 Optimal Predicate Ordering 89
5.6 Conclusion 90

CHAPTER 6 SECURE MANAGEMENT OF CLINICAL GENOMICS DATA WITH CRYPTOGRAPHIC HARDWARE 92
6.1 Introduction 92
6.2 Description of the Proposed Framework 95
6.2.1 Architecture 96
6.2.2 Data Representation 99
6.2.3 Secure Count Queries 101
6.3 Security Analysis 103
6.4 Joining Patient Records 104
6.5 Experiments and Results 106
6.5.1 Join Operation 107
6.5.2 Selection Operation 108
6.6 Discussion 109
6.7 Conclusion 111

CHAPTER 7 CONCLUSION 112

Bibliography 114

Vita

LIST OF TABLES

3.1 Notations used for describing block cipher modes 19
3.2 Encryption of 1 GB data under different block sizes 23
3.3 Results of experiment 3 (reading and decrypting 1 GB file sequentially) 25
3.4 Results of experiment 4 (reading one GB file randomly and decrypting 512 MB) . . 25

4.1 Query execution times for TPC-H queries 5-7-10 and the workload execution times on different partitioning scenarios 53

5.1 Sample query 66
5.2 Notations used for disclosure model 67

6.1 Encryption of Clinical Genomics Data 100

LIST OF FIGURES

1.1 Semi trusted attack model 3
1.2 Fully untrusted attack model 4

2.1 IBM 4764 PCI-X Cryptographic Coprocessor 14

3.1 Effect of key initialization under different granularity 24
3.2 Illustration of page level CTR approach 31
3.3 Experiment results for projection queries 34
3.4 Experiment results for selection queries 34
3.5 Experiment results for selection queries 34
3.6 Experiment results for selection queries 34

4.1 Left deep join tree of TPC-H query-10 42
4.2 Multi table partitioning over indexed tables 46
4.3 Effect of CPU bottleneck on encrypted query processing 46
4.4 Various distributions of type 1 and type 2 queries. 49
4.5 Various distributions of type 3 queries. 49

5.1 System overview 62
5.2 Possible execution plans for a given query 65
5.3 Comparison of query optimizer algorithms 78
5.4 a) Number of SCP accesses during the execution of the workload b) Constraint on disclosure and constraint on performance c) Weighted aggregation approach 79
5.5 Performance overhead of key generation for different granularities 82
5.6 Query execution plans 86
5.7 Execution times of operations 88

6.1 Proposed framework for management of biomedical data in third party cryptographic hardware. 95
6.2 An overview of the secure count protocol. 102

6.3 Execution time of the join operation for various buffer sizes 108
6.4 Execution time of the join operation for various buffer sizes 108
6.5 Execution time for count queries on various datasets with different query sizes (SCP based protocol) 109
6.6 Improvement rate compared to the multiple third party protocol in [50] 110

CHAPTER 1

INTRODUCTION

Databases have become the primary means through which large organizations such as hospitals and financial institutions store and access their valuable customer related information. The highly proprietary nature of data stored in databases has also made these systems the primary target of attackers who want to steal and take advantage of customer specific information. According to a recent report [88], approximately 285 Million records have been stolen from databases in 2008. The same report also indicates that approximately 4.3 million of those records contain personally identifiable information [88]. Another study suggests that companies spend on average between $90 and $305 per lost record after a data theft incident for various forensic, legal, and IT costs [32].

In order to reduce the effect of various attacks and alleviate the disclosure risk in case of a data theft, organizations prefer to store confidential data in an encrypted format in databases. Also recently, legal considerations [26] and new legislations such as California's Database Security Breach Notification Act [74] require companies to encrypt sensitive data. However cryptographic operations that need to be performed so as to securely store sensitive data entail significant amount of cumbersome arithmetic calculations. Accompanied with bad design choices, expensive cryptographic operations needed for querying and processing encrypted data could substantially decrease the performance of a database system.

The goal of our work is to propose alternative solutions to process *encrypted sensitive data* without compromising on the efficiency of the system. In order to minimize the performance

overhead of the cryptographic operations, we present solutions that exploit the hardware characteristics of the underlying architecture. We study this problem under different attack models and propose techniques that provide a significant enhancement in the performance of the system. The attack models that we employ are the *semi trusted attack model* and the *fully untrusted attack model*. In the semi trusted attack model, it is assumed that the server where the queries are executed is trusted and only the disk (storage device) is vulnerable to compromise (See Figure 1.1). In other words, the adversary can only see the files stored at the disk but not the access patterns. Also we assume that the adversary does not have access to the content of the memory. In this case, we only need to satisfy the security under chosen plain text attacks [11]. Put differently, we guarantee that (assuming the blockcipher used (e.g. AES) is secure) by looking at the contents of the disk, any polynomial-time adversary will have negligible probability of learning anything about the sensitive data. This is the most commonly considered attack model in the conventional database systems. In Section 3 and Section 4 we consider a framework where this attack model is considered and propose techniques to deal with the performance issues during query processing.

In the fully untrusted attack model, we assume that the adversary can monitor the content of both the memory and the storage devices (See Figure 1.2). In this model, the only trusted component of the system is the secure coprocessor (SCP). A typical SCP is a single-board computer consisting of a CPU, memory and special-purpose cryptographic hardware contained in a tamper-resistant shell. The device enables developers to implement custom applications and load it to the SCP for execution. Thanks to the special protective hardware, any sensitive encrypted data could be processed within the SCP in any untrusted platform. We provide more information about the details of the secure coprocessors in Section 2.7. In Section 5, we consider a framework where the memory is considered to be untrusted for a certain amount of attack time and the disk is always

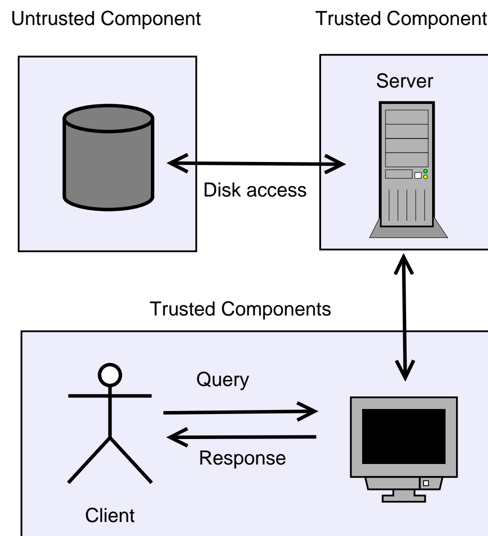

Figure 1.1. Semi trusted attack model

monitored by the adversary. Under this threat model we show how to handle the cryptographic operations while processing the user queries. In Section 6, we assume that both the memory and the disk is vulnerable to attacks at anytime during the execution of the workloads. This is the strictest model one could consider since the sensitive data should never be stored and processed in plain text format in the memory of the system. Instead, it should be decrypted and processed in the secure processor whenever it is needed.

Below we provide a summary of the problems we address and the solutions we propose to each of these problems in this thesis.

In Section 3 and Section 4, we focus on the performance aspect of processing encrypted data. As we mentioned earlier companies prefer to keep their data in encrypted format for data-at-rest due to the security concerns as well as legal considerations. However, storing the data in an encrypted format entails non-negligible performance penalties while processing queries. Both of these sections target several design issues related to querying encrypted data in relational

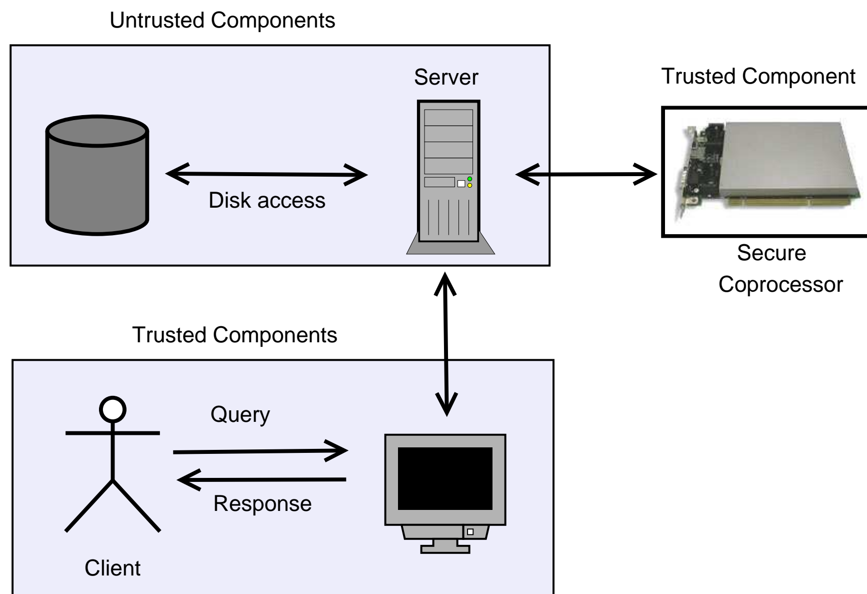

Figure 1.2. Fully untrusted attack model

databases under semi trusted attack model. In Section 3, we propose new and efficient techniques to reduce the cost of cryptographic operations while processing different types of queries. These techniques enable us not only to overlap the cryptographic operations with the IO latencies but also to reduce the number of block cipher operations with the help of selective decryption capabilities. In Section 4, we first present some experiment results conducted on benchmark datasets proving that excessive decryption costs during query processing result in CPU bottleneck. As a solution we propose a new method based on schema decomposition that partitions sensitive and non-sensitive attributes of a relation into two separate relations. Our method improves the system performance dramatically by parallelizing disk IO latency with CPU-intensive operations (i.e., encryption/decryption).

In Section 5, we consider a fully untrusted attack model for a relational database where a malware can monitor both the content of the memory and the disk for a certain amount of time. As we will discuss the details in this chapter, many DBMS products in the market provide built

in encryption support for disk encryption. This solution is quite effective in preventing data leakage from compromised/stolen storage devices. However, recent studies show that a significant part of the leaked records have been done so by using specialized malwares that can access the main memory of systems. These malwares can easily capture the sensitive information that are decrypted in the memory including the cryptographic keys used to decrypt them. This can further compromise the security of data residing on disk that are encrypted with the same keys. In this section, we quantify the disclosure risk of encrypted data in a relational DBMS for main memory-based attacks and propose modifications to the standard query processing mechanism to minimize such risks. Specifically, we propose query optimization techniques and disclosure models to design a data-sensitivity aware query optimizer. We implemented a prototype DBMS by modifying both the storage engine and optimizer of MySQL-InnoDB server. The experimental results show that the disclosure risk of such attacks can be reduced dramatically while incurring a small performance overhead in most cases.

In Section 6, we consider a fully untrusted attack model for a client server architecture which can be used for processing biomedical genomic data. Nowadays, the biomedical community is increasingly migrating toward research endeavors that are dependent on large quantities of genomic and clinical data. Also, various regulations require that such data be shared beyond the initial collecting organization (e.g., an academic medical center). It is of critical importance to ensure that when such data is shared, as well as managed, it is done so in a manner that upholds the privacy of the corresponding individuals and the overall security of the system. In general, organizations have attempted to achieve these goals through *de-identification* methods that remove explicitly, and potentially, identifying features (e.g., names, dates, and geocodes) [14, 24, 31, 36, 41]. However, recent studies demonstrate that de-identified data can be re-identified to named individ-

uals using simple automated methods [59, 12]. As an alternative, one could design a framework that leverages tamper proof hardware to enable secure clinical genomics data storage and processing at a single third party. Since the hardware is capable of performing local computations that are completely hidden from the server, the sensitive data could be stored in a database within an SCP. Despite such enhanced security, however, current SCPs are limited in memory size and computation power. For example, the secure coprocessor used in this work has only 64 MB and a 266MHz PowerPC processor. Yet, GWAS is often executed over large quantities of data. For instance, a typical GWAS study computes the correlation between 600,000 (or more) single nucleotide polymorphisms (SNPs) and a phenotype for 10000 (or more) individuals, which is approximately 1.5 GB of data. It is not possible to fit all of the data into the SCP's memory at the same time. In Section 6, we will explain how, given its limitations, such hardware can be leveraged to craft a secure solution for clinical genomics data processing by a single third party.

CHAPTER 2

RELATED WORK

In this section, we summarize the literature work relevant to our studies presented in the following sections. We also provide a detailed background information about the secure coprocessors we use in our studies.

2.1 Querying Encrypted Data

Querying encrypted data under the untrusted database server attack model was first suggested in [39]. Hacigumus et al. suggested partitioning the client's attribute domains into a set of intervals [39]. The correspondence between intervals and the original values are kept at the client site and encrypted tables with interval information are stored in the database. Efficient querying of the data is made possible by mapping original values in range and equality queries to corresponding interval values. In subsequent work, Hore et al. [43] analyzed how to create the optimum intervals for minimum privacy loss and maximum efficiency. In [23], the potential attacks for interval based approaches were explored and models were developed to analyze the trade-off between efficiency and disclosure risk. In [4], Aggarwal et al. suggested to allow the client to partition its data across two, (and more generally, any number of) logically independent database systems that cannot communicate with each other. The partitioning of data is performed in such a fashion as to ensure that the exposure of the contents of any one database does not result in a violation of privacy. The client executes queries by transmitting appropriate sub-queries to each database, and then piecing together the results at the client side. This line of work is different from the problem

we discuss in Section 3 and Section 4 because none of these studies assume that the sensitive information is processed in plain text format in the server side. On the other hand, interval based techniques could be used for the attack models used in Section 5 and Section 6, however, the data processing capability of the server in the interval based solutions is quite limited. Since the server cannot see the data in plain text format, most of the data records are sent to the client for further processing which increases both the network cost and client side processing cost.

Encrypted data storage problem was also studied in [6, 49, 29, 84]. In [6], Agrawal et al. suggested a method for order preserving encryption (OPES) for efficient range query processing on encrypted data. Unfortunately, OPES only works for numeric data and is not trivial to extend to discrete data. In [49], Iyer et al. suggested data structures to store and process sensitive and non-sensitive data efficiently. The basic idea was to group encrypted attributes on one mini page (or one part of the tuple) so that all encrypted attributes of a given table can be decrypted together. In [29], Elovici et al. also suggested a different way for tuple level encryption. In [10], Bayer and Metzger explored the idea of using block ciphers and stream ciphers for encrypting B+ trees and random access files. They suggest generating different keys for each page based on the page id to break potential correlation attacks (pages encrypted with the same key and the same data will have the same cipher text). The disadvantage of this approach is, changing keys for each page imposes big key initialization costs. Hardjono and Seberry [42] suggested special combinatoric structures to disguise keys in B+ trees. Unfortunately, their combinatoric approach has not received the level of scrutiny required to achieve trust (as have others such as AES).

Our consideration is different from the ones described above in many aspects. Unlike the previous work, we provide complete analysis of block cipher modes suitable for databases, present

analysis of the experiments about overlapping the IO latencies with the cryptographic operations by using multi-threading and propose a new approach for storing encrypted data in database pages by utilizing selective decryption property of CTR mode, in Section 3. Also in Section 4, we discuss the CPU bottleneck problem in encrypted databases with experimental observations, show how to utilize the page level approach for selective decryption of sensitive data, and propose a new workload dependent vertical schema decomposition technique to mitigate the negative impacts of cryptographic operations.

2.2 Authenticity of Query Results

Some of the existing work deal with the problem of authenticity of query results in untrusted database server model. In [85] the authors propose a method for proofs of actual query execution in an outsourced database framework, in which a client outsources its data management needs to a specialized provider. They introduce query execution proofs; for each executed batch of queries the database service provider is required to provide a strong cryptographic proof assuring that the queries were actually executed correctly over their entire target data set. In [91] an integrity audit mechanism is proposed to assure that the query results returned by the service provider are both correct and complete. According to this approach the data owner inserts a small amount of records into an outsourced database so that the integrity of the system can be effectively audited by analyzing the inserted records in the query results. The authors in [28] propose some techniques based on Merkle hash trees to guarantee the authenticity of the answer of queries provided by third party service providers. The following studies also study different angles of the same problem: [92, 53, 61, 72, 65, 71, 93, 9, 52, 33]. Such approaches could be also applicable in the frameworks described in this thesis but we do not consider those as a part of our project.

2.3 Database Auditing and Forensic Analysis

There are many work that address several problems regarding database auditing and forensic analysis and provide alternative solutions to deal with them. Most of these studies focus on detection and protection of database tampering. In [82] Schneier et al. present a general scheme that allows keeping an audit log on an insecure machine, so that log entries are protected even if the machine is compromised. In [38] a special mechanism is proposed to certify the creation and modification of digital documents such as text, video and audio files. They propose two solutions both involve the use of one way hash functions whose outputs are processed in lieu of the actual documents, and of digital signatures. In [87] Stahlberg et al. investigate the problem of unintended retention of data in database systems, and build a database system that is resistant to unwanted forensic analysis. They show how data remnants in databases pose a threat to privacy. They propose specific techniques such as secure record deletion and log expunction to tackle with this problem. In [73] Pavlou et al. focus on the problem of determining when the tampering occurred, what data was tampered with, and who did the tampering, via forensic analysis. They present four forensic analysis algorithms: the Monochromatic, RGBY, Tiled Bitmap, and a3D algorithms, and characterize their ”forensic cost“ under worst-case, best-case, and average-case assumptions on the distribution of corruption sites. In [86] a security mechanism within a DBMS is proposed that prevents an intruder within the DBMS itself from silently corrupting the audit logs. The solution they propose involves use of cryptographically strong one-way hash functions. In the proposed mechanism the DBMS stores additional information to enable a separate audit log validator to examine the database along with this extra information and state conclusively whether the audit log has been compromised. Unlike our work presented in Section 5, none of these studies address the problem of detecting and minimizing what is revealed in databases during a memory attack.

2.4 Encrypted Data Storage Problem From File Systems Perspective

The following studies focus on encrypted data storage issues from file systems perspective: [13, 62, 63, 22]. In [13] a Unix based file system called CFS is introduced. CFS supports secure storage at the system level through a standard Unix file system interface to encrypted files. Users associate a cryptographic key with the directories they wish to protect. Files in these directories are transparently encrypted and decrypted with the specified key without further user intervention. In [62] Mazières et al. propose a secure file system designed to span the internet. To tackle with the key management issues they propose file names and path names to be used as public keys during the communications. This avoids the necessity for assigning file names to encryption keys which is an inefficient technique used in other secure file systems. In [63] the authors discuss the implementation of a trusted network file system on an untrusted server particularly focusing on the detection of tampering attacks and stale data. Some attack types and vulnerabilities in the modern file systems are also discussed in [22] in great detail.

In [20], Chow et al. discuss data lifetime issue in modern operating systems. They present a strategy for reducing the lifetime of data in memory called secure deallocation. With secure deallocation they suggest zeroing data either at deallocation or within a short, predictable period afterward in general system allocators (e.g. user heap, user stack, kernel heap). They show that this substantially reduces data lifetime with minimal implementation effort in the existing systems. In Section 5, we make use of some of the analysis presented in this paper.

2.5 Virtualization Based Solutions

Some other studies in the literature suggest virtual system based solutions to the data security problem. In [19], Chen et al. introduce a virtual-machine-based system called Overshadow that

protects the privacy and integrity of application data in the event of a system compromise. Overshadow presents an application with a normal view of its resources, but the OS with an encrypted view. This allows the operating system to carry out the complex task of managing an application's resources, without allowing it to read or modify them. In [51], Krishnan and Monrose suggest a framework for transparently recording accesses to protected storage in a virtualized environment. In this framework a forensic layer records all transactions in a version-based audit log that allows for faithful reconstruction of accesses to the data store over time. Virtualization based solutions provide strong security against the system attacks however these mechanisms are still susceptible to cold boot attacks which constitute a severe threat as discussed in [58, 40]. Nevertheless the techniques described in these studies can be combined with the security mechanisms that we are suggesting to provide a better protection against all types of attack scenarios.

2.6 Secure Processing of Genomics Data

Secure cryptographic devices have been used for many tasks, including private information retrieval [8], private record linkage [5, 46], and tamper-resistant intrusion detection [90]. Recently, Trusted Computing Platform Alliance chips were used to protect outsourced private customer data [64]. To our knowledge, none of the previous work applies secure cryptographic hardware for managing or processing clinical genomics data, which has distinct issues in terms of scale and type.

In [47], Illiev et al. propose a system for providing obliviousness for arbitrary large computations on secure coprocessors for any algorithm using circuit evaluation methods in the context of SMC. In Section 6, we provide much more efficient solutions for processing genomic data in untrusted platforms.

Not only the cryptographic hardware but also the cryptographic techniques were used for processing of genomics data. In [50], Kantarcioglu et al. introduces a cryptographic framework that can be leveraged for secure clinical genomics data storage. In this framework, the clinical genomics data records are first encrypted by the data owners (e.g., hospitals) and stored in a centralized repository. Using a specific type of public key encryption, the queries of biomedical researchers (e.g. count queries) are executed by the repository managers, without decrypting the clinical genomics data. Instead, only the aggregated result is decrypted.

This framework provides a cryptographically secure way of executing the queries. However, this protocol required the integration of two third parties to achieve such computations, which, as we alluded to in Section 6, is not always practical. Additionally, excessive processing cost of specialized encryption techniques can make this mechanism impractical for very large datasets. In Section 6, we provide a much more efficient solution by leveraging the capabilities of secure cryptographic hardware.

2.7 Background on Secure Coprocessors

The tamper proof hardware we use in Section 5 and Section 6 is already commercially available in the form of secure coprocessors. An example of an SCP is the IBM 4764 Cryptographic coprocessor [44].

This is a single-board computer consisting of a CPU, memory and special-purpose cryptographic hardware contained in a tamper-resistant shell. The hardware is certified to level 4 under FIPS PUB 140-1 [1]. When installed on a server, it is capable of performing local computations that

[1]FIPS PUB 140-1 is the benchmark standard for evaluating the security and proper algorithmic implementation of a commercial cryptographic product. Under the supervision of the USA and Canadian

Figure 2.1. IBM 4764 PCI-X Cryptographic Coprocessor

are completely hidden from the server. If tampering is detected, then the SCP clears the internal memory. Despite such enhanced security, current SCPs are limited in memory size and computation power. For example, IBM 4764 Cryptographic coprocessor has only 64 MB and a 266MHz PowerPC processor. Therefore, secure coprocessors are mostly used for handling key management tasks rather than being used for bulk encryption and decryption operations. In Section 5, we describe how to use the secure coprocessor for generating low granularity keys to encrypt sensitive data records. In Section 6, we use the SCP for performing all operations on the sensitive records. We implement some buffering techniques to speed up the performance of the operations.

The coprocessor has three “segments”: Segment 1, Segment 2, and Segment 3. Each segment has a status, an owner identifier and holds software, a validation public key. Segment 1 has “miniboot” which contains dignostics and code loading controls. Software in this segment administers the replacement of software already loaded to Segment 1 and administers the loading of data and software to segments 2 and 3. This segment is loaded at the factory, but can be replaced

---

Governments, independent laboratories conduct thorough analyses of the product design and actual tests of products. The test report is discussed with the governmental bodies, and when found acceptable, a certificate is issued [44].

using the CLU utility. Segment 2 has an embedded control program, i.e., a mini operating system. The operating system supports applications loaded into Segment 3. Segment 3 has either CCA or a custom created application. The coprocessor Support Program includes a CCA application program that can be installed into Segment 3. The application functions according to the IBM CCA and performs access control, key management, and cryptographic operations [1].

IBM 4764 secure coprocessor comes with a programming API called "Common Cryptographic Architecture (CCA) application programming interface (API)" that enables programmers to develop applications that communicate with the secure coprocessor. CCA provide a variety of cryptographic processes and data-security techniques [1]. The application developed by the programmers can call verbs to perform the following functions:

- Data confidentiality: Encrypt and decrypt information, typically using the AES or DES algorithms in Cipher Block Chaining (CBC) mode to enable data confidentiality.

- Data integrity: Hash data to obtain a digest, or process the data to obtain a message authentication code (MAC), that is useful in demonstrating data integrity.

- Non-repudiation: Generate and verify digital signatures to demonstrate data integrity and form the basis for non-repudiation.

- Authentication: Generate, encrypt, translate, and verify finance industry personal identification numbers (PINs) and American Express, MasterCard and Visa card security codes with a comprehensive set of finance-industry-specific services.

- Key management: Manage the various AES, DES, and RSA (PKA) keys necessary to perform the above operations.

CHAPTER 3

DESIGN AND ANALYSIS OF QUERYING ENCRYPTED DATA IN RELATIONAL DATABASES

Cryptographic operations, required to store sensitive data securely, entail significant amount of cumbersome arithmetic calculations. Coupled with bad design choices, expensive cryptographic operations needed for querying and processing encrypted data could decrease the system performance dramatically. On the other hand, good design choices may drastically affect the overall performance. With this in mind, in this section, we discuss some of the design issues to encrypt and query the sensitive data that resides in database disks. We propose new techniques to reduce the cost of cryptographic operations while processing different types of queries. We show that by exploiting the benefits of multi core CPUs, the cryptographic operations could be performed while reading and writing database pages. Also we show that the number of block cipher operations could be reduced with the help of selective decryption capabilities of certain encryption modes.

In this section, we consider a semi trusted attack model for the database system where the queries are executed. As we discussed in Section 1, in the semi trusted attack model, it is assumed that the server where the queries are executed is trusted and only the disk (storage device) is vulnerable to compromise.

3.1 Introduction

To reduce the performance loss that arises from using cryptographic techniques, we analyzed different block cipher modes of operations and compared their performances under various disk access patterns to see which modes are suitable for databases and allow us to parallelize the IO latencies with the cryptographic operations. Based on our experiments and analyses, we suggest using an efficient and provably secure encryption mode of operation that also enables selective decryption of large data blocks. We also propose a new approach for storing and processing encrypted data and we show that by using suitable encryption mode, the additional time needed for processing different types of queries can be significantly reduced.

3.2 Block Cipher Modes Suitable for Databases

To store privacy-sensitive data securely, we need to use encryption methods secure against chosen plaintext attacks. Also we need general solutions that could support all kinds of data that needs to be encrypted. For example, the Order Preserving Encryption (OPES) idea suggested in [6] works for only numeric data.

Securely encrypting any kind of data is well studied in the cryptography domain. Any kind of long data is encrypted using operation modes based on secure block ciphers (e.g, AES[66]). All of these encryption modes process the long data by dividing into fixed size blocks(e.g 16 bytes in AES[66]) that can be processed by the block cipher. The obvious question is which block cipher mode is preferable for encrypting sensitive data in databases. To choose the best

mode possible, we need to analyze the effect of these modes under specific database operation conditions. Specifically we look at:

- **Performance of Encryption Modes under Different Granularity:** We need to encrypt and decrypt the data at different granularity. If we are encrypting at the tuple level, we may need to decrypt small blocks at a time, if we use page level encryption, we need to decrypt potentially a page long encrypted data. Ideally, we should have a mode where different granularity has little effect on the total performance.

- **Performance of Encryption Modes under Different Disk Access Patterns:** We need to have an encryption method that enables efficient decryption under different disk access patterns.

3.2.1 Overview of Block Cipher Modes

Before we briefly give an overview of different block cipher modes, we discuss the notations we use through out the section. Let $E_K()$ be the encryption operation with the key K and $D_K()$ be the decryption operation with the key K. Let P_i be the ith plain text block, C_i be the ith ciphertext block, IV be the initial random vector and b be the block cipher input size (e.g 128 bits for AES[66]). Let $LSB_s(x)$ be the s least significant bits of x and similarly let $MSB_s(x)$ be the s most significant bits of x. Also below, $S[i..j]$ denotes the ith through j^{th} bit of string S , $||$ denotes the string concatenation, and $\oplus$ denotes the bitwise xor operation. Table 3.1 summarizes the notations used for describing block cipher modes.

Table 3.1. Notations used for describing block cipher modes

$E_K()$	Block cipher encryption with the key K
$D_K()$	Block cipher decryption with the key K
b	Block cipher block length in bits
C_i	i^{th} ciphertext block
P_i	i^{th} plain text block
$LSB_s(x)$	s least significant bits of x
$MSB_s(x)$	s most significant bits of x
$S[i..j]$	i^{th} through j^{th} bit of string S
$\|\|$	String concatenation
$\oplus$	binary xor operation

Also, for the sake of simplicity, we assume that the plaintext P is n blocks long (i.e. $n \cdot b$ bit long plaintext). Under these assumption, the block cipher modes of operations suggested by NIST can be summarized as follows:(The details can be found in FIPS-SP 800-38A [2])

- **Electronic Code Book(ECB):** In ECB mode, each block of the plaintext is encrypted independently. Similarly each block of the ciphertext decrypted independently. Unfortunately, this mode is not secure since it reveals distribution information. (Note that if $P_i = P_j$ then $C_i = C_j$)

ECB Encryption:	**ECB Decryption:**
$C_i = E_K(P_i),\ i = 1..n$	$P_i = D_K(C_i),\ i = 1..n$

- **Cipher Block Chaining(CBC):** Each block of the ciphertext is created by encrypting the result of the previous ciphertext block xored with the current plaintext block.

CBC Encryption:	**CBC Decryption**
$C_1 = E_K(P_1 \oplus IV)$	$P_1 = D_K(C_1) \oplus IV$
$C_i = E_K(P_i \oplus C_{i-1}),\ i = 2..n$	$P_i = D_K(C_i) \oplus C_{i-1},\ i = 2..n$

- **The Cipher Feedback Mode:(CFB)** In this mode, successive segments (let s be the size of these segments where $1 \leq s \leq b$ and C_i^s, P_i^s denote the s-bit sized segments of ciphertext and plaintext respectively.) of ciphertext are concatenated to create an input for forward cipher to create blocks that are xored with plaintext segments.

CFB Encryption:
$I_1 = IV$
$I_i = LSB_{b-s}(I_{i-1})|C_{i-1}^s, i = 2..n$
$O_i = E_K(I_i), \quad i = 1..n$
$C_i^s = P_i^s \oplus MSB_s(O_i), \quad i = 1..n$

CFB Decryption:
$I_1 = IV$
$I_i = LSB_{b-s}(I_{i-1})|C_{i-1}^s, i = 2..n$
$O_i = E_K(I_i), \quad i = 1..n$
$P_i^s = C_i^s \oplus MSB_s(O_i), \quad i = 1..n$

- **Output Feedback Mode(OFB):** This mode is very similar to CFB with s taken as b. In this simplified description of OFB, we assume that ciphertext is $n * b$ bits long.

OFB Encryption:
$I_1 = IV$
$I_i = O_{i-1}, \quad i = 2..n$
$O_i = E_K(I_i), \quad i = 1..n$
$C_i = P_i \oplus O_i, \quad i = 1..n$

OFB Decryption:
$I_1 = IV$
$I_i = O_{i-1}, \quad i = 2..n$
$O_i = E_K(I_i), \quad i = 1..n$
$P_i = C_i \oplus O_i, \quad i = 1..n$

- **Counter Mode(CTR):** In this mode a counter(ctr) is incremented and the encrypted counter value is xored with plaintext block to get the ciphertext block.

CTR Encryption:
$C_i = P_i \oplus E_K(ctr + i), i = 0..n - 1$

CTR Decryption:
$P_i = C_i \oplus E_K(ctr + i), i = 0..n - 1$

3.2.2 Evaluating the Performance of Encryption Modes under Different Encryption Granularity

In [49], Iyer et al. states that encrypting the same amount of data using few encryption operations with large data blocks is more efficient than many operations with small data blocks. In order

to support this claim, they conduct an experiment using three ciphers: AES[66], DES[69] and Blowfish[80] under different data blocks: 100 bytes, 120 bytes, 16 KBytes. Based on the results of this experiment, they conclude that encrypting data few small units at a time takes longer than encrypting the same amount of data using larger data blocks.

As they stated in their paper [49], the key initialization costs constitute the most important cause of the high amount of time spent in encrypting small blocks of data. We expect to observe less time difference when the data is encrypted under different granularity if we can use **one key** per database table. Using one key per database table is feasible for many practical applications, because we can encrypt 2^{128} bits of data securely [55] using CTR mode of encryption with a given key. To test this view, we conducted some experiments to investigate the effect of key initialization. These experiments are performed under different encryption modes to see which mode is more appropriate to use under different encryption granularity.

In our experiments we used the OpenSSL [21] crypto library's CBC, CTR, OFB and CFB implementation of AES[66]. We chose AES because it is the current standard and supported by various companies. ECB is not used since it is not regarded as a secure mode of operation. We also implemented a slightly modified version of CTR (referred as CTR4). Modified CTR implementation encrypts all the counter values first (i.e. calculates all the $E_K(ctr + i)$ first) and then xors it with the plain text data. Algorithms 1 and 2 describe the details of the modified CTR implementation. In Algorithm 1, we first create a string S using the encrypted counter values and then xor the first l bit of S with plaintext message P to get the ciphertext C. Since encrypted counter values are xored with the plaintext, we do not need to do any padding if the size of the plaintext is not the multiple of the block-cipher length b.

Require: Plain text P with length l, initial counter value ctr, and block-cipher length b.
Set $S = E_K(ctr+0)||E_K(ctr+1)||\ldots||E_K(ctr+\lceil\frac{l}{b}\rceil-1)$
Set $C = P \oplus S[0..l-1]$
return (ctr, C)

Algorithm 1: Modified CTR Encryption (CTR4)

Similarly, in Algorithm 2, we first create a string S using the encrypted counter values and then xor the first l bit of S with ciphertext C to get the plaintext P

Require: Ciphertext C with length l, initial counter value ctr, and block-cipher length b.
Set $S = E_K(ctr+0)||E_K(ctr+1)||\ldots||E_K(ctr+\lceil\frac{l}{b}\rceil-1)$
Return P where $P = C \oplus S[0..l-1]$

Algorithm 2: Modified CTR Decryption (CTR4)

Since all of the block cipher operations are done independent of ciphertext data, CTR4 decryption can be executed in a multi-threaded fashion. One thread could be used for creating the encrypted counters (i.e., for computing S in Algorithm 2), and other thread could be used to read the encrypted data from the disk (i.e. reading C from the disk in Algorithm 2). The original implementation of CTR mode in OpenSSL crypto library calculates C_{i-1} before calculating $E_K(ctr+i)$. Thus it is not suitable for multi-threading.

Experiments:

All of our experiments are conducted on a 2.79 GHz Intel Pentium D machine with 2 GB memory on Windows XP platform. We ran each experiment five times and reported the average results.

In the first experiment, we encrypted a total of one GB data which is cached in memory 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, and 8192 bytes at a time. In this experiment, the same key is used for encrypting the entire one GB data, and the key initialization is done only

once. In other words, AES_set_encrypt_key() function is called once. [1] In Table 3.2, we report the results of experiment 1 in seconds.

Table 3.2. Encryption of 1 GB data under different block sizes

	Experiment 1 (**With** key initialization)					Experiment 2 (**Without** key initialization)				
Block size (Byte)	CBC	CTR	OFB	CFB	CTR4	CBC	CTR	OFB	CFB	CTR4
16	24.98	26.19	27.09	25.64	23.02	49.27	50.77	50.50	50.84	47.91
64	24.31	25.75	26.52	25.03	22.81	30.44	32.13	31.41	31.53	29.16
128	23.78	25.56	26.27	25.20	22.31	26.92	28.83	28.27	28.58	25.44
256	23.56	25.50	26.39	25.44	22.11	25.17	27.42	26.61	27.09	23.95
512	23.42	25.50	26.48	25.41	22.17	24.34	26.52	25.89	26.22	23.23
1024	23.38	25.52	26.47	25.39	22.03	24.00	26.23	25.45	25.75	22.78
2048	23.33	25.48	26.55	25.42	22.05	23.78	25.98	25.23	25.66	22.63
4096	23.36	25.53	26.34	25.38	22.28	23.72	25.84	25.25	25.34	22.78
8192	23.25	25.56	26.39	25.41	22.30	23.53	25.84	25.14	25.42	22.75

In the second experiment, we again encrypted one GB of data similar to experiment 1 but in this experiment, key initialization is done at the beginning of each data block. Average results of experiment 2 are shown in Table 3.2 in seconds.

Experiment 1 indicates that CTR4 mode is the fastest, if the key initialization done just once at the beginning of the encryption process. Also, if we compare the results of experiment with or without key initialization for CTR4 (shown in figure 3.1), we can observe that encrypting larger blocks requires less amount of time if the key initialization is performed every time before encrypting each data block. However, the time required to encrypt the data does not depend on block size if key initialization is done just once. In other words, if the blocks used are not too small (i.e. larger or equal to 64 bytes) and one key is used per database table, the encryption block size does not effect the total decryption times. Actually, it is easy to see this fact from the API of

[1]Initialization means generating substitution tables based on the secret key and this can be a costly operation as reported in [49].

modern crypto packages like the OpenSSL Library. For example in OpenSSL, to start encrypting with a key (similar for decryption), the key initialization function (AES_set_encrypt_key()) should be called first. After the initialization is done, encryption function can be called on various size blocks.

At the same time, experiment 1 indicates that encrypting small blocks (i.e. less than 64 bytes) at a time could be a problem. Fortunately, clever usage of CTR4 mode can solve the small block size problem. (This is not possible with other modes) Since each block is encrypted independently, we can combine blocks from different tuples (i.e., combine 4 blocks from 4 different tuples to create a 64 byte block) and decrypt/encrypt them together.

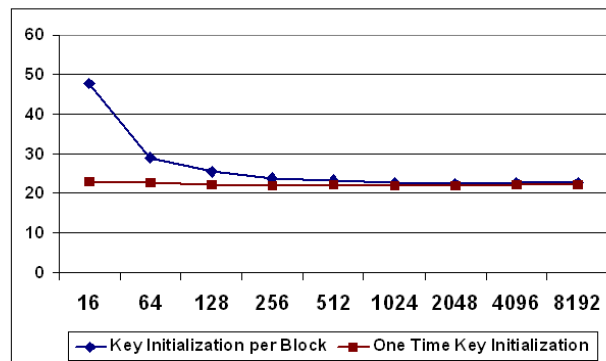

Figure 3.1. Effect of key initialization under different granularity

3.2.3 Performance of Encryption Modes under Different Disk Access Patterns

In this part we conducted four experiments to evaluate the performance of encryption modes under random and sequential disk access patterns. In each experiment, data is accessed 4 KB page at a time, since it is the default page size in many DBMSs such as IBM DB2 [45]. We describe those four experiments below.

- **Experiment 3:**

  In the first part, one GB of encrypted file cached in memory is read **sequentially** and decrypted using different encryption modes. In the second part, the same experiment is repeated with the same file which is accessed from disk (i.e. file is not cached in memory). The results of the experiment are shown in Table 3.3.

Table 3.3. Results of experiment 3 (reading and decrypting 1 GB file sequentially)

Crypto Mode:	CBC	CTR	OFB	CFB	CTR4
from memory	29.02	31.17	30.92	30.89	27.89
from disk	28.91	31.94	30.5	31.05	27.31

  The results show that whether the data resides in memory or not, does not affect the time for reading and decrypting data sequentially. This implies that during the decryption of the current page disk controller can prefetch the next page.

- **Experiment 4:**

  In the first part, 512 MB of the 1 GB encrypted file is accessed **randomly** 4K page at a time from memory and decrypted using different encryption modes. In the second part, the same experiment is repeated with the same file which is accessed from the disk. The results of the experiment are shown in Table 3.4

Table 3.4. Results of experiment 4 (reading one GB file randomly and decrypting 512 MB)

Crypto Mode:	CBC	CTR	OFB	CFB	CTR4
from memory	15.7	16.73	16.38	16.52	15.03
from disk	262.63	262	271.94	266.3	267.64

  The results of the experiment 4 indicate that random access to the pages causes significant delays if the data is not cached in the memory.

- **Experiment 5:**

  Multi-threaded version of CTR4 [2] is used to decrypt the 512 MB **randomly** accessed data, which took 257.3 seconds in the average. This experiment is not performed with the other modes since they do not allow multi threading during decryption.

  If the result of experiment 5 is compared with experiment 4, we can see that multi threaded version of CTR4 reduces the decryption cost significantly, by overlapping the IO operations with decryption operations. In experiment 4, decrypting 512 MB of 1 GB file randomly takes 266.1 seconds in the average. However, performing the same experiment takes 257.3 seconds in this experiment. Therefore, multi threaded version of CTR4 runs almost 9 seconds faster than other modes of operations in the average.

  In order to support this claim, we have calculated the time to decrypt 512 MB allocated memory space sequentially with CTR4, which took 11.2 seconds to perform. This result indicates that 9 seconds of running experiment 5 is overlapped with IO operations.

  This implies that using hardware accelerators, most of the time required for cryptographic operations can be overlapped with the random access IO operations.

- **Experiment 6:**

  Multi-threaded CTR4 is used to decrypt the 1 GB file **sequentially**. The average of the results indicates that it takes 28.05 seconds to decrypt the file sequentially. When compared with the results of experiment 3, it can be concluded that the time to read and decrypt one GB file sequentially is not significantly reduced by using the multi threaded version of CTR.

[2] i.e One thread reads the data, other thread encrypts the counter values. When both threads are done, their results are xored to get the plain text.

The above results imply that the cost of the cryptographic operations can be overlapped with the cost of the IO operations using fast hardware based multi-threaded implementation of CTR mode, which could improve the performance especially in storage area networks where disk access latency could be higher.

3.2.4 Which Mode?

We would like to have a block cipher mode that is suitable for **efficient** processing of encrypted data in databases. Due to reasons stated below, CTR mode emerges as the **best** choice among classic and proven to be secure block cipher modes for efficiently querying encrypted data. The properties of CTR mode that is useful for database encryption can be given as follows:

- **Efficient Implementation:** In all of our experiments, modified version of CTR (CTR4) was the fastest. Since the encryption of each block is independent, modern processor architecture's properties such as aggressive pipelining, multiple cores, and large number of registers can be utilized for even more efficient implementation [54]. For example, in [54], optimized version of CTR mode is four times faster than the optimized version of CBC mode. Also CTR mode is suitable for parallel processing.

- **Selective Decryption** CTR mode could be used to decrypt arbitrary parts of the plaintext. For example, for each tuple encrypted using CTR mode, during the selection operation, we may just decrypt the selection field first and decrypt the rest if the selection criteria is satisfied. To see why we can decrypt the arbitrary substring of a given ciphertext, consider the algorithm 3. In algorithm 3, first the counter values that are used to encrypt the $P[u..v]$ are calculated. Later on, using the counter values $\lfloor \frac{u}{b} \rfloor$ and $\lceil \frac{v}{b} \rceil$, encrypted counter values are

created. Finally, the appropriate segment of the encrypted counter values are xored with the required part of the ciphertext (i.e. $C[u..v]$) to compute $P[u..v]$.

Other block cipher modes such as CBC, CFB, OFB defined for AES[66] in the FIPS-SP 800-38A[2] standard do not allow for selective decryption because the encryption of a block depends on the encryption of the previous block. This implies that we cannot selectively decrypt the required part of the data. ECB and CTR modes of operation encrypt each block independently. Unfortunately, ECB mode reveals the underlying distribution of the data. Therefore, it does not provide the desired level of security.

- **Preprocessing** In CTR mode, most costly part of the encryption and decryption (evaluating $E_K(ctr + i)$) can be done without seeing the data. Actually, we used this property in Experiment 5 and showed that this can reduce the encryption cost significantly. This is not possible for all the modes except OFB mode but OFB mode does not allow selective decryption.

Require: Ciphertext C with length l, initial counter value ctr, block-cipher length b, decryption start index u and decryption end index v
Set $S = E_K(ctr + \lfloor \frac{u}{b} \rfloor) || \ldots || E_K(ctr + \lceil \frac{v}{b} \rceil - 1)$
return $P[u..v] = C[u..v] \oplus S[(u - \lfloor \frac{u}{b} \rfloor \cdot b)..(v - \lfloor \frac{u}{b} \rfloor \cdot b)]$

Algorithm 3: Decryption of arbitrary substring of ciphertext in CTR mode

3.3 A New Approach for Storing Encrypted Data in Database Pages

Granularity of encryption is another important design issue that needs to be considered for efficient storage of encrypted data. The overall database performance can potentially be affected by small changes in the design choices regarding the way of keeping the records in the pages. Tuple level and page level encryption are the most well known options for this purpose. Alternatively, we can use the mini-page approach suggested in [49].

In tuple level encryption, each tuple is encrypted or decrypted separately. If the database needs to retrieve some of the tuples, there is no need to decrypt all of the tuples in the table.

Page level encryption will correspond to a mechanism where a particular page is completely decrypted when buffer pool needs to access the page from the disk. After some modifications, the page is encrypted again and written to disk.

Mini page level for database encryption was first suggested for encrypted data in [49] based on the work of Ailamaki et.al [7]. In this technique, when a tuple is inserted into a page, its attributes are partitioned and stored in corresponding mini pages on the same page.

Deciding to use one of these approaches depends on two factors:

- the cost of the required block cipher operations under different types of queries
- the amount of modification required to implement these approaches in conventional databases

The mini page level encryption, as discussed in [49], is designed in such a way that the sensitive attributes of the records are encrypted with AES [66] in CBC mode and inserted at the beginning of the page. When a page is read from disk, just the first part of the page is decrypted. Since all the sensitive attributes of the records are located at the beginning of the page, the decryption process continues until all sensitive attributes are decrypted.

Although this idea is a neat solution for encrypting sensitive data, there are two issues that are not addressed. The first issue is that it does not allow selective decryption since the encryption mode is selected as CBC mode. When a projection query needs to get specific attributes among the sensitive attributes, all of the encrypted attributes should be decrypted. This cost will linearly

increase if the length of the sensitive attributes increases proportionally to the total length of the records.

Secondly, the mini page level encryption requires significant amount of modification in the page structure of conventional databases. This is because, the attributes of the records are kept in two different parts of the page rather than in a consecutive order.

In the page level encryption, the whole of the page is encrypted before writing into disk and decrypted before loading into buffer pool. If all the data needs to be kept secret, this level of encryption is appropriate. In addition to that, implementing such an approach requires less modification in the page structure of existing databases. However, this is not preferable since it unnecessarily encrypts nonsensitive data along with sensitive data.

Because of the problems discussed above, we propose a new encryption method which we refer to as page level CTR. This method basically utilizes the selective decryption property of CTR4 and combines the useful aspects of page level and tuple level encryption methods. Unlike the mini page approach, sensitive and nonsensitive attributes of the records are kept consecutively. The decryption is performed for each sensitive attribute of the records separately after a page is read from the disk. Therefore, it resembles to tuple level approach.

In our encryption method, we propose using CTR4 for cryptographic operations. So we need to somehow store the counter values in the page. Since each counter value requires a 16-byte of location, keeping one counter value for each record will entail a nonnegligible storage cost. Fortunately, we can use one counter value per page instead of one record. With respect to storage of counter values, our page level CTR approach is similar to page level encryption. Also

we modified and optimized the increment function of CTR mode to increment counter values as much as needed at a time, instead of just one by one.

The page structure of the proposed method is illustrated in figure 3.2 with an example, where we assume that a projection query requires to read the encrypted attributes of two consecutive records. The start index of the attribute of record 1 is α_1 = 220 bytes and of record 2 is α_2 = 430 bytes, with respect to the beginning of the page. Before starting the decryption, the counter value of the page is read from the beginning of the page. Then this value is incremented by δ_1 where $\delta_1 = \lfloor \frac{\alpha_1}{\beta} \rfloor = \lfloor \frac{220}{16} \rfloor = 13$ with the optimized increment function of page level CTR approach (β = 16 since the AES block size is 16 bytes). After finding the necessary counter value, the CTR4 is used to decrypt the sensitive attribute of record 1. Then, the same operation is repeated for record 2. After reading the counter value from the beginning of the page, it is incremented by δ_2 where $\delta_2 = \lfloor \frac{\alpha_2}{\beta} \rfloor = \lfloor \frac{430}{16} \rfloor = 26$. Using this value, the sensitive attribute of record 2 is decrypted with CTR4.

Here, we illustrated decrypting the sensitive attributes of two records in a page. However, this operation will be repeated for all records in every page, if there is a projection query that needs to read all of the records in a table.

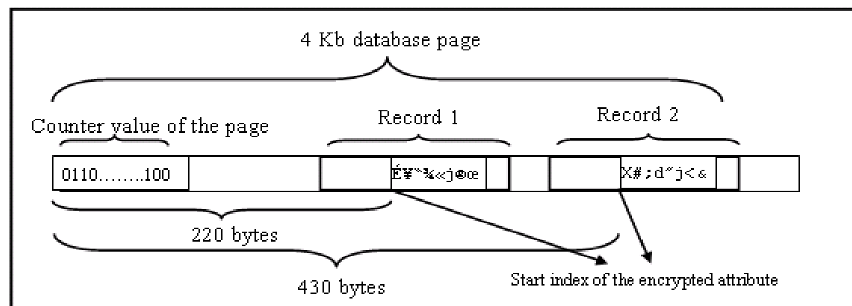

Figure 3.2. Illustration of page level CTR approach

In the next section, the performance of mini page and page level CTR methods are compared based on our experiment results. In these experiments, we observed that selective decryption aspect of page level CTR causes significant performance gain under different query types.

3.3.1 Experiments and Analyses

In order to compare the performance of Mini page and Page level CTR methods, we conducted some experiments. The first experiment is implemented to analyze the performance when all incoming queries to database are projection queries. The second experiment differs from experiment 1 since the queries are selection queries.

In both of these experiments it is assumed that the tuple lengths are 500 bytes and 70% of the tuples (350 byte/record) are encrypted. In each run, a 512 MB file is read sequentially from the disk and processed. In each read operation, 4 KB data is read from the disk since it is the default page size in many DBMSs.

The mini page method is implemented as it is discussed in the previous section. When a page is read from the disk, only the first part of the page is decrypted. Since all the sensitive attributes of the records are located at the beginning of the page, the decryption process continues until all sensitive attributes are decrypted.

On the other hand, page level CTR method is implemented by consecutively locating each record, which has sensitive and nonsensitive attributes. As it is discussed before, when a particular sensitive attribute of a record needs to be read, the counter value of that page is incremented as much as needed. Then CTR4 is used to decrypt the required sensitive attribute.

Projection Experiments:

To observe the selective decryption property of page level CTR encryption method, we wanted to analyze the performance of projection queries in this experiment. As it is shown in figure 3.3, in the page level CTR method, the time required to decrypt the encrypted projection attribute is proportional to the length of the attribute. However, it is independent in the mini page level since this method decrypts all sensitive attributes to read the projection attributes of tuples. In contrast, page level CTR, decrypts as much data as needed to be decrypted. Therefore, both of the techniques require almost the same amount of time when the projection attribute is the only sensitive attribute in the record.

As a result of this experiment, we observed that, compared to mini page level encryption, page level CTR has a significant impact on reducing the cost of projection queries.

Selection Experiments:

In projection experiments of page level CTR we were decrypting a certain amount of data for each record. However, in this experiment, in addition to selection attribute, we may need to decrypt the rest of the other sensitive attributes if the selection criterion is satisfied during query processing. This can be illustrated via an example, in which we have a table T with attributes A1, A2, and A3, where A1 and A2 are encrypted but A3 is not. In order to process a query such as "SELECT * FROM T WHERE A1 = X", the attribute A1 of each tuple should be decrypted. If the result of the condition A1 = X is true, not only A1 but also A2 should be decrypted. So, the performance of page level CTR depends on the selection condition of each tuples. Because of this reason, we repeated the experiments for different probability values of selecting a tuple.

The results of the experiments are shown in Figure 3.4, 3.5, and 3.6 where the probability of selecting a tuple is 30%, 60% and 100% respectively.

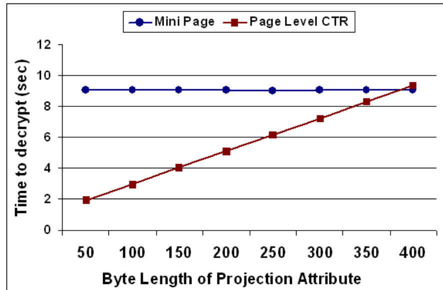

Figure 3.3. Experiment results for projection queries

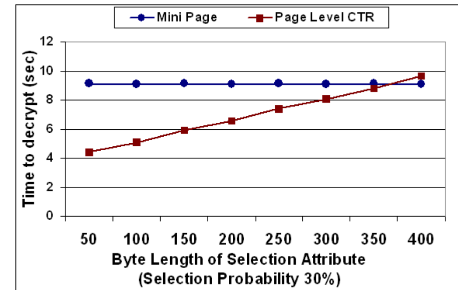

Figure 3.4. Experiment results for selection queries

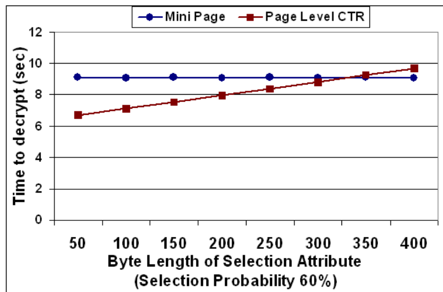

Figure 3.5. Experiment results for selection queries

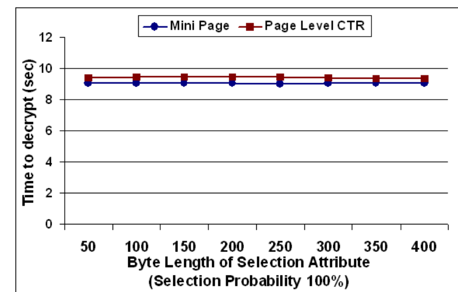

Figure 3.6. Experiment results for selection queries

An important point to note is that the time required to decrypt tuples increases gradually when the probability of selecting tuples increases. When the probability is 100%, the cost of decrypting tuples becomes independent of the byte length of selection attribute. In addition, the cost of decrypting tuples in page level CTR is slightly more than the cost associated with the mini page as it is seen in Figure 3.6. The main reason for this is the overhead of extra index calculations in page level CTR approach.

In the selection experiment, we observed that selective decryption feature of page level CTR causes significant performance gain if the selection probability is low.

3.4 Discussion

We now discuss other encrypted data related issues in database management systems.

- **Key Management:** Using careful implementation of CTR mode, we can encrypt up to 2^{128} bits of data with a single key. Therefore, for most cases, we only need one encryption key per table. These encryption keys must be either stored in tamper-proof hardware or encrypted with a master key. If the master key option is chosen, during the system start, this master key can be loaded by the database administrator. If we do not want to trust the DB administrator with the master key, we can use classic threshold schemes to store the master key. For example, using a (k, t) secret sharing scheme, we can distribute this master key to t people and any k or more of them can come together to construct the master key [83].

  For security purposes, in the CTR mode, the same counter value should not be used for encrypting two different blocks. A simple way to solve this problem is to maintain a global counter value for each encryption key in use and update the counter value after each incrementation.

- **Insertion, Deletion, and Updates:** As mentioned above, counter values in CTR mode could not be used again. At the same time, we keep one initial counter value per page and calculate the counter values needed for selectively decrypting some attributes of the tuples using this initial counter values. When we need to update a part of the page, we need to encrypt the entire page with a different initial counter value.

- **Transaction Management** When there is an insert, delete, or update operation, DBMS will write a log to the log file. Therefore, to protect the sensitive data, we also need to encrypt

the log file pages corresponding to encrypted tables. Otherwise, sensitive data values could be extracted from log files.

3.5 Conclusion

In this section, we discussed the performance of different block cipher modes, under different encryption granularity and disk access patterns. Based on our experiments and analyses, we suggested a CTR based approach for encrypting data in DBMSs. We showed its potential for processing encrypted data faster by starting decryption process even before seeing the encrypted data. In addition to that, we proposed a page level encryption method by utilizing the selective decryption feature of CTR mode. Based on the results of our experiments, we compared the performance of the encryption method that we propose with the performances of other approaches and show its advantages under different query types.

CHAPTER 4
QUERY OPTIMIZATION IN ENCRYPTED RELATIONAL DATABASES BY VERTICAL SCHEMA PARTITIONING

As we mentioned in Section 3, storing the data in encrypted format entails non-negligible performance penalties while processing queries. In this section, we first present some experiment results conducted on benchmark datasets proving that excessive decryption cost during query processing result in CPU bottleneck. As a solution we propose a new method based on schema decomposition that partitions sensitive and non-sensitive attributes of a relation into two separate relations. Throughout this section we consider a semi trusted attack model similar to the previous section. As we discussed in Section 1, in the semi trusted attack model, it is assumed that the server where the queries are executed is trusted and only the disk (storage device) is vulnerable to compromise.

4.1 Introduction

Encrypted storage of sensitive data in databases can be achieved at various levels of granularity. Tuple level, page level and column level encryption are the most well known options for this purpose.

In tuple level encryption, each tuple is encrypted or decrypted separately. If the database needs to retrieve some of the tuples, there is no need to decrypt all tuples in the table. The major drawback of this technique is that selective encryption of sensitive attributes will cause fragmentation of encrypted data within pages. Since the records are kept consecutively, encryption of the

sensitive attributes of each tuple will be fragmented. Therefore it is not an efficient technique since decryption of all data one at a time is not possible [49].

Page level encryption corresponds to a mechanism where a particular page is completely decrypted whenever it is accessed. It is the most convenient granularity option if all attributes of a table are sensitive [49]. Additionally, page level approach does not require major changes in the design of the databases. Since the page structure is not changed, it can be implemented between the buffer manager and file manager layers with a slight modification. However, page level encryption is not preferable if a table includes both sensitive and non-sensitive attributes. This is because, non-sensitive attributes are unnecessarily encrypted along with sensitive attributes.

Column level encryption can be implemented by mini page approach which is proposed by [49] based on the work of Ailamaki et al. [7]. In this technique, when a tuple is inserted into a page, its attributes are partitioned and stored in corresponding mini pages within the same page. Hence, sensitive attributes of records are kept altogether within the page. The most important aspect of this technique is that, it allows selective decryption of sensitive data. However, this approach has not been popular in conventional databases since it requires major changes in the storage engine. Additionally, accessing the sensitive and non-sensitive attributes of the records inside a page incurs an extra cost since the attributes are not stored together.

In this section, we propose a new technique to encrypt sensitive data. Our method does not suffer from any of the above disadvantages and even provides efficiency in terms of query execution time. Instead of keeping sensitive and non-sensitive attributes in the same table, we propose partitioning the table completely and storing them in two separate tables. By doing so, we can both prevent unnecessary decryption operations and reduce the number of pages retrieved

to the buffer pool. Whenever a query needs to access only non-sensitive attributes, we do not need to retrieve encrypted parts of the relations to the buffer pool. For queries that require accessing both sensitive and non-sensitive attributes, we can then retrieve partitioned tuples from encrypted and unencrypted relations and join them. Despite the cost associated with these join operations, the overall query evaluation performance could be boosted since partitioning also prevents CPU bottleneck. A detailed analysis of this issue is presented in section 4.2.

We now introduce vertical partitioning over an example schema that contains only one relation. Consider a company that stores the following information of its customers: $Customer$ (TupleID, Name, BirthDate, Address, Phone, SSN, CreditCardNumber). Let us assume that the database administrator designates SSN and CreditCardNumber fields as sensitive attributes and requests the DBMS to store these attributes encrypted. If the DBMS only permits page level encryption, one obvious solution is to encrypt the entire $Customer$ relation. The solution that we propose is vertically partitioning $Customer$ into two sub-relations containing only non-sensitive and sensitive attributes respectively. These two options are listed below:

- $Option_1$: Storing the relation without decomposition and encrypting the entire table.
- $Option_2$: Partitioning the relation into two sub-relations such that: $Customer_1$ (TupleID, Name, BirthDate, Address, Phone) $Customer_2$ (TupleID, SSN, CreditCardNumber) and encrypting relation $Customer_2$ but not $Customer_1$.

If the data file for $Customer$ relation is 7000 pages long, assuming all attributes are of equal length, $Customer_1$ and $Customer_2$ relations should fit in roughly 5000 and 3000 pages respectively. Please note that a sensitive attribute might be part of the primary key (i.e., SSN

for $Customer$), in which case, partitioning requires replacing the primary key with an additional attribute unique across the records (i.e., TupleID attribute of $Customer$).

Next, we briefly compare the two options through a workload that consists of three queries over the $Customer$ relation:

- $Query_1$: Name and Phone attributes of customers are accessed (only non-sensitive attributes).
- $Query_2$: TupleID and SSN attributes of customers are accessed (only sensitive attributes).
- $Query_3$: Name and SSN attributes of customers are accessed (both sensitive and non-sensitive attributes).

When $Option_1$ is employed, regardless of sensitivity of the accessed attributes for each query, the DBMS fetches and decrypts all 7000 pages of the $Customer$ relation. Therefore, non-sensitive attributes are decrypted unnecessarily.

If $Option_2$ is selected, $Query_1$ can be answered using only $Customer_1$ because sensitive attributes are irrelevant. Since $Custormer_1$ is not encrypted, there is no associated decryption cost. Also recall that $Customer$ consists of 7000 pages, while $Customer_1$ is assumed to fit in 5000 pages. Therefore, overall cost of evaluating $Query_1$ under $Option_2$ should even be lower than the cost over an unencrypted version of $Customer$ due to less IO latency.

Choosing $Option_2$ as the method of encryption is advantageous for $Query_2$ as well. Instead of decrypting the entire $Customer$ relation (7000 pages), we can process the query using only $Customer_2$ (3000 pages). The savings for this particular example are quite significant: around 57% (4000 pages) less IO and decryption costs with $Option_2$.

$Query_3$ involves attributes from both $Customer_1$ and $Customer_2$. Therefore evaluating $Query_3$ requires fetching all pages of both, decrypting $Customer_2$ and additionally joining $Customer_1$ and $Customer_2$ using the primary key, TupleID. For $Query_3$, IO costs of $Option_2$ are always higher than $Option_1$ because the primary key is stored redundantly (once for each vertical partition) to ensure lossless join of the partitions. Decryption costs of $Option_2$, on the other hand, will be lower since non-sensitive attributes are not encrypted with $Option_2$. Please note that join cost is specific to $Option_2$. Overall, effectiveness of vertical partitioning depends primarily on the trade-off between encrypting non-sensitive attributes and joining the partitions.

The decision of vertical partitioning depends heavily on types and frequencies of the queries in query workload. If majority of the queries do not require joining the partitions (i.e., $Query_1$ and $Query_2$), then vertical partitioning might significantly improve the performance. On the other hand, if sensitive and non-sensitive attributes of the table in question have close affinity with each other (i.e., $Query_3$), then the overall workload performance might be poor. This is because most of the queries within the given workload will require significant number of costly join operations.

4.1.1 Advantages of the Proposed Approach

Throughout the section we discuss the following problems and suggest solutions:

Preventing CPU bottleneck by vertical partitioning: Preventing encryption of non-sensitive attributes is not the only advantage of vertical partitioning. Our experimental results reveal that, separating non-sensitive attributes from sensitive ones also provide higher CPU utilization during pipelined execution of a query involving multiple relations with sensitive attributes.

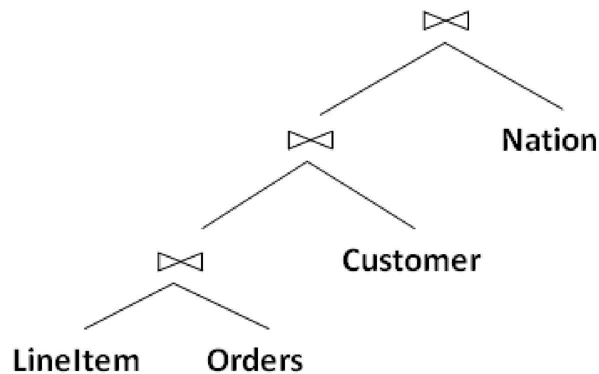

Figure 4.1. Left deep join tree of TPC-H query-10

Details of these experiments are discussed in section 4.2.

Partitioning single relations: As we have pointed in section 4.1, the decision of partitioning a relation depends primarily on how frequently the partitions will be joined. We model the decision problem of whether partitioning a relation is efficient as an optimization problem in section 4.3.

Partitioning relations of a schema: At the schema level, partitioning decision is also determined by interactions among different relations. Therefore, considering the workload over the entire schema makes more sense than solving the decision problem for each relation independently. We formally present this problem and propose a heuristic solution in section 4.4.

4.2 Preventing CPU Bottleneck by Vertical Partitioning

Most conventional database systems use pipelining for query processing. Pipelining improves query evaluation efficiency by joining the tuples of intermediate results with the tuples of the outer relations without waiting for completion of all intermediate join operations. Therefore, join operations are executed by reading pages of each table simultaneously.

To observe the impact of cryptographic operations on real database systems, we implemented a cryptography layer within Mysql-InnoDB storage engine [48] and conducted several experiments using TPC-H dataset and queries [89]. We repeated our experiments with different buffer sizes and different selectivity ratios. Our experimental results suggest that CPU becomes a bottleneck if all table accessed by the query are encrypted. This, in turn, translates to a significant increase (50-60 %) in query processing time.

If all relations are composed of sensitive attributes only, then this increase is inevitable. On the other hand, if the relations include both sensitive and non-sensitive attributes, vertical partitioning into sensitive and non-sensitive sub-relations eliminates this problem effectively. In our experiments, we observed that rather than the amount of data being decrypted, the number of encrypted tables being joined has more impact on query execution time. By separating the non-sensitive attributes from sensitive ones, the number of encrypted relations joined during query execution can be reduced. If a query accesses only non-sensitive attributes of a relation, partitioning will prevent retrieving the encrypted attributes of that relation. Hence, we get a considerable improvement for execution of a given query workload.

4.2.1 Details of the CPU Bottleneck Experiments

In the following example, we illustrate how pipelining prevents CPU from becoming a bottleneck. Figure 5.4.1 shows the join tree of TPC-H query # 10 [89]. Suppose all attributes of LineItem table are sensitive and the remaining three tables include both sensitive and non-sensitive attributes. Let q be a query which accesses only the non-sensitive attributes of these three tables and some attributes of LineItem table.

If we apply partitioning, sensitive and non-sensitive attributes of Orders, Customer and Nation tables will be stored in separate tables. During pipelined execution of query q, only the pages of LineItem table will be decrypted. That is, only one out of four pages will be decrypted within unit time due to concurrent reads.

On the other hand, if partitioning is not applied, all four tables will be decrypted as they all contain some sensitive attribute. Hence, accessing non-sensitive attributes of Orders, Customer and Nation require decryption of both sensitive and non-sensitive attributes. This time, during the pipelined execution of query q, any page read from any of these four tables will be waiting for decryption, i.e., four out of four pages will be decrypted within unit time. In our experiments, we observed that decrypting all retrieved pages overloads the CPU, which in turn increases query execution time.

As stated in the above example, partitioning reduces the number of pages decrypted within unit time. However, for some cases, keeping non-sensitive and sensitive attributes in the same table might yield a better outcome in terms of overall workload execution performance. We will discuss this issue in sections 4.3 and 4.4 in detail. In order to quantify the increase in query execution time caused by CPU bottleneck, we prepared a test environment using MySQL and TPC-H data. In this implementation, we used AES as the block cipher algorithm and employed OpenSSL library [21] for cryptographic operations. To implement the cryptographic layer we modified two components of MySQL-InnoDB source code. To encrypt the dirty data pages before writing to the disk the file manager (fil0fil.c) were modified. To decrypt the pages retrieved to the memory the buffer manager (buf0buf.c) were modified. As modes of operations, we used Counter mode to encrypt the data. For a detailed discussion on key management and transaction

management issues please refer to [15] as we followed the same procedure. All experiments were conducted on a 2.79 GHz Intel Pentium D machine with 2 GB memory on NT platform. We prepared three database instances each with 1 GB TPC-H data, and composed of 8 distinct tables. In our experiments, we used only four of these eight tables: LineItem, Orders, Customer and Nation. These four tables occupy 85 % of all the file space in each database. In the first instance we did not encrypt any of these four tables. In the second instance we encrypted only the LineItem table which occupies almost 80 % of all the file space occupied by these four tables. In the third instance all four tables were encrypted.

After preparing these instances, we ran Query-10 of TPC-H benchmarking dataset which joins these four tables and measured the execution times over the three database instances. To make sure that these results are independent of the database buffer pool size we repeated the same experiment with different buffer pool sizes. The results of the experiments are given in figure 4.2.

In figure 4.2, the queries run on the unencrypted and LineItem-only encrypted database instances take almost the same amount of execution time. On the average, execution time for the unencrypted instance is 5 % less than the latter, but the two series are barely distinguishable. Notice that although LineItem table occupies 80 % of the database file space, associated decryption cost only introduces 5 % overhead. However, if the other three tables (which occupy only the remaining 20 %) are encrypted as well, decryption cost becomes 50-60 %. We conclude that the overhead resulting from decryption is not directly proportional to the amount of data. Pages of LineItem table can be decrypted while pages of the other three tables are being retrieved from the disk. But, if partitioning is not applied, IO latency can not be parallelized with CPU-intensive decryption operations. Therefore CPU becomes the bottleneck.

Figure 4.3 presents a cross section of query execution times when the buffer pool size is fixed at 400MB. The query execution time in the LineItem-only (all-tables) encrypted database instance is 523 (853) seconds which is 5 % (71%) more than the query execution time over unencrypted database instance. Due to pipelined query execution, time spent on cryptographic operations over LineItem-only instance is almost overlapped with page read operations.

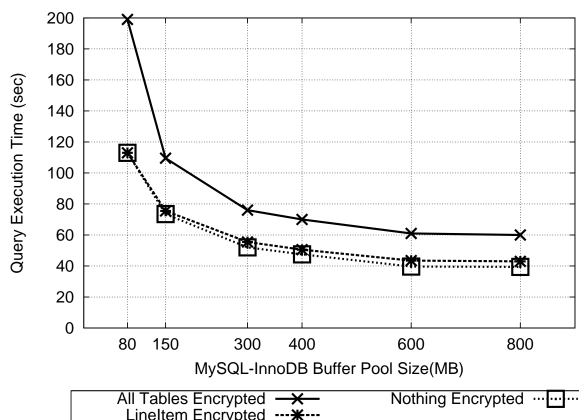

Figure 4.2. Multi table partitioning over indexed tables

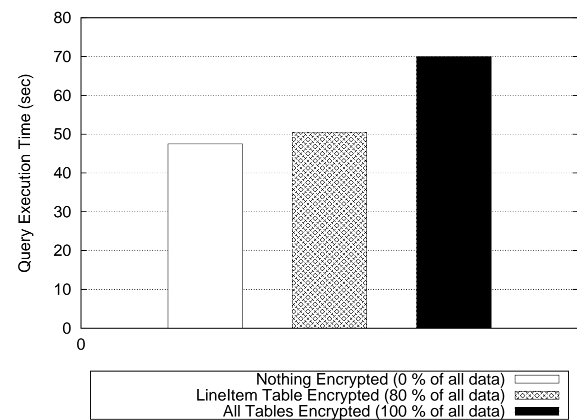

Figure 4.3. Effect of CPU bottleneck on encrypted query processing

4.2.2 Mitigating the Join Cost Due to the Partitioning

As we discussed above, vertical partitioning prevents CPU bottleneck problem effectively. However, partitioning every relation that includes sensitive and non-sensitive attributes may not always be the best solution. For those tables where the affinity between the attributes is very high, it might be preferable to keep the attributes together. To make an optimal decision, we will propose workload dependent approaches for single table and multi table queries. In section 4.3, we describe the partitioning issue in a detailed manner and analyze the workload dependent vertical partitioning approach for single table queries. Later on in section 4.4, we discuss the same notion for multi

table queries. We discuss that finding the optimal problem is not tractable and propose a heuristic to make an effective partitioning decision.

4.3 Partitioning a Single Relation

We have discussed the advantages of partitioning a relation in section 4.1. In this section, we formally define the problem and provide our experimental results.

4.3.1 Formal Definition of the Problem

Given a relation $R = \{A_1, A_2, ...A_n\}$ suppose, without loss of generality, that the attributes $A_j, A_{j+1}, ...A_n$ are sensitive attributes that should be stored encrypted whereas the remaining attributes are non-sensitive. Let $E(r)$ denote the relation r in encrypted format. We will consider two transformations for storing the tuples of R: $E(R) \leftarrow T_0(R)$ and $(R_0, E(R_1)) \leftarrow T_1(R)$ where $R_0 = \{A_1, ..., A_{j-1}\}$ and $R_1 = \{A_j, ..., A_n\}$. Here $E(R)$ represents encryption of unpartitioned relation R. R_0 and $E(R_1)$ represent partitions of relation R that contains unencrypted non-sensitive attributes (R_0) and encrypted sensitive attributes (R_1) respectively.

Suppose there is a workload $\Gamma = \{Q_1, Q_2, ..Q_\kappa\}$ defined on the relation R and w_t is the weight of query Q_t in the workload. We denote the *minimum cost* of running query Q_t over the transformed relations as $C_t(T_b(R))$. For example, $C_1(T_0(R))$ denotes the minimum cost of running Q_1 over the set of transformed relation $E(R)$ whereas $C_1(T_1(R))$ denotes the minimum cost of running Q_1 over the set of transformed relations R_0, $E(R_1)$ (note that $T_0(R) = E(R)$ and $T_1(R) = (R_0, E(R_1))$).

Let TC^{UnPart} be the overall query evaluation cost for a given workload while the relation R is unpartitioned and TC^{Part} be the overall query evaluation cost while the relation R is

partitioned. Using the above notation, we can define TC^{UnPart} and TC^{Part} as follows:

$$TC^{UnPart} = \sum_{t=1}^{\kappa} w_t.C_t(T_0(R))) \qquad TC^{Part} = \sum_{t=1}^{\kappa} w_t.C_t(T_1(R)))$$

For a given workload if $TC^{UnPart} < TC^{Part}$, then partitioning does not improve the overall performance of a given workload because the cost of join operations suppresses the savings from cryptographic operations. On the other hand, $TC^{UnPart} \geq TC^{Part}$ implies that partitioning is advantageous, since it reduces the overall execution time.

4.3.2 Experiment Results

To observe the effectiveness of partitioning on single table queries we conducted experiments using TPC-H dataset. We observed that if majority of the queries in a given query workload does not require joining sensitive and non-sensitive attributes, then decomposition will improve the performance significantly.

In this experiment, we generated two instances of 1 GB database using TPC-H dataset and used LineItem table of these instances since LineItem is the largest table of the TPC-H dataset. It occupies almost 80% of the whole database. In the first instance, we encrypted all attributes of LineItem table and stored the table without partitioning. In the second instance, LineItem table is partitioned such that half of the attributes are stored in LineItem_1 in plaintext and half of the attributes are stored in LineItem_2 in encrypted format. In terms of storage, LineItem_1 occupies 467 MB whereas LineItem_2 occupies 532 MB disk space.

To build a workload, we prepared three types of queries using TPC-H query # 6 as the basis. Query type 1 accesses only non-sensitive attributes of LineItem table whereas Query type

2 accesses only sensitive attributes. Query 3 accesses both sensitive and non-sensitive attributes of the LineItem table.

Query 1 and 2 are very similar to query 3. The only difference is that they do not require joining sensitive and non-sensitive attributes.

After running these three queries in both instances of the databases, we represented the results in various query workload scenarios. In these workloads there are 100 queries. Figure 4.4 represents a query workload where 20 queries require accessing both sensitive and non-sensitive attributes (Type 3 queries). The remaining 80 queries include both query 1 and 2. Therefore 20 % of the queries require a join operation while 80 % does not. As it is seen in figure 4.4, if the number of type 3 queries is low, then partitioning the tables significantly reduces the overall query execution time.

In figure 4.5, we can see that if the number of type 3 queries increases then partitioning becomes less effective. As it is shown , if the number of type 3 queries is less than 40 %, partitioning will still be effective. Otherwise, keeping relations unpartitioned is a better choice.

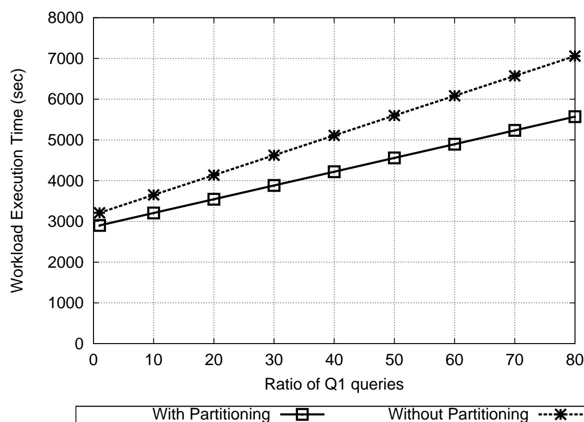

Figure 4.4. Various distributions of type 1 and type 2 queries.

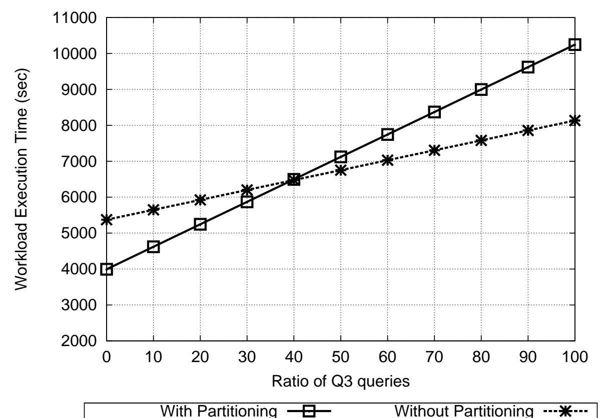

Figure 4.5. Various distributions of type 3 queries.

4.4 Partitioning Multiple Table

The partitioning decision in single table queries is rather simple: Given a relation R and a workload κ, should we partition the relation or not. However the same decision for multiple tables additionally requires considering the interaction among different tables. Therefore given n relations we need to evaluate 2^n different combinations of partitioning decisions per table. Theoretically, if the overall query execution time for each of these combinations were available, choosing the best decision would be simple: select the one that requires the least amount of time. However, this strategy is not practical in real-world applications since this solution is not tractable. In the following subsection we provide a formal representation of the problem and show that it is an example of binary integer programming.

4.4.1 Formal Definition of the Problem

Given the ith relation, $R^i = \{A^i_1, A^i_2, ...A^i_n\}$ suppose, without loss of generality, that the attributes $A^i_j, A^i_{j+1}, ..., A^i_n$ are sensitive attributes that should be stored encrypted whereas the remaining attributes are non-sensitive.Let $E(r)$ denote the relation r in encrypted format. We will consider two transformations for storing the tuples of R^i: $E(R^i) \leftarrow T_0(R^i)$ and $(R^i_0, E(R^i_1)) \leftarrow T_1(R^i)$ where $R^i_0 = \{A^i_1, ..., A^i_{j-1}\}$ and $R^i_1 = \{A^i_j, ..., A^i_n\}$. Here $E(R^i)$ represents encryption of unpartitioned relation R^i. R^i_0 and $E(R^i_1)$ represent partitions of relation R^i that contains unencrypted non-sensitive attributes (R^i_0) and encrypted sensitive attributes (R^i_1) respectively.

Suppose there is a workload $\Gamma = \{Q_1, Q_2, ..Q_\kappa\}$ defined on the relations $R^1, \ldots, R^v$ and w_t is the weight of query Q_t in the workload. We denote the *minimum cost* of running query Q_t over the transformed relations as $C_t(T_{b_1}(R^1), T_{b_2}(R^2)\ , \ldots, T_{b_v}(R^v))$. For example, $C_1(T_0(R^1), T_1(R^2))$ denotes the minimum cost of running Q_1 over the set of transformed relations

$E(R^1)$, R_0^2, $E(R_1^2)$ (note that $T_0(R^1) = E(R^1)$ and $T_1(R^2) = (R_0^2, E(R_1^2))$).

Using the above notation, we can define the optimum partitioning strategy as a minimization problem as follows:

$$\min_{b_1, b_2, \ldots, b_v} \left[\sum_{t=1}^{\kappa} w_t . C_t(T_{b_1}(R^1), T_{b_2}(R^2), \ldots, T_{b_v}(R^v)) \right]$$
$$\text{subject to } b_j \in \{0, 1\}, 1 \leq j \leq v$$

The above optimization is an example of binary integer programming problem which is known to be NP-Hard [30]. In the next section, we discuss a simple heuristic approach.

4.4.2 One Step at a Time (OSAT) Heuristic

Instead of evaluating all different combinations of partitioning decisions, we propose a greedy heuristic called "one step at a time" (OSAT). According to this heuristic, relations are evaluated one by one in a particular order and once the decision of partitioning a specific relation is made, this decision is considered when other relations are being evaluated.

Assume that we have a set of relations $S = \{R_1, R_2, ..., R_n\}$. These relations have both sensitive and non-sensitive attributes. Given a workload κ, we can decide to partition each relation one by one. First, we evaluate relation R_1. While we are evaluating this relation we need to assume that all remaining relations are not partitioned. Then we need to estimate the total execution time for the given workload for both partitioned and unpartitioned versions of this relation. Depending on the result, say, we decide to partition R_1. Once we decide partitioning R_1, that decision will be used for subsequent evaluations. Therefore, when we evaluate R_2 we assume that R_1 is already partitioned and the remaining relations are still unpartitioned. This process will continue until the

decision of partitioning R_n is made. The order of evaluation is determined by the descending sizes of tables. Depending on query workload, we sort the tables with respect to their sizes and then start evaluating each table in this particular order.

4.4.3 Experiment Results

To show that OSAT is an effective heuristic, we conducted experiments using TPC-H dataset. We observed that the heuristic finds almost the same partitioning strategy as the optimal solution. In addition to that, we observed that constructing the database schema with the optimum partitioning strategy boosts the overall query workload performance tremendously.

In this experiment, we assumed that four tables of the database include some sensitive and non-sensitive attributes. These tables are: Supplier, Customer, LineItem, and Orders. For each of these tables half of the attributes are assumed to be sensitive so that sub-relations are balanced in terms of size. We constructed the workload with TPC-H Queries 5, 7, and 10 since they access these four tables. As to workload, we prepared six plans with different distributions of these three queries.

To be able run the queries, we prepared a database instance such that each of the four tables are inserted to the database as partitioned and unpartitioned. Among the partitioned tables, only the partitions that contain the sensitive attributes are encrypted. In unpartitioned ones, all attributes of these four tables are stored encrypted. We implemented a "Query Rewriter" to generate queries with respect to different partitioning combinations. Since we used four tables, there are $2^4 = 16$ different partitioning scenarios for a given query. Since we have 3 different queries, there are $3 \times 16 = 48$ different queries that we need to run. We run those 48 queries and measured their execution times. The results are given in Table 4.1 in the columns "Query 5", "Query 7" and

"Query 10". Here columns S, L, O and C correspond to Supplier, LineItem, Orders and Customer tables respectively. An entry "P" on a column denotes that the corresponding table is partitioned.

Table 4.1. Query execution times for TPC-H queries 5-7-10 and the workload execution times on different partitioning scenarios

S	L	O	C	Query 5	Query 7	Query 10	W-1	W-2	W-3	W-4	W-5	W-6
-	-	-	-	531	671	543	57900	56500	60460	60340	57540	56260
P	-	-	-	532	670.5	543	57905	56520	60455	60345	57575	56300
-	P	-	-	48	36	34	3740	3860	3780	3920	4160	4140
-	-	P	-	419	653.5	508.5	53410	51065	56310	55415	50725	49275
-	-	-	P	530	670	552	58300	56900	60660	60440	57640	56460
P	P	-	-	57	36.5	34	3935	4140	3985	4215	4625	4600
P	-	P	-	420.5	651.5	508.5	53380	51070	56240	55360	50740	49310
P	-	-	P	533	668.5	552	58315	56960	60645	60455	57745	56580
-	P	P	-	50.5	30	34	3610	3815	3530	3695	4105	4145
-	P	-	P	39	26	32	3160	**3290**	3040	3110	3370	**3430**
-	-	P	P	418	653	517.5	53825	51475	56535	55540	50840	49485
P	P	P	-	60	30	34	3800	4100	3720	3980	4580	4620
P	P	-	P	48	25	32	3310	3540	3170	3330	3790	3860
P	-	P	P	418	651	517.5	53765	51435	56435	55440	50780	49445
-	P	P	P	42	20	33	**3090**	3310	**2830**	**2920**	**3360**	3490
P	P	P	P	51.5	20	33	3280	3595	3020	3205	3835	3965

Using these execution times we calculated the overall workload execution times for 6 different distribution scenarios. The results are shown in Tables 4.1. We denote the ith workload plans as *W-i*.

For four distribution scenarios, exhaustive search technique suggested partitioning Customer, LineItem, and Orders tables but not Supplier table. For the remaining two scenarios only LineItem and Customer tables are suggested to be partitioned. On the other hand, for all six scenarios, OSAT suggested to partition Customer, LineItem, and Orders. Therefore in 4 out of 6 scenarios, both approaches found the same partitioning strategy. However when we analyze the total running times of that 2 differentiating scenarios, total execution times are very close. If the database is created according to the partitioning result of the heuristic, it takes 18995 seconds to run all given workloads. In contrast, it takes 18920 seconds with the exhaustive search technique. Assuming that exhaustive search approach gives the optimal solution, the relative error of the heuristic is 4.22/1000. Therefore we can empirically conclude that OSAT heuristic can be

used instead of exhaustive search technique since its running time is linear and it finds an almost optimal partitioning strategy.

The second important observation is that, the overall workload execution time for the optimal partitioning are significantly less than those for unpartitioned schemas. Therefore, effective schema partitioning improves the workload performance considerably. The average running time of the given workload scenarios is 58166.7 seconds if none of the tables are partitioned. On the other hand, it takes 3165.8 seconds if the tables are partitioned. Therefore, using OSAT, partitioning the tables reduces the overall running time by 94.56 percent.

The rationale behind this improvement is two-fold. First, different partitioning strategies cause the database server to choose different query execution plans and join orders. Some of these choices yield improvements while the others increase the execution cost. Especially in encrypted databases, joining the tables in the correct order has a great impact on query execution performance. Hence, trying different partitioning combinations helps find the best query execution plan. Second, by vertical partitioning the amount of data that is being decrypted is reduced. Separating the non-sensitive attributes from sensitive attributes leads to considerable improvements of performance.

4.5 Conclusion

In this section, we proposed the vertical partitioning approach to prevent unnecessary cryptographic operations over non-sensitive attributes. Experiments conducted on the benchmark TPC-H dataset reveals that another advantage of vertical partitioning is preventing the CPU from becoming the bottleneck during query execution. Our analysis indicates that, due to pipelined execution, the overhead resulting from cryptographic operations is not directly proportional to the

amount of decrypted data but rather the number of encrypted relations involved in a query. While vertical partitioning introduces only 5 % overhead, evaluating the same query over the unpartitioned database instance takes 50-60 % longer.

In our method, the decision of vertically partitioning a relation depends on how frequently the partitions are joined to answer a query. When extended to the entire schema, making these decisions for each relation require also considering the interactions among the relations. While exhaustive search implies an exponential search space, our proposed heuristic solution has linear complexity and achieves 0.4 % error rate in comparison to the optimum partitioning strategy. Overall, heuristic partitioning improves query execution time by 94.5 % on the average.

CHAPTER 5
BUILDING DISCLOSURE RISK AWARE QUERY OPTIMIZERS FOR RELATIONAL DATABASES

In the previous sections, we studied various performance aspects of processing encrypted data for semi trusted attack models. In this model, the memory is assumed to be trusted. However, recent studies show that a significant part of data theft incidents are performed by using specialized malwares that can access the main memory of systems. These malwares can easily capture the sensitive information that are decrypted in the memory including the cryptographic keys used to decrypt them. In this section, we consider an attack model where the disk is always untrusted and the memory is vulnerable to attacks for a certain amount of time. Under this threat model, we propose disclosure models to estimate and quantify the disclosure risk of encrypted data in relational databases. We also propose modifications to the standard query processing mechanism to minimize such risks.

5.1 Introduction

Most organizations today store increasing amounts of sensitive data in computer based systems. At the same time there are increasing concerns related to the security and privacy of the stored data due to many data theft incidents reported in the past. According to a recent report [88], approximately 285 Million records have been stolen from databases in 2008. It is believed that approximately 4.3 million of those records contain personally identifiable information [88]. Ac-

cording to another study, after a data theft incident, companies need to spend between $90 and $305 per lost record for various forensic, legal and IT costs [32].

To reduce the effect of various attacks and limit disclosure risks of sensitive data, organizations prefer to store the confidential data in encrypted format in databases. To support such demand, most of the commercial DBMS products in the market nowadays have built-in encryption support. One of these brands, Microsoft SQL Server 2008 [1], provides Transparent Data Encryption (TDE). TDE provides protection for the entire database at rest without affecting existing applications. In this security mechanism, a cryptographic key hierarchy is used to guarantee the secrecy of the keys. All data and index pages written to the storage device are encrypted with database master key (DMK). All DMKs under a certain service running in the system are encrypted with the corresponding Service Master Key (SMK). At the root of this key tree, service master keys are encrypted with the OS level root key. In this model, since the encryption and decryption operations are performed in memory, the cryptographic keys needed for such operations must be kept in memory as well. Unless a special hardware is used, all of these keys are kept in the main memory as long as they are in use. The implicit threat model assumed by these products is that the database server is trusted and only the disk is vulnerable to compromise. In the event that the physical storage devices are stolen this method prevents data theft effectively. On the other hand, this model is not strong enough to prevent data disclosure if the attacker has access to the main memory of the server itself. According to the Verizon's Data Breach Investigation Report [88], newer varieties of malware utilities bypass existing access controls and encryption, effectively creating vulnerable data stores that can later be retrieved from the victim environment.

[1] Similar functionality is provided by Oracle.

Examples of this include the usage of memory scrapers, sophisticated packet capture utilities, and malware that can identify and collect specific data sequences within memory, unallocated disk space and pagefile. According to this report, 85 % of the 285 million records breached in the year 2008 were harvested by custom-created malware. Clearly, a malware observing memory can easily capture the master keys used to encrypt/decrypt the sensitive information stored in the disks[2]. Once the keys are revealed, all sensitive data may be compromised irrespective of what encryption algorithm has been used.[3]

To prevent this type of attacks, SQL Server provides the Extensible Key Management module (EKM). It enables parts of the cryptographic key hierarchy to be managed by an external source such as Hardware Security Module (HSM), referred to as a cryptographic provider[4]. This external hardware is used to keep the master keys secret. Because of the low processing capacity, HSM is not used to encrypt and decrypt the bulk data. Instead, it is used to encrypt and decrypt the SMKs as needed. The actual data is decrypted by the processors of the server machine. Nonetheless, the naive usage of HSM does not sufficiently protect against main memory attacks. If a large portion of the sensitive data is brought into the main memory during query processing, the disclosure risk increases significantly. Current query optimizers aims to minimize the execution cost of the queries without considering the sensitivity of the table contents. Instead, a data-sensitivity aware query optimizer might choose a plan which minimizes the accesses to the HSM. (An illustrative example is provided in Section 5.5.1). Unfortunately, none of the existing products take this aspect into consideration.

---

[2]Memory scraping is listed among the top 15 data breach threats in [88].

[3]The proposed techniques in this section are also effective against the attack scenarios described in [40] where they describe how to capture cryptographic keys from the memories with cold boot attacks.

[4]We use the term Secure Co-rocessor (SCP) and HSM interchangeably

Another issue is that the granularity of the leaf level keys in a given key hierarchy is not small enough to restrict the disclosure area. For example, if there are multiple SQL Server database instances running in parallel, during a memory attack, all DMKs of these databases will be compromised no matter what particular subset of these databases were in use at that moment.

In this section, to create a better last line of defense and to address some of the issues with current encrypted data storage, we create the following novel solutions:

- We propose a novel query optimization approach that is cognizant of the disclosure risk of sensitive data. We derive appropriate disclosure-risk metrics for all kinds of data access mechanism considered by a typical DBMS optimizer.
- We incorporate our search algorithm by modifying a popular (publicly available) query optimizer to generate plans that minimize main memory disclosure risks while keeping performance overheads within specified bounds. We also illustrate how to integrate the relatively slow cryptographic hardware without incurring substantial overhead.
- We carry out empirical tests to understand the nature of tradeoff between performance and disclosure-risk. We compare the performance of 3 different search mechanisms for determining the most desirable tradeoff points.

Next, we describe our threat model and give an overview of the proposed architecture.

5.1.1 Threat Model and System Overview

Threat model: In this section, we address a threat model where the disk is always assumed to be untrusted but the memory is considered to be untrusted for a certain amount of attack time. Also,

the client and HSM are always considered as trusted. Trusting a hardware component means that there is negligible probability that any malware or malicious agent will have access to (be able to tamper) the data residing on that component. Furthermore, we assume a passive adversary (e.g., malware) has access to the contents of the memory and the entire hard disk from time t_1 to time t_2. In this threat model, we try to limit what the passive adversary can learn by observing the contents of the memory by smartly designing the query optimizer.

Disclosure metric: There are many ways one can model memory scraping attacks in RDBMS. We chose one where the duration of attack is assumed to be longer than the query execution time. Therefore, an adversary may gather either decrypted data or keys off the memory brought in during a particular query execution within the interval of attack. The metrics proposed in this section estimate the "worst case" number of records that may be disclosed as a result of such an attack. They account for all the records in the extents corresponding to any data item retrieved during execution of the query. Modeling attackers in a more sophisticated manner, for example taking bandwidth limits, smaller duration (but perhaps repeated) attacks, effect of buffering and data lifetime, presence of concurrent queries etc. are all interesting directions for future work to address the memory scraping threats in RDMS more comprehensively.

In the proposed system, the decryption operation is performed in the server with the symmetric keys generated by the HSM. Based on the threat model described above, we define the disclosure risk as follows. Let S_{key} be the set of symmetric keys residing in the memory during the attack $[t_1, t_2]$. We assume that all sensitive attributes of the records (i.e., cells) encrypted with any key in S_{key} are compromised. Based on this assumption we propose disclosure models in Section 5.2 for different query evaluation techniques. In these models we use "**the number of**

cells" as the disclosure metric[5]. To clarify the definition of disclosure cost, consider the following example. Let a table T have two sensitive and three non-sensitive attributes and say, each data page of T can accommodate 100 tuples of T. Suppose further that each data page of T is encrypted with a unique key. That is, whenever a data page of T is retrieved to the memory, the HSM generates a unique decryption key for this page and the sensitive records in this page are decrypted with this key in the main memory of the server. If a query plan p requires accessing 1000 data pages of T, then the disclosure risk of p is computed as: 1000 * 100 * 2 = 200,000. Using this metric, we compare the disclosure cost of different query plans.

Proposed Architecture:

The proposed architecture is summarized in Figure 5.1. When a query is issued to the DBMS (item 1 in the figure), after parsing the query, the optimizer starts analyzing alternative query execution plans. The query optimizer solves a multi-objective optimization problem that considers the sensitive record disclosure risk and the query execution time. In this section, we focus on minimizing disclosure risk for each query independently, and do not consider the effect of running multiple queries concurrently. Such independent query evaluation could be considered as the worst case scenario for data disclosure risk. We leave the concurrent query optimization for minimizing sensitive data disclosure as a direction for future work.

During the execution, the pages are requested from the storage engine (item 2). If the page does not exist in the buffer pool, the request is forwarded to the file manager for retrieval (item 3). After the encrypted page is retrieved from the disk (item 4), the file manager requests the decryption key from the HSM (item 5). This request includes the extent ID [6] of the page.

[5] *Cell* is described as the smallest indivisible values of the tuples.
[6] An extent is defined as a set of contiguous blocks allocated in a database tablespace.

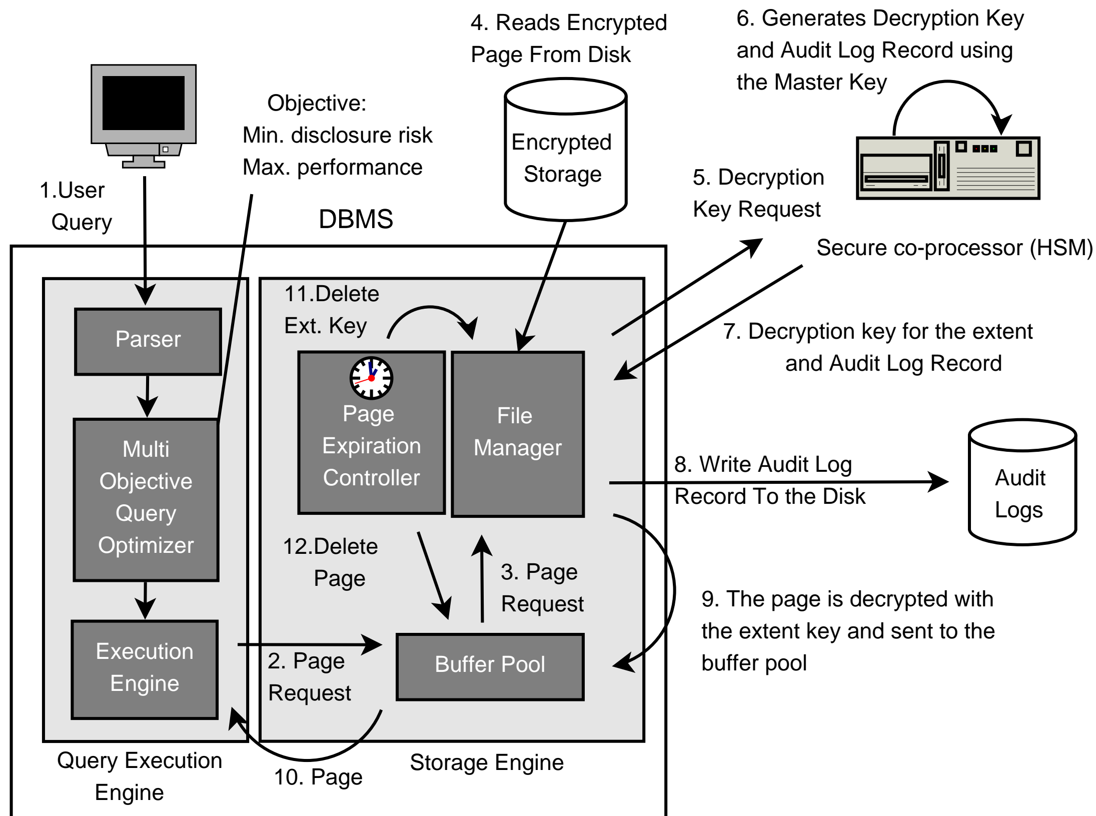

Figure 5.1. System overview

In the proposed system, all of the pages within an extent are encrypted with the same key. One could prefer key management at the level of pages or records but there are negative performance implications of this decision. We discuss the details of why we prefer extent level key management in Section 5.5.2.

Once the key request is received (item 6), HSM computes the hash of this ID and encrypts the output of this hash function with the master key stored within the HSM. Therefore a MAC based approach is used to generate the extent keys. Once the key is generated, it is returned to the file manager along with the audit log record (item 7). Next, the file manager writes the audit log record to the disk (item 8). The audit logs are used to keep track of extent accesses

during the query execution. Since HSM needs to be used to access any encrypted data page, such logs will provide an accurate estimate of what could be leaked during an attack. Next, the file manager decrypts the page using the extent key and forwards the decrypted page to the buffer pool for processing (item 9, 10). Note that the decryption of the data page is performed in the system memory, not in the HSM. If the attacker monitors the content of the memory during these operations, all records within the extent would be compromised since the extent key resides in the memory during the attack.

To guarantee that the remnants of the sensitive information is removed from the memory, the proposed system has a mechanism called *Page Expiration Controller*. The objective of this component is to delete both the extent keys and the sensitive pages (item 11, 12) from the memory after a certain amount of time, t_{life}. In [20], Chow et al. present a technique for reducing the lifetime of sensitive data (i.e., encryption keys, sensitive records) in memory called secure deallocation so as to minimize the data disclosure from the memories. The basic idea is to "zero" the data either at deallocation or within a short, predictable period afterward in general system allocators. They show that this method substantially reduces the data lifetime with minimal implementation effort in the existing operating systems. We use a similar technique to guarantee that given an attack interval $[t_1, t_2]$ there are no remnants of the data pages or the keys retrieved to the memory before $t_1 - t_{life}$. This is essential for the accuracy of the auditing process. If a data page has to stay longer than t_{life} in the memory, an audit log record including this extension event is written to the *Audit Logs* disk.

5.2 Disclosure metrics for memory-based attacks

Current database products are designed to minimize the response times of the queries issued by the clients. Estimating the query execution cost is one of the major steps in the query optimization process. Modern databases consider numerous factors while estimating the cost of alternative plans. Some of these factors are number of page I/Os and blocked I/Os, physical access speed of the storage devices, CPU costs etc. [77]. Using these parameters, the goal of the traditional optimizer is to generate a query plan that minimizes the response time of the issued queries. Typically, query optimization consists of two phases - a heuristics based query rewriting phase that generates a logical plan and a cost-based optimization phase that generates a physical plan starting from the output of the first phase. In the first phase, the optimizer employs standard heuristics to rewrite the parsed query tree where all selection and projection operators (predicates) are pushed down the appropriate branches of the tree as deep as possible. The second phase is a cost-based optimization step where predicate evaluation ordering and join ordering (for multi-relation queries) are selected. However, the sensitivity of contents of the tables are not taken into account while evaluating alternative plans. This, in turn, could lead to a dramatic change in the data exposure risk depending on the selected plan. Our goal in this study is to design an optimizer that not only maximizes the performance but also minimizes the disclosure risk. Below we first discuss this problem by analyzing a query evaluation scenario and then present the measures to estimate the disclosure risk of alternative query execution plans.

In our model since the pages are decrypted as soon as they are brought into the memory, the disclosure risk is only proportional to the number of different extent keys that are accessed. This, in turn is assumed to be proportional to the number of pages brought into the memory under

the *random spread* assumption (i.e., the fact that each matching tuple can reside anywhere on the disk independent of other tuples in the set). Therefore, the disclosure risk is only dependent upon the selectivity of the least selective predicate in the query. In an alternate model where one has a choice to postpone the decryption of a page until processing the sensitive records, the optimal predicate ordering might differ significantly. A detailed discussion on this issue is provided in Section 5.5.3.

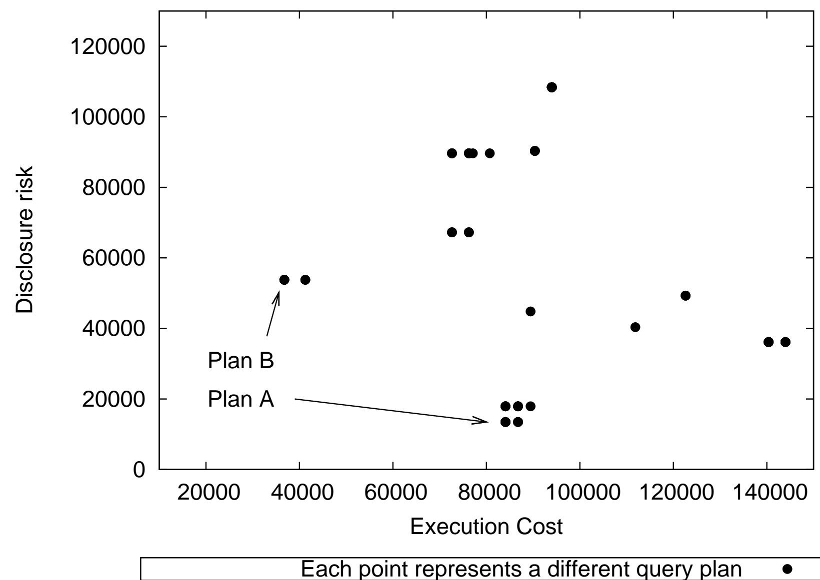

Figure 5.2. Possible execution plans for a given query

Disclosure risk vs. performance cost: Before deriving the disclosure metrics to be used by an optimizer, here we illustrate via an example how the disclosure cost and performance vary across different query plans for a sample query. Using the metrics to be described soon (in Section 5.2.1 and Section 5.2.2) we implemented a disclosure estimator module in MySQL query optimizer. The objective of this module is to estimate the disclosure risk of alternative query execution plans. Using the TPC-H schema [89], we created a database instance and designated four attributes of *Customer* table and eight attributes of *Lineitem* table as sensitive. Next, a query

joining six tables is issued to the database (See Table 5.1 for the query). Consequently, 720 (6!) different enumeration plans are generated. Each point in Figure 5.2 represents an alternative execution plan (Only the top 20 query plans are plotted in this figure). The x axis represents the query execution cost and the y axis represents the disclosure risk for the execution plans. In this figure, plan B minimizes the execution cost whereas plan A minimizes the disclosure risk. Hence, there is no single query execution plan for which both the execution time and the disclosure risk are minimized. This is a common problem in multi-objective optimization which involves the application of special methods in order to obtain a viable solution.

Table 5.1. Sample query

SELECT n_name, sum(l_extendedprice * (1 - l_discount)) as revenue FROM customer, orders, lineitem, supplier, nation, region WHERE c_custkey = o_custkey AND l_orderkey = o_orderkey AND l_suppkey = s_suppkey AND c_nationkey = s_nationkey AND s_nationkey = n_nationkey AND n_regionkey = r_regionkey AND r_name = 'AMERICA' AND o_orderdate ≥ '1993-01-01' and o_orderdate < '2000-01-01' AND O_ORDERKEY BETWEEN 261658 and 271658 GROUP BY n_name ORDER BY revenue DESC

In Section 5.3, we discuss alternative approaches to solve this multi-objective optimization problem. We now present metrics to estimate the disclosure risk given different input parameters and data access paths for a query. We also refer to the disclosure risk metrics as the *cost functions*

5.2.1 Disclosure Risk for Single-relation Queries

Single-relation queries include only one relation in the *FROM* clause. During the evaluation of these type of queries the disclosure cost changes dramatically depending on the existence of indexes. Therefore we will analyse the disclosure issue in single relations under two categories: “Plans without indexes” and “Plans utilizing at least one index” similar to the chronology of discussions in [77]. We use the notations given in Table 5.2 while describing the models.

Table 5.2. Notations used for disclosure model

$D(R)$	Total disclosure cost of accessing the relation R
$N_{ext}(R)$	Number of extents in relation R
$E_{rec}(R)$	Number of records in an extent
$C_{total}(R)$	Number of sensitive and nonsensitive columns in relation R
$C_{sens}(R)$	Number of sensitive columns in relation R
$S_{range}(R)$	Number of records satisfying a range condition defined on R
$I^{i}{}_{nleaf}(R)$	Number of sensitive values in a non-leaf index page of the ith index defined on R
$I^{i}{}_{leaf}(R)$	Number of sensitive values in a leaf index page of the ith index defined on R
$E_{iPage}(R)$	Number of index pages within an extent for the ith index defined on R
$E_{exp}(k)$	Expected number of extents touched while accessing k records through an unclustered index

In the cost models, we assume that all sensitive attributes of the relations have the same disclosure risk. However, different attributes of a table could have different levels of disclosure risk. For instance, an attribute containing the social security numbers of the clients may be considered more sensitive than their phone-number attribute. In this case, the data owner may prefer assigning weights to the sensitive attributes proportional to their sensitivity. The models described below can be easily extended to accommodate such variations.

Granularity of encryption: We provide the disclosure cost models for only column level encryption. We assume that a column encryption scheme similar to one proposed in [49] can be implemented wherein each page that is brought into the memory is decrypted separately. [7] For the databases where the table level encryption is used, the cost estimations can be carried out by assuming that all columns contain sensitive information. (See [18] for a detailed discussion on table level encryption and column level encryption).

Plans without indexes

The simplest way of accessing the records of a relation is to scan the data pages in a linear order if there is no index created on it. During a table scan all data pages are retrieved to the memory.

[7] Here we assume that the cost of page level encryption and decryption are hidden in the I/O latency.

Consequently, all sensitive information stored in relation R will be disclosed. The disclosure cost in this case would be:

$$D(R) = N_{ext}(R) * E_{rec}(R) * C_{sens}(R). \quad (5.1)$$

Plans utilizing an index

In case that the selectivity of predicates is very low, using indexes may improve the performance substantially. Additionally, the disclosure cost will be much less than a table scan if indexes are used. In the following models we assume that B+ tree indexes are used.

Single-Index Access Path: If there is an index on a relation R which matches a range condition in the *WHERE* clause of the query, first, the initial leaf page of the index tree including the data entries (or records if it is a primary index) should be identified. During this operation some non-leaf index pages should be retrieved to the memory. Typically 2-4 I/O's are performed to find this page [77]. These non-leaf index pages may include sensitive values. The disclosure cost of these top down tree traversal is $E_{iPage}(R) * {I^i}_{nleaf}(R)$ per I/O operation.

The disclosure cost of traversing the leaf pages depends on whether the index is clustered. If it is clustered, then the records satisfying a range condition could be found in the consecutive data pages[8]. Since the consecutive pages are stored in contiguous extents, we can compute the disclosure risk by estimating the number of extent keys that need to be retrieved from the HSM to the memory. In the worst case, $\left(\left\lceil \frac{S_{range}(R)}{E_{rec}(R)} \right\rceil + 1\right)$ keys will be requested from the HSM, including

[8]Here we assume that the data entries are the actual data records since this is a primary index.

the first and last extents which includes some extra records out of the range condition. Then, the worst case total disclosure cost of a clustered index access will be:

$$\left(\left\lceil \frac{S_{range}(R)}{E_{rec}(R)} \right\rceil + 1\right) * E_{rec}(R) * C_{sens}(R) + 4 * E_{iPage}(R) * {I^i}_{nleaf}(R) \tag{5.2}$$

If the index is not clustered (i.e., a secondary index), the traversal of data entries will reveal some information if the key values in the data entries are sensitive. Then, the disclosure cost of the traversal will be ${I^i}_{leaf}(R)$ for each consecutive accessed leaf page. In the worst case the I/O cost would be one I/O per matching key value under the random spread assumption. Therefore, the worst case disclosure cost is estimated as $E_{rec}(R) * C_{sens}(R)$, that is, one extent per matching tuple. For average cost estimation we use a model similar to the one in [78]. Suppose that $S_{range}(R)$ records satisfy a given range condition. Then, the disclosure cost will be $E_{exp}(S_{range}(R)) * E_{rec}(R) * C_{sens}(R)$ where $E_{exp}(S_{range}(R))$ can be estimated by using the Cardenas formula [94]. If there are $|R|$ records in $N_{ext}(R)$ extents in relation R, then the number of extents touched while accessing $S_{range}(R)$ records through an unclustered index will be:

$$E_{exp}(S_{range}(R)) = N_{ext}(R) * \left(1 - \left(1 - \frac{1}{N_{ext}(R)}\right)^{S_{range}(R)}\right) \tag{5.3}$$

Then, the worst case total disclosure cost for non-clustered index is:

$$E_{exp}(S_{range}(R)) * E_{rec}(R) * C_{sens}(R) + 4 * E_{iPage}(R) * {I^i}_{nleaf}(R) \tag{5.4}$$

Note that, expression 5.3 is independent of $|R|$ (the total number of records in R). Yao mentions in [94] that the error involved in using this approximation is negligible for cases in which

the number of tuples per page is not a small number (say < 10). This assumption is typically true for page sizes used in today's database systems (4KB-32KB) [78].

Multiple-Index Access Path: If there are multiple indexes (i.e., secondary indexes) matching the predicates in the *WHERE* clause of the query, these indexes can be used together to minimize the number of data pages retrieved to the memory. The disclosure model of this evaluation method is similar to the single index case except the disclosure cost of each non-leaf traversal operation for each index should be calculated independently.

Index-Only Access Path: If all of the attributes mentioned in the query (in the *SELECT*, *WHERE*, *GROUP BY*, or *HAVING* clauses) are included in the search key for some *dense* index on the relation in the *FROM* clause, an index-only scan can be used to compute answers [77]. The disclosure cost of using an index-only plan consists of the disclosure due to the traversal of non-leaf and leaf nodes of the index tree. Similar to the single index case, the disclosure cost of top down tree traversal cost will be $E_{iPage}(R) * {I^i}_{nleaf}(R)$ for each I/O operation. As for the leaf pages, if the index is a primary index then the disclosure cost will be $\left(\left\lceil \frac{S_{range}(R)}{E_{rec}(R)} \right\rceil + 1\right) * E_{rec}(R) * C_{sens}(R)$. If it is a secondary index, then the disclosure cost will be ${I^i}_{leaf}(R)$ for each consecutive accessed leaf page.

We do not derive expressions for the multiple index cases here. In general, different ways of traversing the indexes will lead to differing degrees of disclosure.

5.2.2 Disclosure Risk for Multi-relation Queries

Join Algorithms: The most commonly used join algorithms in the conventional database systems are *nested loop join*, *block nested loop join*, *index nested loop join*, *sort merge join*, and *hash join*. Except *index nested loop join*, all of these algorithms require at least one scan over the

joined relations[9]. Therefore, the disclosure cost of these algorithms except *index nested loop join* is computed similar to the single relation plans without indexes. That is, $D(R) = N_{ext}(R) * E_{rec}(R) * C_{sens}(R)$ for each joined relation. This can be effectively written as:

$$|R| * C_{sens}(R) \tag{5.5}$$

As for the *index nested loop join* algorithm, the disclosure cost of accessing the outer relation, and the inner relation, is estimated separately. The summation of these costs will yield the overall cost of the join operation. Let $D_o(R)$ be the disclosure cost of accessing the outer relation *R* and $D_i(S)$ be the cost of accessing the inner relation *S*. If the relations *R* and *S* are joined with index nested loop join algorithm, then the records of *R* is scanned at least once. Consequently, all sensitive information in *R* will be revealed. Then, $D_o(R)$ will be $|R| * C_{sens}(R)$. On the other hand, the disclosure cost of accessing the inner relation depends on the index type on it (i.e., primary or secondary index). For each tuple from *R*, the index on *S* will be accessed to find matching tuples. Let $D_{index}(S)$ denote the disclosure cost of accessing S via index. In Section 5.2.1 we explained how to estimate $D_{index}(S)$ for a particular equality or range condition. Using this model the disclosure cost of accessing *S* is the following:

$$D_i(S) = Min\,(|R| * D_{index}(S), |S| * C_{sens}(S)) \tag{5.6}$$

The summation of each index access cost on *S* and the linear scan cost of *R*, will then yield the overall disclosure cost of the index nested loop join algorithm.

[9] For *sort merge join*, we assume that there is no index on the join attribute that can be used for merging the tuples. If there is an index used on one of the tables, the disclosure cost of accessing this table could be estimated similar to the one in *index nested loop join*.

5.3 Multi-Objective Query Optimization

We presented various cost metrics for different access methods in the previous section. We now describe how the query optimizer can be modified to take disclosure risk costs into consideration while query plan generation.

5.3.1 Combining two cost measures

Multi-objective (MO) problems are traditionally solved by converting all objectives into a single objective (SO) function. The ultimate goal in this process is to find the solution that minimizes or maximizes this single objective while satisfying the given constraints [68]. Below, we discuss some of the classical methods and how to apply them to our problem.

Weighted Aggregation Approach: Conversion of the MO function into an SO function is usually carried out by aggregating all objectives in a weighted function. The major issue in this approach is that it requires an a priori knowledge of the relative importance of the objectives [68]. In our case, aggregation can be employed by expressing the disclosure risk and execution cost in monetary terms. Next, this aggregate monetary cost can be minimized subject to the constraints. According to a new study by Forrester Research, the average security breach can cost a company between \$90 and \$305 per lost record [32]. Considering the criticality of the system, the cost of the execution time for the issued queries can also be estimated. Then, these unit weights can be multiplied by the estimates for each query plan. The objective of the query optimizer is minimizing the sum of these costs.

Constraint Approach: An MO problem with n objectives can also be solved by transforming $n-1$ objectives into constraints and minimizing or maximizing only one objective subject to the constraints. In this approach a security administrator sets a *tolerance ratio* $\alpha > 0$ on either

the performance loss or the disclosure risk. Suppose κ denotes the minimum disclosure risk that can be achieved among all possible execution plans. Suppose further that the disclosure risk cannot exceed κ+$\kappa * \alpha\%$ is the security constraint. Then, the objective of the optimizer is to minimize the execution cost while ensuring that the disclosure risk is not higher than κ+$\kappa * \alpha\%$. Consider the following example. For a given query q, suppose that the disclosure risk of query plan p_i is 1000 records and this plan minimizes the disclosure risk among all possible execution plans. As for the *tolerance ratio*, the security administrator sets the value of α at 20. Given α, the optimizer is now free to select any query plan p_j as long as its execution cost is less than the execution cost of p_i and the disclosure risk is less than 1200 records. Hence the optimizer may select a query plan p* which has a disclosure risk between 1000-1200 records and an execution time less than that of p_i.

Note that the above logic also applies to the case where the administrator sets a *tolerance ratio* on the execution time and determines the plan which yields the minimum disclosure risk.

5.3.2 Modifications in MySQL Query Optimizer and InnoDB Storage Engine

Query Engine: The suggested algorithms has been implemented by modifying the source code of MySQL 5.1.38 query optimizer. The implementation is about two-three man months, but for more sophisticated optimizers this cost could be slightly higher. Given a set of query tables, *best_extension_by_limited_search* procedure in *sql_select.cc* file searches for the optimal ordering of these tables and the corresponding optimal access paths to each table. The pseudocode of this procedure is given in Algorithm 4. The name of the recursive procedure in the pseudocode is denoted as *findBestPlan*. This is essentially a dynamically pruning, exhaustive search algorithm.

Therefore, the complexity of the algorithm is O(N!) in the worst case where N is the number of base tables. The procedure constructs all alternative left deep trees by iterating on the number of relations joined so far, while pruning suboptimal trees. We changed three parts of this procedure. These are: *cost estimator*, *pruning conditions* and *best plan selection conditions*. The given pseudocode includes the implementation details of the *Weighted Aggregation* approach, *Constraint on Performance* and *Constraint on Disclosure*, *Max Performance*, and *Min Disclosure* approaches. *Max Performance* is the default implementation of the optimizer which aims to maximize the performance by reducing the overall execution cost. We implemented a second approach called *Min Disclosure* and the goal of this is to reduce the disclosure without considering the overall execution cost.

For each table T in a set of base tables S_{tables}, the execution cost and the disclosure cost of joining this table as the inner table with the tables in the current partial plan is estimated and added to the costs accumulated so far (line 2, 3 in Algorithm 4). Then, the pruning conditions are checked to back track the traversal of the tree when a better plan is not available (line 4-28). When *Weighted Aggregation* approach is used, a weighted cost of disclosure cost and execution cost is computed. Using this aggregate cost a decision of pruning the rest of the path is made (line 17, 18).

For *Constraint on Performance* approach, a maximum acceptable limit on the performance cost is computed using κ and α. (κ is the minimum execution cost that can be achieved among all possible execution plans and α is the tolerance ratio). If the traversal of the nodes in the path does not provide a better performance than the best plan selected so far or a better disclosure cost is not available, the path is pruned (line 22-23). Without a full traversal of the tree, the value of κ can

not be estimated. To tackle this problem two methods can be applied. The first method involves traversing the tree twice. In the first traversal, the value of κ is computed and then used in the second run to find a plan which minimizes the disclosure cost while ensuring that the execution cost does not exceed κ+$\kappa * \alpha\%$. The second method has the advantage of traversing the tree once but this solution requires exponentially large memory space in the worst case. The details of this alternate algorithm is discussed in Section 5.3.3.

The pruning steps of *Constraint on Disclosure* approach are very similar to *Constraint on Performance* approach. For this approach, a maximum acceptable limit on the disclosure cost is computed using κ and α (line 26) and the pruning is applied if needed (line 27-28).

After the pruning steps the current partial plan is expanded with a new table in the tree which is not visited in the current path so far (line 31-32). This recursive procedure call includes accumulated costs and the remaining tables that need to be visited in the child nodes.

After all nodes in the current path in the tree are expanded, the decision of updating the best join order variables is made depending on the selected optimization algorithm (line 34-67). To keep track of the best join order, a path variable *bestJoinOrder* is used and updated once a better path is found. At the end of the traversal of the nodes, the best join order is returned to the execution engine for evaluation.

For the disclosure estimations, we used *best_access_path* procedure to get the number of rows satisfying a predicate and the access method (whether a full scan or index access). Since we use the existing cost estimator of the engine to predict the result size of the operators, the accuracy of the implemented disclosure risk estimator depends on the accuracy of the cost estimator of the optimizer. The other parameters that we use in our cost model are read from an external

configuration file. In a real implementation these parameters could be retrieved from the *catalog* tables rather than reading from a configuration file.

Storage Engine: We modified the source code of MySQL 5.1.38 InnoDB storage engine to integrate the secure coprocessor to the storage engine. The encryption/decryption layer has been implemented by modifying the file manager layer (*fil0fil.c*) and buffer pool layer (*buf0buf.c*) of InnoDB storage engine. Before writing a page to the disk, the page frame is encrypted with the key generated by the secure coprocessor within procedure *fil_io* in *fil0fil.c*. When a page including sensitive information is retrieved to the buffer pool from the disk, the page frame is decrypted in *buf_page_io_complete* with the key generated by the SCP. For each SCP accesses, a log record is generated and flushed to a log file for auditing purposes. In the experiments section we discuss the performance overhead of these operations.

5.3.3 An Alternative Implementation Method for Constraint Approach

During the traversal of the tree, a temporary value κ_{tmp} is stored (Note that κ_{tmp} corresponds to the *bestExec* variable in the pseudo-code). Initially, κ_{tmp} is assigned to the largest possible number in the domain of the variable declared. If the execution cost of the path exceeds κ_{tmp}+$\kappa_{tmp} * \alpha\%$, the path is pruned. Otherwise, the nodes in the path are expanded. When all of the nodes in a path are expanded, the disclosure cost of the path is stored in a heap along with the path information. The top element of the heap includes the information of the path which has the maximum execution cost. During the traversal, the value of κ_{tmp} would decrease as better plans are found. As the value of κ_{tmp} is updated, the nodes at the top of the heap are removed until the path corresponding to the top node has execution cost that is no more than κ_{tmp}+$\kappa_{tmp} * \alpha\%$. Once the traversal is finished,

the path which has minimum disclosure risk is selected among the paths in the heap and this path is returned to the execution engine as the best plan.

5.4 Experiments

To compare the effectiveness of the proposed approaches we conducted several experiments using TPC-H schema and queries. After presenting these observations, we discuss the experiment results related to the performance overhead of key generation and logging mechanism.

5.4.1 Experiments with TPC-H workload

We prepared an experiment bed using the TPC-H queries and the TPC-H schema which includes 8 relations. In total, 1.5 GB of disk space is used to create the TPC-H database (Scale factor 1). Using 5 TPC-H queries (Query # 2, 5, 9, 10, 17) as the basis, we prepared a workload of 100 TPC-H queries. The hardware and software specifications are given in Section 5.4.4 and the details of the workload preparation steps are given in Section 5.4.5.

Experiment with constant weights and constraints:

In this experiment, our goal is to compare the performance and disclosure risk of running a workload while choosing alternative multi-objective optimization algorithms. We run the workload repeatedly (five times) on the modified MySQL database server and monitored the cumulative disclosure risk and the execution time estimated by the query optimizer for different algorithms. In the configuration file we designated *Customer* and *Supplier* tables as the tables including sensitive information. The results are given in Figure 5.3. The dashed bars in the figure represents

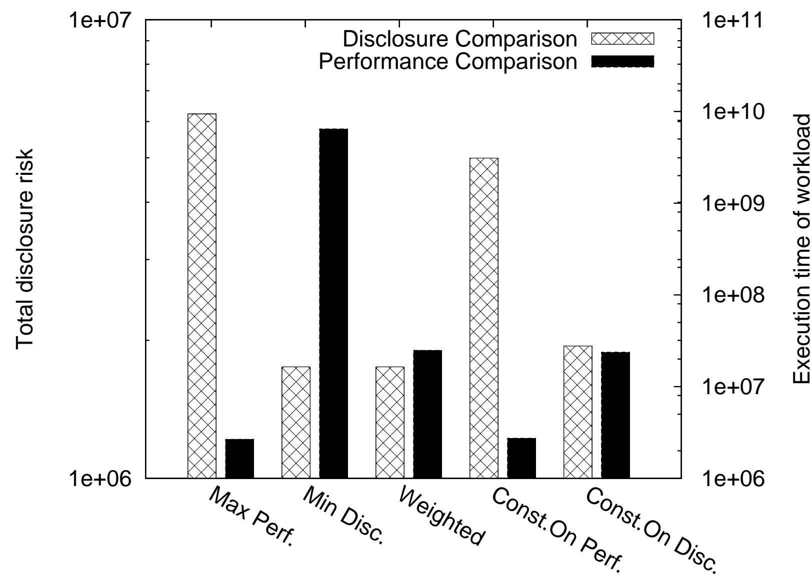

Figure 5.3. Comparison of query optimizer algorithms

the total disclosure risk (see left y axis) for a particular algorithm. The black bars represent the cumulative execution time of the workload (see right y axis). Each pair of bars correspond to a single run of the workload. In the first run *Max Perf.* algorithm is chosen. In this algorithm, the performance is maximized while disregarding the disclosure risk. As seen in the figure, the total execution time of the workload is minimized when this algorithm is chosen. However, this option also maximizes the disclosure risk by retrieving maximum number of sensitive information to the memory. This option represents the default choice of conventional database optimizers. The second algorithm aims to minimize the disclosure risk without considering the performance penalty. Note that the execution time of the workload in this case is about three orders of magnitude greater than the one in *Max Perf.* algorithm while reducing the disclosure risk by a factor of three. The third pair in the figure corresponds to the *weighted aggregation* approach. As we discussed earlier the average security breach can cost a company about $200 per lost record. So we set the weight

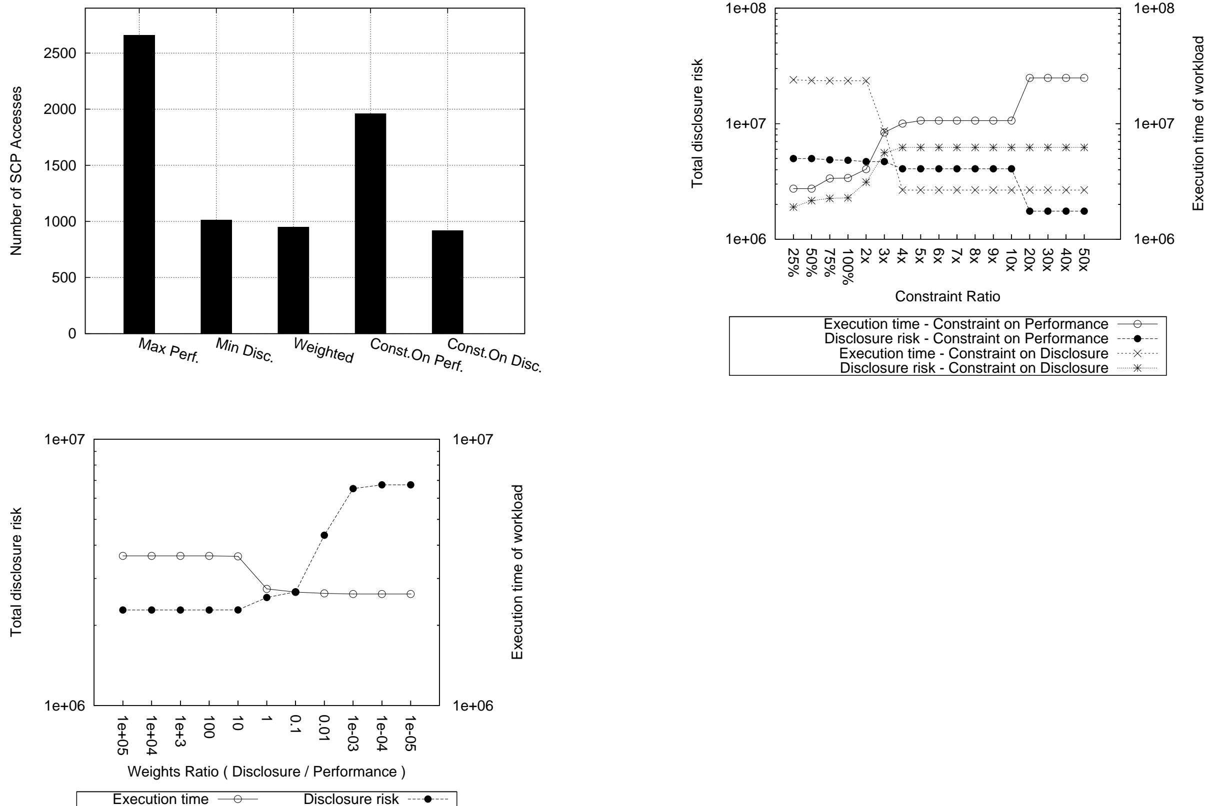

Figure 5.4. a) Number of SCP accesses during the execution of the workload b) Constraint on disclosure and constraint on performance c) Weighted aggregation approach

of the disclosure as 200. As for the execution cost, we assumed that each delayed second of the queries costs a dollar for the company. So we set the disclosure cost weight as 1, in this particular scenario. Given these parameters, we observed that the cumulative disclosure risk of the workload in this case is similar to the one in *Min Disc.* approach whereas the execution time is reduced by about two orders of magnitude. Compared to the *Max Perf.* approach, the execution cost is increased. If one considers both the disclosure risk and execution cost of the workload this approach would yield more preferable results compared to both *Max Perf.* and *Min Disc.* approaches. As for the fourth experiment, we tuned the database with the *constraint* approach parameters. In this run, we assumed that there is 30 % limit on the performance loss. That is, the optimizer aims

to minimize the disclosure risk while guaranteeing that there is no performance loss more than 30 % compared to the best running time. As seen in this figure, there is a slight increase in the overall workload execution performance (less than 3 %) whereas the disclosure risk is better than the one in *Max Perf.* approach (about 20 %). The last experiment is similar to the fourth one. In this run, we assume that there is 30 % limit on the disclosure risk. The results are similar to what we observe in *weighted aggregation* approach. Compared to the other three approaches this one provides more balanced results in terms of both performance and disclosure.

We implemented a program to analyse the audit logs generated by the DBMS. Using this program, we counted the number of SCP accesses during the execution of the workload (see Figure 5.4-a). We observed that the number of SCP accesses in *Max Perf.* is reduced by a factor of three in *Min Disc.*, *Weighted Aggregation* and *Constraint on Disclosure* approaches which is parallel with the disclosure estimations of the query optimizer.

Experiment with changing constraints:

In this experiment, our goal is to observe the impact of constraint parameter on the disclosure and performance of a given workload. Using the database instance described above we run the same workload on MySQL while changing the constraint parameter. Each point in Figure 5.4-b represents the disclosure and execution time of a single run of the workload. As discussed in Section 5.2, a constraint could be defined either on disclosure or performance. Therefore, for various constraint ratios (see the x axis), we run the workloads for both “Constraint on Performance (COP)” and “Constraint on Disclosure (COD)”. In the COP approach, as the constraint ratio on performance is relaxed, the execution time increases while the disclosure risk reduces. In the COD approach, the opposite of this behavior is observed as expected.

Experiment with changing weights:

In this experiment, our goal is to observe the impact of different weights on the disclosure and performance of a given workload. Similar to the constraint approach experiments, each point in Figure 5.4-c represents the disclosure and execution time of a single run of the workload. For various weight ratios (ratio of disclosure cost to the execution cost), we run the workload. As the weight of the performance cost gets higher, the execution time starts decreasing while the disclosure risk starts increasing.

5.4.2 Performance Overhead of Key Generation and Audit Log Generation Mechanism

As we discussed earlier accessing the key generation hardware at smaller granularities such as single key per record or single key per page will incur too much performance cost while reducing the disclosure risk. Therefore, we proposed the idea of extent level key management. To prove the effectiveness of this approach we run a query workload and measure the overall execution time while changing the granularity of key generation. We observed that generating keys (i.e., accessing the HSM) per each extent provides substantial improvement in terms of both performance and disclosure. The details of the experiments are discussed in Section 5.4.3. Additionally, we measured the overhead of generating and flushing the audit logs to the disk while executing the queries. We observed that this process incurs less than 3 % performance overhead during the execution of workloads.

5.4.3 Performance Overhead of Key Generation and audit log generation mechanism

Using a similar technique described in Section 5.4 we prepared a workload of 50 TPC-H queries and run on the modified MySQL database server. In this experiment setting, we created a new

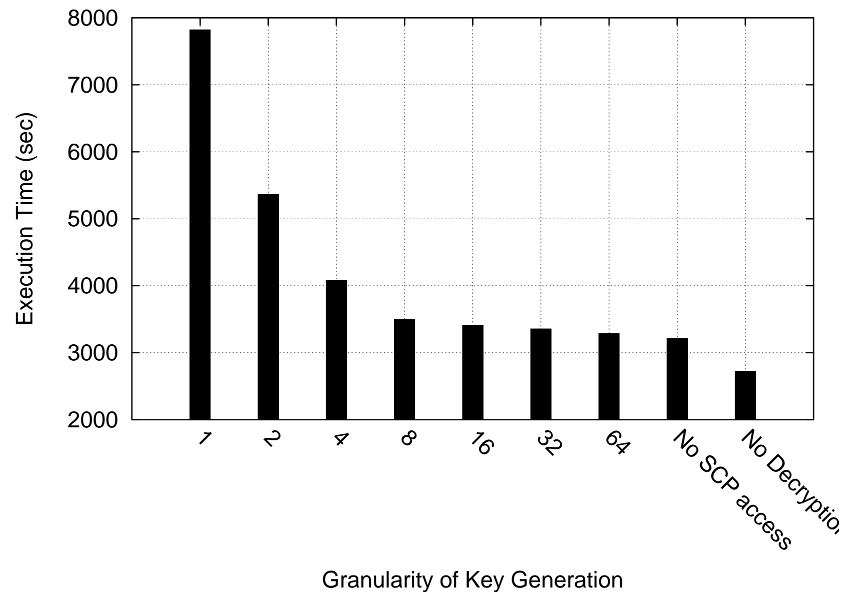

Figure 5.5. Performance overhead of key generation for different granularities

TPC-H instance where we assume all attributes of the tables are sensitive and therefore stored in encrypted format in the disk. Each bar in Figure 5.5 represents the execution time of the workload for different key generation granularity. The left-most bar shows the overall execution time when the HSM device is accessed for each page retrieved from the disk. As the granularity increases the execution time declines. After a certain number of pages (single key per 8 pages) the overhead of key generation remains steady. These results show that key generation per extent would be a good design choice in terms of both performance and disclosure risk. The second right-most bar in this figure shows the execution time when a single key is used for all page decryption operations. Therefore there is no HSM accesses. Note that the execution time in this case is almost similar to the case where the HSM device is accessed per each extent. This observation shows that using extent level key generation incurs no extra cost compared to using a single master key for the database. This, in turn, reduces the disclosure cost while having no extra performance cost. The

last bar in the figure shows the execution time when there is no decryption at all. Hence, one can observe the overhead of the decryption operations by comparing this one with the other execution times. We observe 18 % increase in the execution time of the workload when we assume that all accessed tables include sensitive information.

5.4.4 Hardware & Software Specifications

All experiments are conducted on a 32 bit SLES 10.2 (Linux kernel 2.6.16.60) operating system. For the prototype database implementation, we modified the source code of MySQL 5.1.38. As for the platform, we used an IBM x3500 which has 16GB of main memory and a 8 core Intel(R) Xeon(R) CPU E5420 @ 2.50GHz processor.

As for the tamper resistant hardware we used IBM 4764 PCI-X Cryptographic Coprocessor (See Section 2.7 for the details of IBM 4764). For the cryptographic operations in both the server side and the secure co-processor we used AES in CBC mode with 16 byte key size as the encryption algorithm.

5.4.5 Preparation of TPC-H database instance and query workload

The TPC-H benchmark is a decision support benchmark widely used in the database community to assess the performance of very large database management systems [89]. The benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and provide answers to critical business questions. Using the TPC-H schema and queries, we prepared an Operational Data Store (ODS) environment to compare the effectiveness of proposed query optimization approaches. Below, we explain the details of how we prepared the experiment bed and the workloads.

We prepared a workload using the TPC-H queries and the TPC-H schema which includes 8 relations. In total, 1.5 GB of disk space is used to create the TPC-H database (Scale factor 1). During the execution of the workload, the buffer pool size of the database was set to 80 MB (default buffer pool size in InnoDB).

The workload used in the experiments is constructed using 5 TPC-H queries (Query # 2, 5, 9, 10, 17) with the objective of maximizing the number of joined tables during the execution of each query. We subsequently modified these queries to simulate an Operational Data Store (ODS) environment where some of the queries in the workload require processing large ranges of data while others process smaller ranges.

The major difference between an ODS and a data warehouse (DW) is that the former is used for short-term, mission-critical decisions while the latter is used for medium and long-range decisions. The data in a DW typically spans a five to ten years horizon while an ODS contains data that covers a range of 60 to 90 days or even shorter. In order to simulate an ODS environment, more predicates are added to the "where" clause of the TPC-H queries. This in turn, reduces the number of rows returned. As a result, we obtained a workload comprising of queries which scan both small and large ranges of data. The query given in Table 5.1 provides an example for this modification.

In this query, the predicate "o_orderkey between 261658 AND 271658" is added to the original query to reduce the range of the data that has been accessed. In order to reduce the buffer pool hit ratio, the predicate values are randomly drawn from a uniform distribution with range (0, n) where n is the maximum in the domain of the predicate attribute. Using this technique, 100 TPC-H queries are generated and issued to the database.

5.5 Discussion

5.5.1 Motivating Example For Query Optimization

Suppose that the following query is issued to the database: $Select * From\ A, B, C\ Where\ A.pk = B.pk\ and\ B.pk = C.pk$. Table A includes sensitive information and all of the data pages of this table are stored in an encrypted format in the disk. Tables B and C include non-sensitive information and the records of these tables are not encrypted. Suppose further that two alternative join plans given in Figure 5.6 are considered. Without the knowledge of the sensitivity of the tables, the query optimizer selects the plan on the left assuming that this is the least costly plan. However, this plan requires accessing all of the rows of table A and therefore maximizes the risk of disclosure. To be able to access all of the rows, all of the data pages of A should be decrypted. Suppose that table A on the second plan is accessed via index only plan and retrieves less than 2 % of the data pages of A to the memory. Compared to the former plan this could be a slower one but may reduce the disclosure risk effectively. If an attacker monitors the content of the memory during the execution of this query, the amount of information leakage in the second plan would be much less than the one in the first plan. Being aware of the sensitive information, a query optimizer may reduce the risk of disclosure effectively while maximizing the performance.

5.5.2 Granularity of Key Management

To minimize the risk of disclosure, we propose the idea of "extent level key management". In this approach a unique key is used for each extent to encrypt/decrypt the data pages. An extent is defined as a set of contiguous blocks allocated in a database tablespace. For instance, if the extent size in MySQL-InnoDB is configured as 128KB then eight contiguous data pages can be placed in one extent assuming that 16KB data pages (physical blocks) are used. For OLTP environments the

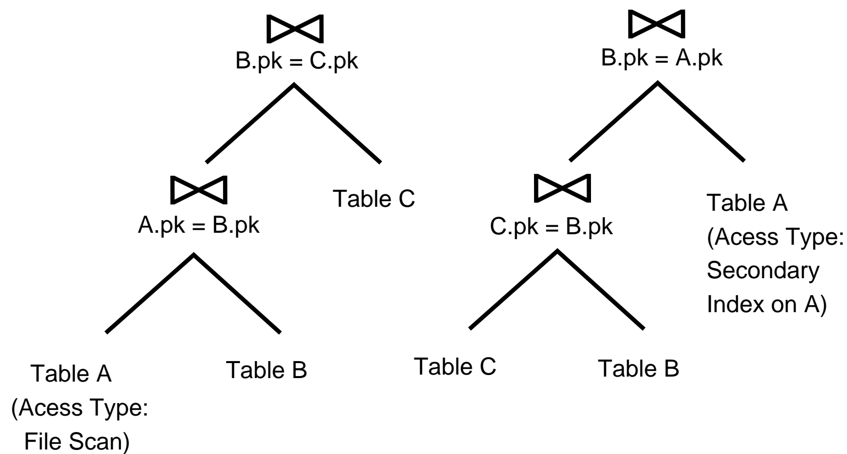

Figure 5.6. Query execution plans

typical size of an extent could vary between 4 to 16 pages whereas for DSS (OLAP) environments larger extent sizes are preferred: 8 to 64 pages per extent. All of the pages within an extent are encrypted with the same key. Whenever a page within an extent is retrieved to the memory the extent key is used to decrypt that page. In our approach extent keys are not stored physically but generated whenever they are needed. To be able to decrypt the content of a page the extent key is requested from the HSM device. This request includes the extent ID. HSM device computes the hash of this id and encrypts the output of this hash function with the master key stored within the HSM. Therefore a MAC based approach is used to generate the extent keys. Once generated the key is used to decrypt the requested page.

To minimize the information disclosure, one can suggest key management at the level of data records, (i.e., one key per record). All of these keys are encrypted with a master key which is protected by a hardware security module. Whenever a row needs to be accessed, the key of this row will be decrypted by the HSM and then used to decrypt the row. This approach would certainly reduce the amount of disclosure during the attack because only the keys used to decrypt the rows in use will be compromised. Although this scheme provides much better protection, it

is not preferable due to the performance concerns. First and foremost, for each row access, the HSM needs to be accessed. Such approach would drastically slow down the query processing performance. The second problem is the burden of key initialization. Key initialization is a costly operation and can significantly affect the query processing time if it is performed for each row. The third problem is the encryption/decryption speed of HSM devices. Due to limited processing capacity of these devices, decryption speed of the keys may significantly slow down the data processing. Assuming that 128 bit keys are used, just scanning a table with six million records could require more than one minute extra processing time with a IBM Secure Coprocessor as the HSM device. The fourth problem that we need to address is the storage of encryption keys. An efficient key management strategy should be employed to tackle the issues related to storage of the keys in DBMS. Keeping a single key per row may create a non-negligible storage problem if there is high contention for the shared memory space in the buffer pool.

Key generation in the level of records may not be preferable. However, one could argue that a single key could be used for the encryption of each page. Although this approach provides better protection, we observed that extent level key management provides significant performance improvement compared to the page level key approach. In our preliminary experiments we used a 4764 IBM Secure Coprocessor as the HSM device which is connected to a server machine over PCI-X bus. The execution times of different operations for a single database page are given in Figure 5.7. The first bar represents the time to generate a single extent key with the HSM device. The second bar and third bar shows the time to decrypt a single data page with the HSM and CPU correspondingly. As seen in this figure decryption speed of the CPU is more than an order of magnitude faster than the HSM. Therefore performing the decryption of bulk data in the server side is much faster. The third and fourth bars show how much time is spent to retrieve a single page

to the memory from the disk with a sequential and random disk access pattern correspondingly. Not surprisingly the random access cost is much greater than the sequential access cost due to the physical head movements of the magnetic disk. In OLAP type database applications the disk I/O operations are dominated by the sequential disk accesses because most of the queries require scanning the tables. If page level key generation approach is used the key generation cost would exceed the I/O cost because key generation is twice as costly as reading a page from the disk sequentially. Therefore the query execution time would be dominated by the key generation cost which is not desirable. On the other hand if extent level approach is preferred only a single key generation is required for all pages within an extent. Assuming that we have eight pages within an extent only a single key is generated per eight page accesses. Therefore key generation process does not create a bottleneck while processing the queries.

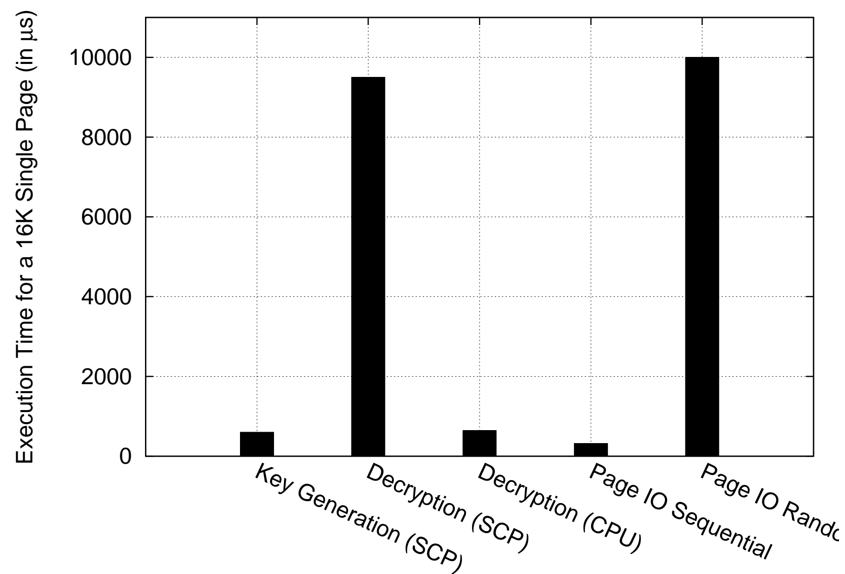

Figure 5.7. Execution times of operations

Also for random disk accesses using extent level keys does not yield any problem because random access cost of a page is much greater than the cost of key generation. Hence, query processing performance will not get stalled by the key generation process.

5.5.3 Optimal Predicate Ordering

In our model since the pages are decrypted as soon as they are brought into the memory, the disclosure risk is only proportional to the number of different extent keys that are accessed. In an alternate model where one has a choice to postpone the decryption of a page the optimal predicate ordering might differ significantly as illustrated by the following example - consider three selection uncorrelated predicates p_1, p_2 and p_3 with selectivity 0.3, 0.4 and 0.5 respectively. Also, let p_1 be a sensitive predicate (i.e., one that involves a sensitive attribute) and p2 and p3 be predicates over non-sensitive attributes. Then, simply using the performance criteria the optimizer would schedule p_1 to be evaluated first, but this would mean a complete disclosure assuming a table scan is used. Instead, if p_2 and p_3 are evaluated first and the resulting tuples pipelined to p_1, we can expect only 0.2 fraction of the keys to be exposed under the random spreads assumption. In multi-relational queries where join ordering is required, pushing down selections all the way in the query tree may significantly increase the exposure risk in general. A smart way to explore all possible orderings of joins and selections can be carried out by masking a selection predicate as a join with a virtual table containing a single tuple in the range of the predicate. For instance, a condition like $\sigma_{age=50}(R)$ can be modeled as natural join between R and a virtual table $V(age)$ with a single record having value of attribute age as 50. Other common selection conditions can also be represented similarly. Unfortunately, given the exponentially large number of ordering,

this technique is likely to degrade the optimizer severely. In a related problem regarding expensive predicate placement in query plans, it might be possible to develop an approach similar to what was done in Hellerstein [sigmod93]. They consider the problem of expensive predicate ordering in query plans and use an optimal constraint aware sequencing algorithm to order the operators optimally be defining a notion of a stream in a query plan. Anyway, we do not go into the details of this alternative model in this section and this remains an important direction of our future work.

5.6 Conclusion

In this work, we consider the problem of estimating disclosure cost incurred in query processing over data with sensitive information. Specifically, we study the risk of sensitive data exposure when the adversary is able to access the contents in the main memory of the server during query execution. We enhanced the query optimization techniques to consider both the sensitive data disclosure risk and the execution costs and provide knobs to the administrator to select the desired tradeoff between the two. In addition, we developed methods to securely generate audit logs which is critical for forensic analysis after an attack occurs. Our results indicate that careful consideration of disclosure risk and query execution cost is indeed required to balance security and efficiency of query processing.

```
1  Recursive Procedure: findBestPlan
   Input: execCost, discCost, bestExec, bestDisc, S_tables, bestJoinOrder
   // S_tables is the set of tables to traverse
   Output: bestJoinOrder
2  foreach table T in S_tables do
3      cummDiscCost = discCost + estimateDiscCost(T) ;
4      cummExecCost = execCost + estimateExecCost(T) ;
5      if OptAlgorithm = "Max. Perf" then
6          if cummExecCost > bestExecCost then
7              backTrack ;
               // Prune sub-optimal plan
8      else if OptAlgorithm = "Min. Disc" then
9          if cummDiscCost > bestDisc then
10             backTrack ;
11     else if OptAlgorithm = "Weighted Aggregation" then
12         bestWeightedCost =
13         (discWeight * bestDisc) + (perfWeight * bestExec) ;
14         cummWeightedCost =
15         (discWeight * cummDiscCost) + (perfWeight * cummExecCost) ;
16         if cummWeightedCost > bestWeightedCost then
17             backTrack ;
18     else if OptAlgorithm = "Constraint On Performance" then
           // κ: minimum execution cost among all possible query plans
           // α: tolerance ratio on performance
19         maxAllowedCost = κ+κ * α% ;
20         if cummExecCost > maxAllowedCost and cummDiscCost > bestDisc then
21             backTrack ;
22     else if OptAlgorithm = "Constraint On Disclosure" then
           // κ: minimum disclosure cost among all possible query plans
           // α: tolerance ratio on disclosure
23         maxAllowedCost = κ+κ * α% ;
24         if cummDiscCost > maxAllowedCost and cummExecCost > bestExec then
25             backTrack ;
       // Recursively expand the current partial plan
26     if S_tables − T is not empty then
27         findBestPlan(cummExecCost, cummDiscCost, bestExec, bestDisc, S_tables − T, bestJoinOrder) ;
28     if OptAlgorithm = "Max. Perf" then
29         if cummExecCost < bestExec then
               // A better path is available
30             bestExec = cummExecCost ;
31             updateBestJoinOrder(T, bestJoinOrder) ;
32     else if OptAlgorithm = "Min. Disc" then
33         if cummDiscCost < bestDisc then
               // A better path is available
34             bestDisc = cummDiscCost ;
35             updateBestJoinOrder(T, bestJoinOrder) ;
36     else if OptAlgorithm = "Weighted Aggregation" then
37         bestWeightedCost =
38         (discWeight * bestDisc) + (perfWeight * bestExec) ;
39         cummWeightedCost =
40         (discWeight * cummDiscCost) + (perfWeight * cummExecCost) ;
41         if cummWeightedCost < bestWeightedCost then
               // A better path is available
42             bestExec = cummExecCost ;
43             bestDisc = cummDiscCost ;
44             updateBestJoinOrder(T, bestJoinOrder) ;
45     else if OptAlgorithm = "Constraint On Performance" then
           // κ: minimum execution cost among all possible query plans
           // α: tolerance ratio on performance
46         maxAllowedCost = κ+κ * α% ;
47         if cummExecCost < maxAllowedCost and cummDiscCost < bestDisc then
               // A better path is available
48             bestExec = cummExecCost ;
49             bestDisc = cummDiscCost ;
50             updateBestJoinOrder(T, bestJoinOrder) ;
51     else if OptAlgorithm = "Constraint On Disclosure" then
           // κ: minimum disclosure cost among all possible query plans
           // α: tolerance ratio on disclosure
52         maxAllowedCost = κ+κ * α% ;
53         if cummDiscCost < maxAllowedCost and cummExecCost < bestExec then
               // A better path is available
54             bestExec = cummExecCost ;
55             bestDisc = cummDiscCost ;
56             updateBestJoinOrder(T, bestJoinOrder) ;
57 end
```

Algorithm 4: Pseudocode of the modified query optimization algorithm (with additional pruning and decision steps)

CHAPTER 6

SECURE MANAGEMENT OF CLINICAL GENOMICS DATA WITH CRYPTOGRAPHIC HARDWARE

In this section, we consider a fully untrusted attack model for a client server architecture which can be used for processing biomedical genomic data. We propose a framework which supports joining genomic data records of patients taken from different hospitals and querying these records for scientific analyses without revealing the identity of the patients. Compared to the previous section, we assume that the attacker on the server has access to both the memory and storage devices. To be able to protect the privacy of the records without violating the privacy requirements we use a tamper proof secure coprocessor on the server side. Below we describe the details of this framework.

6.1 Introduction

The integration of clinical decision support tools and high throughput technologies into the healthcare domain is evolving the practice of medicine from a one-size-fits-all model toward a personalized system [56]. To further this goal, biomedical scientists are conducting large-scale investigations between patients' clinical status and molecular data, such as DNA sequences, to determine how variation in the latter influences patients' susceptibility to disease, response to treatment, and potential to succumb to adverse events (e.g., drug-drug interactions) [35, 37].

The amount of data needed to conduct such studies is often beyond the capability of a sole institution [70] and emerging regulations, such as the U.S. National Institutes of Health (NIH)

genome wide association study (GWAS) policy, encourage biomedical scientists to share person-specific data for reuse [67]. In order to support such regulations, various technologies have been established to facilitate data sharing, such as the Database of Genotypes and Phenotypes (dbGaP) in the U.S. [57] and the Welcomme Trust's biobanking program in the U.K. [75]. Notably, these systems centralize data from disparate organizations. To ensure the continued growth of such tools, it is critical to ensure the security of such systems, as well as the privacy of the records they incorporate.

Traditionally, administrators attempted to achieve privacy by "de-identifying" data through the suppression of particular attributes, such personal names, dates or geocodes. However, de-identification does not guarantee protection from "re-identification" [59, 12] and may obscure certain types of information that are critical to longitudinal or epidemiologic studies. As an alternative, security frameworks have been devised to manage clinical genomics data, in its most specific form, in a centralized database [50]. In such frameworks, data holders transmit encrypted versions of clinical genomics records to a third party. Subsequently, the third party administrator could integrate data coming from different sources and then execute queries on behalf of a scientist (e.g., How many records contain a diagnosis of *juvenile diabetes* and a particular *genomic variant*?) without decrypting any record in the database. To achieve this goal, the administrator at the third party evaluates a query against each stored record securely, and the results are sent to a separate third party who is in control of the decryption keys and learns only the *aggregation* of the result (i.e., the number of satisfying records). The result is then revealed to the researcher. One advantage of such a framework is that there is no opportunity to decrypt individual records unless both third parties collude. Thus, if a hacker breaks into one of the third party's computer systems, he cannot expose the sensitive information.

The framework just mentioned, however, has several drawbacks which limits its general applicability. First, to facilitate evaluation of a researcher's query against the database, a computationally expensive cryptographic technique needs to be executed. Second, it may not be practical to set up multiple disparate third parties due to economic, trust, or legal concerns. Third, even when the first two problems are surmountable, there remain potential bandwidth challenges because a non-trivial amount of encrypted data must be communicated between the parties involved in the protocol.

The overarching goal of this work is overcome the aforementioned limitations through the application of secure hardware. We design a framework that leverages tamper-resistant hardware to enable secure clinical genomics data storage and processing at a single third party. To the best of our knowledge, this is the first work to propose such a framework for biomedical data.

Some of the advantages of the proposed framework are as follows:

- We illustrate that by using the capabilities of the cryptographic hardware, different organizations can contribute their data to centralized sites, where they can be joined and queried, in a secure manner.

- We show that a small program can be executed on secure hardware attached to a third party's server that hosts the clinical genomics data. In the process, we demonstrate that the program has sufficient power to facilitate common queries used in emerging biomedical investigations, such as count queries as well as join operations.

- We show that our strategy can counter various attacks. In particular, even if the system is hacked, as long as the cryptogprahic hardware is not compromised, no information that is stored (whether it is genomic, clinical, or demographic) will be leaked.

- We provide experimental studies to demonstrate feasibility with existing hardware for real world applications.

6.2 Description of the Proposed Framework

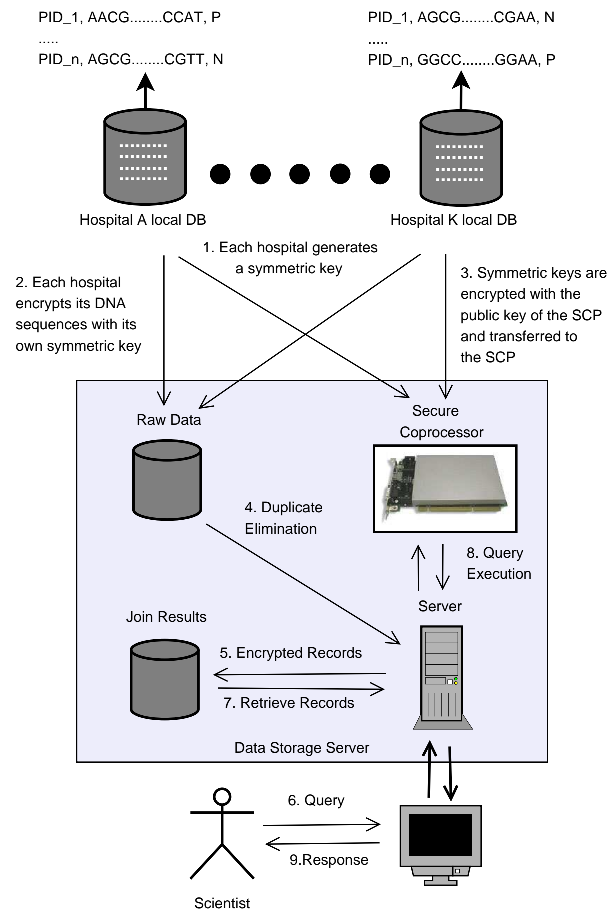

Figure 6.1. Proposed framework for management of biomedical data in third party cryptographic hardware.

For convenience, we assume that the set of data owners correspond to a set of hospitals. In our framework, each hospital can verify that the server to which they plan to transfer the clinical genomics data contains an approved SCP, and provide the SCP with the data using the SCP's public key. For efficiency purposes, the data will be encrypted using a symmetric key encryption

scheme and the key used for the encryption will itself be encrypted with the public key of the SCP.[1] The trusted coprocessor can now execute join operations and run various aggregate queries over encrypted clinical genomics data. Since only the SCP can recover the key needed for decrypting the data, no individual with access to the server can interpret the stored data.

Clearly, encrypted storage can prevent the disclosure of the data even if the server is hacked. Any attempt by the server to take control of, or tamper with, the coprocessor, either through software or physically, will clear the coprocessor. In doing so the keys will be destroyed, which eliminates the ability to decrypt any information stored or transmitted.

To guarantee that the application running on the secure coprocessor is non-malicious software and was loaded by a trusted OS, we use the remote attestation mechanism provided by the secure coprocessor [34].

The following subsection provides further details related to our framework.

6.2.1 Architecture

The overview of the proposed architecture is illustrated in Figure 6.1. Our framework incorporates a third party called the Data Storage Server (DS). The encrypted records of the hospitals are stored in this server.

We assume that the DS is untrusted. All of the interactions among the storage devices, the server and the secure coprocessor are monitored by an adversary. To perform the data operations without disclosing any sensitive information, all of the data processing operations are performed within the secure coprocessor. To achieve such processing, each hospital first generates a symmet-

[1] In symmetric key encryption, both sender and receiver share the same key to encrypt/decrypt messages. Symmetric key encryption is typically much faster than public key encryption. See [81] for more details.

ric encryption key and encrypts its records. This key is later transferred to the secure coprocessor located in the DS. The SCP provides a secure Ethernet channel through which the hospitals can communicate and transfer the key.

The information flow in this framework consists of two phases: *Data integration phase* and *Query processing phase*. In the data integration phase, the DS receives the patient records from the hospitals and eliminates the duplicates by executing the join operations on the secure coprocessor. In the query processing phase, the DS receives the queries of the scientists and executes these queries on the encrypted patient records with the help of the secure coprocessor.

For the data integration phase, we use the sovereign join algorithms introduced in [5]. Sovereign joins are applied to prevent adversaries from making inferences through the observation of the interactions between a secure coprocessor and the server to which it is attached. Agrawal et al. [5] formulated this problem and propose alternative techniques to prevent information leakage through patterns in I/O while maximizing the performance. We provide the implementation details of this protocol for our framework in Section 6.4.

For the query processing phase, we propose a secure protocol that prevents adversaries from inferring information through interactions between the server and the secure coprocessor. We describe the details of this protocol in Section 6.2.3.

Below we describe a step-by-step walkthrough of the framework (See Figure 6.1 for the corresponding protocol).

- Step 1: Each hospital generates a symmetric key K_i to encrypt the data of their patients. Without loss of generality, we assume the underlying data consists of structured clinical terms, DNA sequences, and demographics.

- Step 2: The records of hospital H_i are encrypted with symmetric key K_i and transferred to DS.

- Step 3: Each hospital H_i encrypts its private key K_i with the public key of the SCP and transfers the key to the SCP over the Ethernet channel of the SCP. An attacker at the server side cannot interpret the encrypted clinical genomics data because the keys are secured in tamper-resistent hardware.

- Step 4: The server retrieves the records from the raw data disk for duplicate elimination.

- Step 5: The server communicates with the SCP and performs the join operations on the encrypted records and eliminates duplicate records.

- Step 6: The encrypted join results are written to a disk.

- Step 7: Suppose that a researcher wants to learn how many records in the database contain a particular combination of SNPs when a patient is diagnosed with a certain disease. This can be represented in a logical query, such that $\{SNP_{j_1} = AT \wedge \cdots \wedge SNP_{j_k} = CG \wedge$ Disease Diagnosis $=$ positive (P)$\}$, where $j_1, ..., j_k$ is an arbitrary subsequence of the data attributes. The server first receives the query.

- Step 8: Next, the encrypted records are fetched from the storage device for query processing.

- Step 9: The query is forwarded to the SCP along with the encrypted attributes. The SCP processes the query and returns the response to the server.

- Step 10: Finally, the response is transferred to the researcher.

In the following subsection, we provide details about how biomedical data, with a focus on genomic sequences, are stored and queried in our framework.

6.2.2 Data Representation

Genomic sequence data is constructed from the four letter alphabet of nucleotides $\{A, C, G, T\}$, each of which can be represented as a pair of bits as shown in Table 6.1(a). In the context of GWAS, we adopt the model used in standard genome management software [79], where each SNP is modeled as a pair of nucleotides. Using this representation, each genomic sequence can be modeled as a series of binary values.[2] Table 6.1(b) presents four samples with three SNPs each, in the four letter alphabet along with the patient ID (PID), diagnosis information and the corresponding binary representations (based on Table 6.1(a)). PIDs are used to compare the records taken from different medical institutions and eliminate duplicates.

As mentioned earlier, we use a symmetric key encryption algorithm to encrypt the biomedical records (See Table 6.1(c)). Table 6.1(d) depicts the application of encryption function and Table 6.1(e) the resulting encrypted data.

From a practical perspective, encrypting each letter separately requires padding each encrypted block, which unnecessarily wastes memory. To reduce memory consumption, we construct blocks of biomedical data, which are encrypted one block at a time. We use the AES block cipher algorithm [66] to perform encryption, which supports a block size of 128-bits.

[2]We note the framework can handle various representations of genomic and health data and we show SNPs without loss of generality.

Table 6.1. Encryption of Clinical Genomics Data

(a) Mapping from a nucleotide alphabet to a two-bit binary value

$A \rightarrow 00$
$C \rightarrow 01$
$G \rightarrow 10$
$T \rightarrow 11$

(b) Binary representations of SNP sequence samples

$PID_1\|AA\|GG\|AT$	0000 1010 0011
$PID_2\|AG\|CG\|AA$	0010 0110 0000
$PID_3\|GG\|CC\|AT$	1010 0101 0011
$PID_4\|AG\|CG\|TT$	0010 0110 1111

(c) Original data (Each block has 64 letters)

B_1 :	$PID_1, AACG$........$CCAT$, Diagnosis=P	00000110........01010011
B_2 :	$PID_2, AGCG$........$CGAA$, Diagnosis=N	00100110........01100000
B_3 :	$PID_3, GGCC$........$GGAA$, Diagnosis=P	10100101........10100000
B_4 :	$PID_4, AGCG$........$CGTT$, Diagnosis=N	00100110........01101111

(d) Encrypting the data

$E_{K_i}(B_1)$:	E_{K_i}(00000110........01010011)
$E_{K_i}(B_2)$:	E_{K_i}(00100110........01100000)
$E_{K_i}(B_3)$:	E_{K_i}(10100101........10100000)
$E_{K_i}(B_4)$:	E_{K_i}(00100110........01101111)

(e) The encrypted data

$E_{K_i}(B_1)$:	01001110........01001011
$E_{K_i}(B_2)$:	10111010........00101101
$E_{K_i}(B_3)$:	01001110........11011010
$E_{K_i}(B_4)$:	01101011........00101101

6.2.3 Secure Count Queries

One of the more common tasks that genome-based scientists need to perform is to determine how many samples satisfy certain characteristics. For example, scientists are interested in learning if there exist associations between SNPs and a range of clinical phenotypes [25, 60]. To enable a researcher to discover such associations, a secure framework needs to report the frequency of genomic and clinical feature occurrences. Such frequencies can be discovered by first counting the number of records the pattern occurs in and then normalizing this quantity by the total number of records in the database. Unfortunately, count queries were not designed to be executed over encrypted data in an untrusted server. Thus, we developed a secure count protocol which calculates the frequency of scientist-specified patterns without compromising the data.

The overview of the protocol is visually depicted in Figure 6.2. The protocol is executed between the server and the SCP to evaluate the queries. The server has a dataset of n records with m attributes and the goal is to process a query q with k predicates defined over the columns of the dataset. Each attribute of the records is encrypted with the symmetric key (K) of the hospital using AES in CTR mode of encryption [55]. Before the execution of the query processing phase, the encryption key K is transferred to the SCP over a secure Ethernet channel. Now the goal is to transfer these records to the SCP and determine the number that satisfy the query.

As mentioned earlier, SCPs have limited memory. Thus, rather than sending all attributes of the records to the SCP for decryption, the server sends data from only the attributes of the predicate values defined in the query. Suppose that the first predicate, P_j is defined on the jth column of the record. As shown in Figure 6.2, the server sends $E_K(Dataset[0, j])$ to the SCP along with the row ID, column ID and the predicate P_j. Once these values are received, the SCP

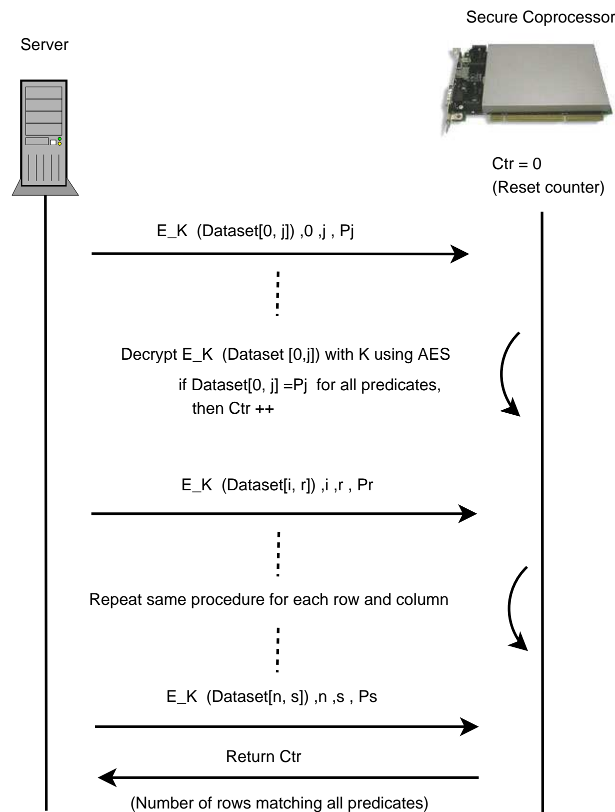

Figure 6.2. An overview of the secure count protocol.

decrypts $E_K(Dataset[0, j])$ with *K*. After the decryption, the SCP matches the predicate values with the corresponding column's values. The SCP keeps a counter value (or set of values) to store the number of records satisfying all predicates while processing the records. For a particular record, if all predicates are matched successfully, then the counter is incremented. After the final iteration, the counter value, which corresponds to the response of the query, is returned to the server. By applying clever buffering schemes (e.g., [5]), such an approach could handle any amount of data.

Depending on the security requirements of the application, the query results returned to a researcher can be encrypted as follows. Prior to the protocol, a secure SSL channel can be

established between the researcher and the SCP.[3] Using this channel, the query results can be transferred to the researcher without revealing the content to the server or any other observers.

6.3 Security Analysis

Since the operations inside the secure coprocessor are executed securely and all the sensitive biomedical data that is read or written to the server by the coprocessor are encrypted, the information leakage is possible only due to the access patterns of the algorithms running inside the secure coprocessor. Below we prove that for the two algorithms specified in this section, an adversary cannot learn anything by observing the access patterns. This is achieved by proving that the access patterns of the algorithms run on the secure coprocessor do not change based on the input [76].

Definition 1. *The access pattern for algorithm A on input x (i.e., $A(x)$) is the sequence of accesses performed by the secure coprocessor. An algorithm A is said to run securely on the secure coprocessor, if for any two equal length inputs x, x' of the client, the access patterns $A(x)$ and $A(x')$ are computationally indistinguishable for any observer, except the secure coprocessor.*

We can now prove that the count query given in Figure 6.2 is secure according to Definition 1.

Theorem 1. *The count protocol discussed in Figure 6.2 is secure according to Definition 1.*

Proof. The count protocol always reads each row of the input and outputs just one encrypted count value. Therefore for any two equal length inputs x, x', the access patterns of the count algorithm

[3] IBM 4764 SCP provides functionality to establish SSL channel to communicate with remote applications.

are exactly the same. Furthermore, due to the security of the encryption scheme, any polynomial-time adversary cannot distinguish between the different encrypted output count values. Therefore, the count protocol satisfies the Definition 1. □

A similar proof for the join algorithms is provided in [5].

Traditional oblivious RAM protocols [76] provide general purpose solutions to hide the access pattern of any algorithm. In contrast, we provide a solution that is tailored for genomic data processing that is much more efficient than such generic solutions. General purpose oblivious RAM solutions require $O(log^2(n))$ overhead for each record access where n is the total number of records. In our case, for count queries, for each record access, our overhead is just $O(1)$.

6.4 Joining Patient Records

To perform join operations we implement the sovereign join algorithms introduced in [5].

The join operations in our framework require interactions between the server and the SCP. The implementation of the algorithms for the server and SCP are provided in Algorithm 5 and Algorithm 6, respectively. These algorithms describe how records from two hospitals can be joined using a buffering approach that minimizes the number of passes over the inner relation. The set of records of the first and second hospital is depicted as DB_1 and DB_2, respectively.

Algorithm 5 summarizes the execution of the join operation on the server side. For more details on sovereign joins please see [5]. For all records in DB_1, the server prepares a message request MSG, including the record ID and the encrypted record. Using a blocking function called "$send_MSG$" provided by the server, the server sends MSG to the SCP and waits to receive an acknowledgement from the SCP. Using a counter variable $bufferCtr$, the server tracks the

number of records buffered in the SCP. If the number of DB_1 records transferred to the SCP exceeds the buffer size, the server starts sending the encrypted records of DB_2 together with their record IDs. For all records in DB_2, an acknowledgement is received from the SCP including the knowledge of whether there is a match. After making a full pass over DB_2 records, $bufferCtr$ is reset to continue with buffering new records of DB_1.

1 **Input:** DB_1, DB_2
2 **Output:** Matching records

```
 1: bufferCtr ← 0
 2: for rec_1 in DB_1 do
 3:    MSG ← "DB_1", recordID_1, rec_1 {Prepare message}
 4:    send_MSG(MSG) to SCP {send message}
 5:    get ACK_1 from SCP {get acknowledgement}
 6:    bufferCtr ← bufferCtr + 1 {update buffer counter}
 7:    if bufferCtr = BUFFER_SIZE then
 8:       for rec_2 in DB_2 do
 9:          MSG ← "DB_2", recordID_2, rec_2
10:          send_MSG(MSG) to SCP
11:          get ACK_2 from SCP
12:          outputMatchedDB1Id(ACK_2) {output match result}
13:       end for
14:       bufferCtr ← 0 {reset buffer counter}
15:    end if
16: end for
```

Algorithm 5: Joining patient records with Sovereign Join: Server side

Algorithm 6 summarizes the sub-steps of the join operation executed on the SCP side. The application running on the SCP waits for new messages to be received in a blocking state. Once a new message is received the database identifier of the message is checked using the function get_DB_ID. If the message includes a record of DB_1, the record is decrypted and buffered in the internal memory of the SCP. For decryption, the API provided by the SCP is used. Also, the record ID information is buffered in an array called $recordDB1$. The $get_recordID$ function is used to extract the record identifier from the message. If the message contains a record from DB_2, the record is decrypted first and stored in another buffer. The decrypted record is then compared

with all other records of DB_1 iteratively using a preferred distance metric. If there is a match, an acknowledgement message including the matching identifier information is sent to the server.

```
1 Input: Request messages (MSG)
2 Output: Matching results
 1: bufferCtr ← 0
 2: while TRUE do
 3:    wait for new message MSG in blocked state
 4:    if get_DB_ID(MSG) = "DB1" then
 5:       dataDB1[bufferCtr] ← decrypt(get_record(MSG)) {buffer the record}
 6:       recordDB1[bufferCtr] ← get_recordID(MSG)
 7:       bufferCtr ← bufferCtr + 1 {update buffer counter}
 8:    else
 9:       if get_DB_ID(MSG) = "DB2" then
10:          dataDB2 ← decrypt(get_record(MSG))
11:          recordDB2 ← get_recordID(MSG)
12:          for i = 0 to BUFFER_SIZE do
13:             if distanceMetric(dataDB1[i], dataDB2) < 1 then
14:                matchedDB1Id ← recordDB1[j] {send match result}
15:             end if
16:             matchedDB1Id ← "No match"
17:             send_ACK2(matchedDB1Id) {no match found}
18:          end for
19:          bufferCtr ← 0 {reset buffer counter}
20:       end if
21:    end if
22: end while
```

Algorithm 6: Joining patient records with Sovereign Join: SCP side:

6.5 Experiments and Results

We developed a prototype implementation of the framework introduced in Figure 6.2 to measure the performance of the data integration and query processing phases. The framework was implemented using the C programming language on a real server with an SCP. The server was an IBM x3500 server with 32GB of main memory and an 8-core Intel Xeon CPU E5420 @ 2.50GHz processors. The server ran a 32-bit SLES 10.2 (Linux kernel 2.6.16.60) operating system. The SCP

was an IBM 4764 PCI-X Cryptographic Coprocessor (See Section 2.7). For the cryptographic operations, we used AES in CTR mode as the encryption algorithm, with a key size of 128-bits.

In the first part of the experiments we present the performance for the data integration phase. In the second part we discuss the performance of processing the count queries in the proposed framework. We note that the hardware specification of the server has no direct impact on the benchmarks performed because all cryptographic operations and query evaluation steps are performed on the SCP. The server is only used to submit the queries to the SCP and retrieve the output. We used synthetic binary datasets ranging in size from 1000 to 30000 records and queries that involve between 10 and 50 binary attributes. Note that the performance of the experiments does not depend on whether we use a real dataset or synthetic dataset since the SNP values does not have any impact on the performance of the proposed framework.

6.5.1 Join Operation

We implemented the sovereign join algorithm described in Section 6.4 to evaluate the performance of the data integration phase. In Figure 6.3, the execution time of the join operation is depicted for a buffer size ranging from 1 to 3000 records[4]. The results indicate that buffering the records in the SCP yields significant performance gains. Specifically, for 1000 records a buffer size of 1000 records provide 5 times improvement compared to a protocol in which no buffering is used.

The experiments show that the total execution time spent for the join operations is relatively high. However, it should be noted that the data integration phase is a one time step executed before the query execution phase. In the real applications, the system administrators could allow this step to run overnight or even a week to finish. Once the data integration phase is completed,

[4]See Figure 6.4 for the execution time of 10000 records.

the researchers can execute their queries in a much faster speed as we discuss the query processing performance experiments in the next section.

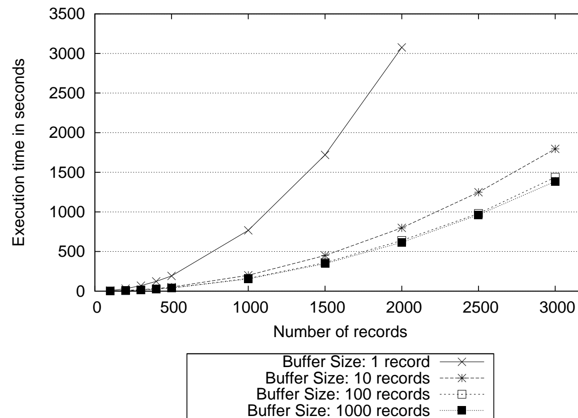

Figure 6.3. Execution time of the join operation for various buffer sizes

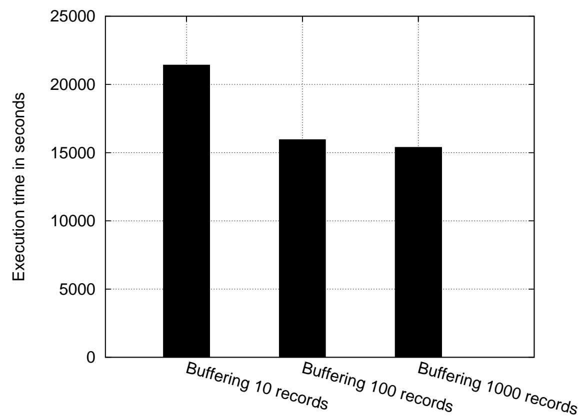

Figure 6.4. Execution time of the join operation for various buffer sizes

6.5.2 Selection Operation

In this section we evaluate the performance of executing secure count queries on the biomedical data. Figure 6.5 shows the query processing time for the count queries on various datasets with different query sizes. The *x*-axis shows the number of predicates defined in the queries and the *y*-axis represents the total execution time of the protocol in seconds. We observe that the execution

time increases linearly in query size. Moreover, the rate of change between the execution time and the query size increases as the number of records in the dataset grows. These results indicate that processing 10000 records could be handled in under one minute in our framework. Even for 30000 records, the protocol could process a query with 10 predicates in approximately two minutes.

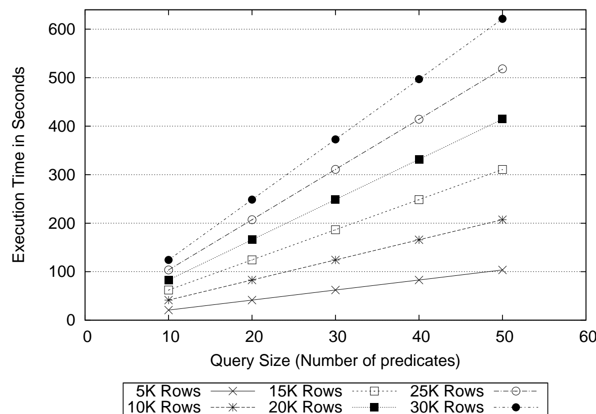

Figure 6.5. Execution time for count queries on various datasets with different query sizes (SCP based protocol)

In Figure 6.6 we demonstrate the improvement in runtime of our protocol compared to the secure count algorithm of the expensive encryption method used in the multiple third party framework [50]. Recognize that for a query size of 10 predicates, our protocol runs about 80 times faster. We observe a decline in the improvement rate as the number predicates increases. Nevertheless, the difference is more than an order of magnitude for 40 predicates. One interesting point to note is that the improvement rate is independent of the dataset size, which implies that our approach scales to large datasets.

6.6 Discussion

In this section, we showed that secure coprocessors can be applied to efficiently share, store, and query structured biomedical data, such as genomic sequences. Using a real implementation, we

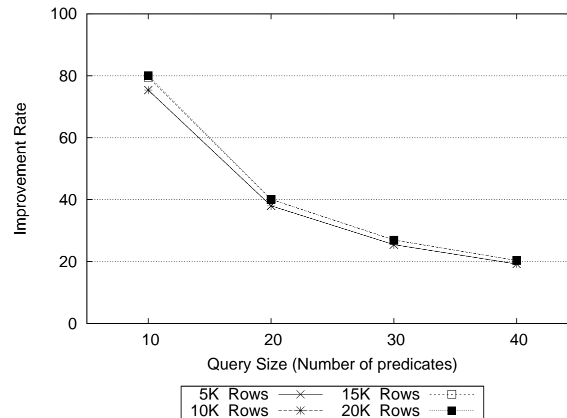

Figure 6.6. Improvement rate compared to the multiple third party protocol in [50]

empirically demonstrated that our framework is almost two orders of magnitude more efficient than alternative solutions based on public key cryptography.

Beyond efficiency gains, another advantage of our framework is that the SCP can facilitate the generation of secure audit logs. Consider, when a query request is sent to the cryptographic device, a new log entry could be created by the device. This entry would include the query information and a timestamp. The most recently generated log ID would always be stored within the device. Once created, the log entry would be output to the server. On the server, these logs could be stored in append-only mode and force-written to write-only disk. The creation of secure logs could be utilized to satisfy some of the more stringent recommendations for the secure management of patient information as specified in regulations such as the HIPAA Security Rule [27].

Though the data remains encrypted at all times within the framework (except in the tamper-resistent SCP), the results of scientists' queries themselves can violate privacy requirements. For instance, if the answer to a researcher's query is such that there is only one record with a DNA sequence *"AATCAATGAA"* and a positive diagnosis of *juvenile Alzheimer's*, then the researcher has uniquely pinpointed an individual's record. In the event that uniqueness is a violation, it

will be necessary to ensure that query results for a researcher do not permit the triangulation of an individual's record. This can be achieved through a process known as *query restriction* and its application is necessary to ensure that our framework achieves identity protection. For instance, it is possible to implement a query restriction manager that refuses to report query results if it is below some certain threshold. This topic has been studied extensively in the database security community [3]. We believe that our framework can be extended to run any compact query restriction policy manager within the SCP and intend to design such solutions in future work.

6.7 Conclusion

In this section, we presented a secure framework by which person-specific biomedical data, such as genomic sequences, can be joined and queried using cryptographic hardware. In contrast to formal privacy models for biomedical data that "perturb" or "generalize" records, our methods ensure that data is shared in its most specific state. Beyond a theoretical basis, we experimentally validated that the architecture is much more efficient when compared to solutions based on expensive public key encryption protocols. We also reduced the trust requirements from multiple third parties to a single third party, which may be more viable for real world applications.

CHAPTER 7

CONCLUSION

This thesis presents efficient solutions for processing *encrypted sensitive data* under different threat models. For the conventional database management systems where semi trusted threat model is used we compare the performance of different block cipher modes, under different encryption granularity and disk access patterns. Based on our experiments, we suggest a CTR based approach for encrypting data in DBMSs. In addition to that, we propose a page level encryption method by utilizing the selective decryption feature of CTR mode. As a future work, we plan to implement our proposed approach using cryptographic accelerators in a distributed database environment.

We also propose a vertical partitioning approach to prevent unnecessary cryptographic operations over non-sensitive attributes during query processing. Our experiments prove that vertical partitioning prevents the CPU from becoming the bottleneck during query execution. We propose an optimal partitioning technique for single relational query workloads. For multi relational query workloads, we show that the partitioning problem is an instance of binary integer programming and proposed a heuristic. We show that our proposed heuristic solution has linear complexity and achieves 0.4 % error rate in comparison to the optimum partitioning strategy. Overall, heuristic partitioning improves query execution time by 94.5 % on the average. As a future work, we are planning to extend our study to consider column stored databases. In addition, we will explore the horizontal partitioning aspect of the problem.

In Section 5, we consider a fully untrusted attack model for a relational database where a malware can monitor both the content of the memory and the disk for a certain amount of time. We quantify the disclosure risk of encrypted data in a relational DBMS for main memory-based attacks and propose modifications to the standard query processing mechanism to minimize such risks. We demonstrate how to use secure coprocessors to reduce the risk of such attacks. Our work in this thesis just scratches the surface of this important new direction of potential research that could lead to new studies. As a future work, we will analyze the impact of various query rewriting strategies and heuristics (e.g., pushing selections) on disclosure risk. Also, we will explore how to efficiently store and analyze the audit log for effective forensic analysis for reducing disclosure risk. Furthermore, we plan to extend our framework to consider active memory attacks where the attacker can modify the contents of the memory as well.

We also consider a fully untrusted attack model for a client server architecture which can be used for processing biomedical genomic data. We introduce a framework that removes the need for multiple third parties by collocating services to store and to process sensitive biomedical data through the integration of cryptographic hardware. Within this framework, we define a secure protocol to process genomic data and perform a series of experiments to demonstrate that such an approach can be run in an efficient manner for typical biomedical investigations. This study proves that the secure cryptographic hardware could be used in many biomedical information processing applications where privacy is considered. We recognize this framework does not explicitly address privacy violations that can be directly extracted from the query results, but are confident this issue could be resolved by leveraging the computation capabilities of the secure hardware. Also, we are planning to extend our studies with some new experiments to demonstrate that GWAS queries could be executed efficiently within the tamper proof hardware.

BIBLIOGRAPHY

[1] IBM 4764 PCI-X Cryptographic Coprocessor. `http://www-03.ibm.com/security/cryptocards/pcixcc/library.shtml`.

[2] Recommendation for block cipher modes of operation methods and techniques. Technical Report NIST Special Publication 800-38A, National Institute of Standards and Technology, 2001. `http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf`.

[3] N. R. Adam and J. C. Worthmann. Security-control methods for statistical databases: a comparative study. *ACM Comput. Surv.*, 21(4):515–556, 1989.

[4] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. X. 0002. Two can keep a secret: A distributed architecture for secure database services. In *CIDR*, pages 186–199, 2005.

[5] R. Agrawal, D. Asonov, M. Kantarcioglu, and Y. Li. Sovereign joins. In 22nd *IEEE International Conference on Data Engineering*, page 26, 2006.

[6] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order-preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, Paris, France, June 13-18 2004.

[7] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In 27th *International Conference on Very Large Data Bases*, pages 169–180. Morgan Kaufmann Publishers Inc., 2001.

[8] D. Asonov and J. Freytag. Almost optimal private information retrieval. In 2nd *International Workshop on Privacy Enhancing Technologies*, pages 209–223. Springer-Verlag, 2002.

[9] M. J. Atallah, Y. Cho, and A. Kundu. Efficient data authentication in an environment of untrusted third-party distributors. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 696–704, Washington, DC, USA, 2008. IEEE Computer Society.

[10] R. Bayer and J. K. Metzger. On the encipherment of search trees and random access files. *ACM Trans. Database Syst.*, 1(1):37–52, 1976. `http://doi.acm.org/10.1145/320434.320445`.

[11] M. Bellare and P. Rogaway. Introduction to modern cryptography. In *UCSD CSE 207 Course Notes*, page 207, 2005.

[12] K. Benitez and B. Malin. Evaluating re-identification risks with respect to the HIPAA Privacy Rule. *J. Am. Medical Informatics Assoc.*, 17(2):169–177, 2010.

[13] M. Blaze. A cryptographic file system for unix. In *CCS '93: Proceedings of the 1st ACM conference on Computer and communications security*, pages 9–16, New York, NY, USA, 1993. ACM.

[14] L. Burnett, K. Barlow-Stewart, A. Proos, and H. Aizenberg. The "GeneTrustee" a universal identification system that ensures privacy and confidentiality for human genetic databases. *Journal of Law and Medicine*, 10(4):506–513, May 2003.

[15] M. Canim and M. Kantarcioglu. Design and analysis of querying encrypted data in relational databases. In S. Barker and G.-J. Ahn, editors, 21st *IFIP WG 11.3 Working Conference on Data & Applications Security*, volume 4602 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2007.

[16] M. Canim, M. Kantarcioglu, B. Hore, and S. Mehrotra. Building disclosure risk aware query optimizers for relational databases. *PVLDB*, 3(1):13–24, 2010.

[17] M. Canim, M. Kantarcioglu, and A. Inan. Query optimization in encrypted relational databases by vertical schema partitioning. In *Secure Data Management*, pages 1–16, 2009.

[18] M. Canim, M. Kantarcioglu, and A. Inan. Query optimization in encrypted relational databases by vertical schema partitioning. In *Secure Data Management*, pages 1–16, 2009.

[19] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2008. ACM.

[20] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: reducing data lifetime through secure deallocation. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 22–22, Berkeley, CA, USA, 2005. USENIX Association.

[21] M. Cox, R. Engelschall, S. Henson, and B. Laurie. The OpenSSL Project. `http://www.openssl.org/`.

[22] A. Czeskis, D. J. S. Hilaire, K. Koscher, S. D. Gribble, T. Kohno, and B. Schneier. Defeating encrypted and deniable file systems: Truecrypt v5.1a and the case of the tattling os and applications. In *HOTSEC'08: Proceedings of the 3rd conference on Hot topics in security*, pages 1–7, Berkeley, CA, USA, 2008. USENIX Association.

[23] E. Damiani, S. D. C. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational dbmss. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 93–102. ACM Press, 2003. `http://doi.acm.org/10.1145/948109.948124`.

[24] G. de Moor, B. Claerhout, and F. de Meyer. Privacy enhancing techniques - the key to secure communication and management of clinical and genomic data. *Methods of Information in Medicine*, 42(2):148–153, 2003.

[25] J. Denny, M. Ritchie, M. Basford, J. Pulley, L. Bastarache, K. Brown-Gentry, D. Wang, D. Masys, D. Roden, and D. Crawford. PheWAS: demonstrating the feasibility of a phenome-wide scan to discover gene-disease associations. *Bioinformatics*, 26(9):1205–1210, 2010.

[26] Dept. of Health and Human Services. Standard for privacy of individually identifiable health information. *Federal Register*, 67(157):53181–53273, Aug. 14 2002.

[27] Dept. of Health and Human Services. Standards for protection of electronic health information. *Federal Register*, 45(164), Feb. 20 2003.

[28] P. T. Devanbu, M. Gertz, C. U. Martel, and S. G. Stubblebine. Authentic third-party data publication. In *DBSec*, pages 101–112, 2000.

[29] Y. Elovici, E. Shmueli, R. Waisenberg, and E. Gudes. A structure preserving database encryption scheme. In *Workshop on Secure Data Management in a Connected World (SDM'04)*, Aug. 30 2004. `http://www.extra.research.philips.com/sdm-workshop/RonenSDM.pdf`.

[30] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[31] D. Gaudet, S. Arsenault, C. Belanger, T. Hudson, P. Perron, M. Bernard, and P. Hamet. Procedure to protect confidentiality of familial data in community genetics and genomic research. *Clinical Genetics*, 55:259–264, 1999.

[32] S. Gaudin. Security breaches cost $90 to $305 per lost record. `http://www.informationweek.com/news/security/showArticle.jhtml?articleID=199000222`.

[33] T. Ge and S. Zdonik. Light-weight, runtime verification of query sources. In *ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 30–41, Washington, DC, USA, 2009. IEEE Computer Society.

[34] K. Goldman and E. Valdez. Matchbox: Secure data sharing. *IEEE Internet Computing*, 8(6):18–24, 2004.

[35] E. Green, M. Guyer, and National Human Genome Research Institute. Charting a course for genomic medicine from base pairs to bedside. *Nature Genetics*, 470:204–213, 2011.

[36] J. Gulcher, K. Kristjansson, H. Gudbjartsson, and K. Stefansson. Protection of privacy by third-party encryption in genetic research in iceland. *European Journal of Human Genetics*, 8(10):739–42, 2000.

[37] D. Gurwitz, J. Lunshof, and R. Altman. A call for the creation of personalized medicine databases. *Nature Reviews Drug Discovery*, 5(1):23–26, 2006.

[38] S. Haber and W. S. Stornetta. How to time-stamp a digital document. *J. Cryptology*, 3(2):99–111, 1991.

[39] H. Hacigumus, B. R. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the da tabase-service-provider model. In *Proceedings of the 2002 ACM SIGMOD In ternational Conference on Management of Data*, pages 216–227, Madison, Wisconsin, June 4-6 2002. `http://doi.acm.org/10.1145/564691.564717`.

[40] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, 2009.

[41] K. Hara, K. Ohe, T. Kadowaki, N. Kato, Y. Imai, K. Tokunaga, R. Nagai, and M. Omata. Establishment of a method of anonymization of dna samples in genetic research. *Journal of Human Genetics*, 48(6):327–330, 2003.

[42] T. Hardjono and J. Seberry. Search key substitution in the encipherment of b-trees. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 50–58. Morgan Kaufmann, 1990.

[43] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In 30th *International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 2004.

[44] IBM. IBM 4764 cryptographic coprocessor, 2004.

[45] IBM. Table Space Design, 2006. `http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=/com.ibm.db2.udb.admin.doc/doc/c0004935.htm`.

[46] A. Iliev and S. W. Smith. Private information storage with logarithm-space secure hardware. In *International Information Security Workshops*, pages 199–214, 2004.

[47] A. Iliev and S. W. Smith. Small, stupid, and scalable: secure computing with faerieplay. In *Proceedings of the fifth ACM workshop on Scalable trusted computing*, STC '10, pages 41–52, New York, NY, USA, 2010. ACM.

[48] Innobase. InnoDB, Transactional Storage Engine. `http://www.innodb.com/`.

[49] B. Iyer, S. Mehrotra, E. Mykletun, G. Tsudik, and Y. Wu. A framework for efficient storage security in RDBMS. In 9th *International Conference on Extending Database Technology*, pages 627–628, 2004.

[50] M. Kantarcioglu, W. Jiang, Y. Liu, and B. Malin. A cryptographic approach to securely share and query genomic sequences. *IEEE Transactions on Information Technology in Biomedicine*, 12.

[51] S. Krishnan and F. Monrose. Timecapsule: secure recording of accesses to a protected datastore. In *VMSec '09: Proceedings of the 1st ACM workshop on Virtual machine security*, pages 23–32, New York, NY, USA, 2009. ACM.

[52] A. Kundu and E. Bertino. Structural signatures for tree data structures. *Proc. VLDB Endow.*, 1(1):138–150, 2008.

[53] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 121–132, New York, NY, USA, 2006. ACM.

[54] H. Lipmaa. Idea: A cipher for muldimedia architectures? In *Stafford Tavares and Henk Meijer, editors, Selected Areas in Cryptography '98*. Springer-Verlag, 1998.

[55] H. Lipmaa, P. Rogaway, and D. Wagner. Ctr-mode encryption. In *NIST, Computer Security Resource Center, First Modes of Operation Workshop*, 2000. `http://csrc.nist.gov/CryptoToolkit/modes/workshop1/papers/lipmaa-ctr.pdf`.

[56] M. Khoury, et al. Comparative effectiveness research and genomic medicine: an evolving partnership for 21st century medicine. *Genetics in Medicine*, 11(10):707–711, 2009.

[57] M. Mailman, et al. The NCBI dbGaP database of genotypes and phenotypes. *Nature Genetics*, 39(10):1181–1186, 2007.

[58] C. Maartmann-Moe, S. E. Thorkildsen, and A. rnes. The persistence of memory: Forensic identification and extraction of cryptographic keys. *Digital Investigation*, 6(Supplement 1):S132 – S140, 2009. The Proceedings of the Ninth Annual DFRWS Conference.

[59] B. Malin. An evaluation of the current state of genomic data privacy protection technology and a roadmap for the future. *J. Am. Medical Informatics Assoc.*, 12(1):28–34, 2005.

[60] T. Manolio. Collaborative genome-wide association studies of diverse diseases: programs of the NHGRI's office of population genomics. *Pharmacogenomics*, 10(2):235–241, 2009.

[61] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[62] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. *SIGOPS Oper. Syst. Rev.*, 33(5):124–139, 1999.

[63] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 108–117, New York, NY, USA, 2002. ACM.

[64] M. C. Mont, S. Pearson, and P. Bramhall. Towards accountable management of identity and privacy: Sticky policies and enforceable tracing services. In 14th *International Workshop on Database and Expert Systems Applications*, 2003.

[65] M. Narasimha and G. Tsudik. Authentication of outsourced databases using signature aggregation and chaining. In *DASFAA*, pages 420–436, 2006.

[66] National Institute of Standards and Technology. Advanced encryption standard (aes). Technical Report NIST Special Publication FIPS-197, National Institute of Standards and Technology, 2001. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[67] National Institutes of Health. Policy for sharing of data obtained in nih supported or conducted genome-wide association studies (gwas), August 2007.

[68] P. Ngatchou, A. Zarei, and M. El-Sharkawi. Pareto multi objective optimization. In *Intelligent Systems Application to Power Systems, 2005. Proceedings of the 13th International Conference on*, pages 84–91, Nov. 2005.

[69] Data encryption standard (des). Technical Report FIPS PUB 46-2, National Institutes of Standards and Technology, Jan. 22 1988.

[70] P. Burton, et al. Size matters: just how big is big?: Quantifying realistic sample size requirements for human genome epidemiology. *International Journal of Epidemiology*, 38(1):263–267, 2009.

[71] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan. Verifying completeness of relational query results in data publishing. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 407–418, New York, NY, USA, 2005. ACM.

[72] H. Pang and K.-L. Tan. Authenticating query results in edge computing. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 560, Washington, DC, USA, 2004. IEEE Computer Society.

[73] K. E. Pavlou and R. T. Snodgrass. Forensic analysis of database tampering. *ACM Trans. Database Syst.*, 33(4):1–47, 2008.

[74] S. Peace. California database security breach notification act, Sept. 2002. `http://info.sen.ca.gov/pub/01-02/bill/sen/sb_1351-1400/sb_1386_bill_20020926_chaptered.html`.

[75] T. Peakman and P. Elliott. The UK biobank sample handling and storage validation studies. *International Journal of Epidemiology*, 37 Suppl 1:i2–i6, 2008.

[76] B. Pinkas and T. Reinman. Oblivious ram revisited. In *Proceedings of the 30th annual conference on Advances in cryptology*, CRYPTO'10, pages 502–519, Berlin, Heidelberg, 2010. Springer-Verlag.

[77] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2003.

[78] R. Ramamurthy and D. J. DeWitt. Buffer-pool aware query optimization. In *CIDR*, pages 250–261, 2005.

[79] S. Purcell, et al. PLINK: a tool set for whole-genome association and population-based linkage analyses. *American Journal of Human Genetics*, 81(3):559–575, 2007.

[80] B. Schneier. The blowfish encryption algorithm. *Dr. Dobb's Journal*, pages 38–40, Apr. 1994.

[81] B. Schneier. *Applied Cyptogography: Protocols, Algorithms and Source Code in C*. Wiley, 1996.

[82] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Trans. Inf. Syst. Secur.*, 2(2):159–176, 1999.

[83] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[84] E. Shi, J. Bethencourt, H. T.-H. Chan, D. X. Song, and A. Perrig. Multi-dimensional range query over encrypted data. In *IEEE Symposium on Security and Privacy*, pages 350–364, 2007.

[85] R. Sion. Query execution assurance for outsourced databases. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 601–612. VLDB Endowment, 2005.

[86] R. T. Snodgrass, S. S. Yao, and C. Collberg. Tamper detection in audit logs. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 504–515. VLDB Endowment, 2004.

[87] P. Stahlberg, G. Miklau, and B. N. Levine. Threats to privacy in the forensic analysis of database systems. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 91–102, New York, NY, USA, 2007. ACM.

[88] V. B. R. Team. 2009 data breach investigations supplemental report, anatomy of a data breach. `www.verizonbusiness.com/go/09SuppDBIR`.

[89] TPC. TPC-H, Decision Support Benchmark. `http://www.tpc.org/tpch/`.

[90] X. Zhang, et al. Secure coprocessor-based intrusion detection. In *ACM SIGOPS European Workshop*, Sept. 2002.

[91] M. Xie, H. Wang, J. Yin, and X. Meng. Integrity auditing of outsourced data. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 782–793. VLDB Endowment, 2007.

[92] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated join processing in outsourced databases. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 5–18, New York, NY, USA, 2009. ACM.

[93] Y. Yang, S. Papadopoulos, D. Papadias, and G. Kollios. Spatial outsourcing for location-based services. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 1082–1091, Washington, DC, USA, 2008. IEEE Computer Society.

[94] S. B. Yao. Approximating block accesses in database organizations. *Commun. ACM*, 20(4):260–261, 1977.

VITA

Mustafa Canim received the Ph.D. degree from the University of Texas at Dallas in 2011 under the supervision of Dr. Murat Kantarcioglu. He earned his M.S. degree in Computer Science from the University of Texas at Dallas in 2007 and his B.S. degree in Computer Science from Bilkent University, Ankara, Turkey in 2005. Before joining the Ph.D. program, he worked as a senior researcher with the database research group at TUBITAK, a prestigious research institute in Turkey. His research interests lie in the area of architecture conscious data management with a focus on privacy and performance issues. He interned at the Database Research Group of IBM T.J. Watson in 2008 and 2009 in the “Solid State Sensitive Systems” project. His work at IBM has led to several patent applications and top level conference papers. Mr. Canim also worked at the Engineering Tools Department at Google as an intern and gained significant experience on cloud computing platforms. The product he developed at Google is now running on Google data centers and used by Google engineers.