STEGANOGRAPHIC DATABASE MANAGEMENT SYSTEM

MA XI

(M.Sc. in Computer Science, HUST, China)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER CIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2005

Acknowledgement

I am deeply indebted to my supervisors, Dr. Pang Hwee Hwa and Prof. Tan Kian Lee, for their guidance, help, insight and encouragement on my study and life throughout the course of my master program.

I owe many thanks to my parents, for their continuous moral support. I thank my girlfriend for her enormous patience during the course of my work. I could not have finished it without their help.

I would like to thank the numerous referees who have reviewed parts of this work prior to publication and whose valuable comments have contributed to the clarification of many of the ideas presented in this thesis. Of these, Mr. Yang yanjang, Mr. Yang Zixiang, Mr. Niu Zhengyu, Mr. Zhou Xuan deserve special thanks. Their moral support and suggestions were of great value.

TABLE OF CONTENTS

List of Figures

List of Tables

Summary

The database community is witnessing the emergence of two recent trends set on a collision course. On the one hand, the outsourcing of data management has become increasingly attractive for many organizations with the advances in the network technologies. On the other hand, escalating concerns about data privacy, recent governmental legislation, as well as high-profile instances of database theft, have sparked keen interest in enabling secure data storage and access. The two trends are indirectly conflict with each other. A client using an outsourced database service is required to trust the service provider with potentially sensitive data, leaving the door open for damaging leaks of private information.

A steganographic database management system (StegDB) could support steganography in data storage and query. In particular, it grants access to a protected database partition only if the correct access key is supplied; without it, an adversary could get no information about whether the protected partition ever exists. The protected partitions are hidden not only logically but physically, which ensures that a system intruder cannot detect the existence of those sensitive data even if he understands the hardware and software completely.

In this thesis, we present two different StegDB designs: 1) model 1 in which only the data storage is outsourced, and 2) model 2 in which the database server entirely expose to untrusted environment. For model 1, we propose two data mining tools to mine informational patterns from access sequence on data storage which might be intercepted by an adversary. To counter this security threat, we then provide two sequence transforming schemes which are aimed at diminishing access patterns in sequences. For model 2, we propose a framework based on secure coprocessor. To meet the security requirement of a StegDB under this framework, we then give a query decomposition algorithm and hide the interplay between operations in open processors and access activities on private storage. This work is pioneering in the sense that it fills the gap between the high level security requirement and ubiquitous computing environment.

Chapter 1

Introduction

Ubiquitous computing entails the permeation of computing in every facet of our lives, be it work, personal or leisure, to a point where users take it for granted and stop to notice it. The data that underlie the ubiquitous services have to be persistent and available anywhere-anytime. This means that the data must migrate from devices that are local to individual computers, to shared network storage, or even to outsourced storage systems that are hosted and managed by external storage service providers. A development that would facilitate this migration is the shared storage area network (SAN) that is accessible over TCP/IP network through iSCSI, a block-level access protocol. Another supporting development is the recent interest in building reliable logical storage volumes on unreliable nodes in a peer-to-peer platform (e.g. [1]). Following the outsourcing of data storage, the outsourcing of data management has become increasingly attractive for many organizations with the advances in the network technologies. The use of an external database service promises reliable data storage at a low cost, eliminating the need for expensive in-house data-management infrastructure; the high availability of outsourced database service is attractive in supporting the ubiquitous computing environment. In the next five years, we expect that clients would query and update their precious data anytime through portable wireless-enabled devices.

On the other hand, escalating concerns about data privacy, recent governmental legislation, as well as high-profile instances of database theft, have sparked keen interest in enabling secure data storage. With the development of electronic commerce on the Internet, among others, privacy is a must. In some stricter situations, operations on the sensitive data need to be as secret as their contents. What if the CEO of a companies suddenly begin frequent query on the encrypted salaries of a particular group of employees? In the business world, this might be enough to very a rumor of some personnel changes. In ideal environments, unauthorized database users, including the database administrator, could not even detect the existence of secrets.

The two trends described above indirectly conflict with each other. While shared network storage provides the availability needed for ubiquitous computing, it introduces new challenges in data security - Since data reside on open networks, there are several avenues from which an attacker could attempt to circumvent conventional access control and encryption mechanisms. However, a client using a database service is however required to trust the service provider with potentially sensitive data, leaving the door open for damaging leaks of private information. Even the sensitive data is encrypted during entire processing procedure, which is widely researched currently (e.g. [2, 3]), the existence of sensitive data might be regarded as some kind of information leakage, in high security environment. Steganography, the art of hiding information in ways that prevent its detection, offers a way to achieve the desired protection. It is a better defense than cryptography alone -While cryptography scrambles a message so it cannot be understood, steganography goes a step further in making the ciphertext invisible to unauthorized users. A database service providing reliable storage and efficient query execution without revealing even the existence of potential sensitive data is particularly useful in this outsourced environment. Such a service also helps the service provider by limiting their liability in case of break-ins into their system.

1.1 The StegDB Problem

Current commercial database systems already have available access control sub-system which allows users to specify access policies for data objects. For example, a database user can be granted or revoked to select, update, and delete privileges. These access control mechanisms can also be extended or complemented by data encryption. Encrypting some data in database can prevent database administrator from visiting the content of data. But in both environments, an unauthorized observer can still establish it existence and coerce the owner into unlocking them.

Therefore, there is a need for a steganographic database management system (StegDB) which could provide steganography in data storage and query. In particular, it should grant access to a protected database partition only if the correct password or access key is supplied; without it, an adversary could get no information about whether the protected partition ever exists. The protected partition should be hidden not only logically but physically, which ensures that a system intruder cannot detect the existence of those sensitive data even if he understands the hardware and software of the database system completely, and is able to scour through its data structures and the content on the raw disks. It should also protect the query on those hidden data from disclosure. For legitimate users, the data in those hidden protected partitions would be accessed nearly transparently, like in a conventional DBMS. A StegDB could also prevent an attacker from verifying whether user acting under compulsion actually discloses all the data. However, having a data service backend that is untrusted, or under a different administrative domain, requires a re-examination of the security architecture of the DBMS. In the untrusted environment, adversaries could launch their attacks in several places. How to provide steganographic feature in database management system poses a challenge.

1.2 Contribution

In this thesis, we present two different StegDB design, i.e. model 1 and 2. In model 1, only the data storage is outsourced; in model 2, the database server is entirely exposed to untrusted environment. For model 1, we propose two data mining tools to mine informational patterns from access sequence on data storage which might be intercepted by an adversary. To spot the specious patterns quickly, the mining tools can also support various constraints. To counter the security threat exposed by the mining tools, we then provide two sequence transforming schemes which are aimed at diminishing access patterns in sequences. The experiment shows that, with less than 40% overhead, patterns in access sequence could be masked to the level that most patterns never repeat twice. For model 2, we propose a framework based on secure coprocessor. In order to meet the security requirement under this framework, we provide a query decomposition algorithm and hide the interplay between operations in open processors and access activities on private storage. The work on model 2 guarantees that a StegDB server which is outsourced in a hostile environment will not sacrifice its security standard and still provide stenography feature to users.

This work is pioneering in the sense that it fills the gap between the high level security requirement and ubiquitous computing environment. It's the first step towards outsourcing multi-level secure database.

1.3 Thesis Outline

The remaining chapters of this thesis are organized as follows:

Chapter 2 first gives the problem statement in detail and the definition of plausibility. It then present three model of StegDB (i.e. model 0, model 1 and 2), the latter two will be further explored in the following chapters. Following that, it reviews related work on various security problems in outsourced database system.

Chapter 3 proposes an attack method—mining the frequent patterns in access sequences of outsourced storage. Those patterns might disclose the existence of hidden data. It presents two algorithms for mining the frequent episodes – Episode Prefix Tree (EPT) and Position Pairs Set (PPS). In particular, EPT mines frequent episodes by growing a prefix tree. In contrast, PPS adopts a divide-and-conquer strategy to avoid the iterative full sequence scans that occurs in EPT. It splits the sequence into slices according to the frequent episodes' prefix, and then grows each prefix by exploring only those slices. Both algorithms allow a systematic way to push various constraints into the mining process. Performance study shows that the proposed algorithms run considerable faster than MINEPI[4] which is the state-of-the-art algorithm for mining frequent episodes.

Chapter 4 proposes two countermeasures to mitigate the risk of attacks initiated through analyzing the shared storage server's activity for page patterns. The first countermeasure relocates data pages according to which page sequences they are in. The second countermeasure enhances the first by randomly prefetching pages from predicted page sequences. To evaluate our schemes objectively, we also propose a quantity based on entropy from information theory. We have implemented the two countermeasures in MySQL, and experiment results demonstrate their effectiveness and practicality.

Chapter 5 gives a framework of StegDB in model 2, where the database engine is moved to the untrusted environment. Through the use of secure coprocessors, the database system is divided into two parts: one processes shared data and the other processes hidden data. It analyses the access problem on private storage and proposes a query decomposition algorithm which separates a client query on shared and hidden data separately. It then analyses the intersection size problem which might leak the existence information of sensitive data and give a two-zone scheme to counter this security threat.

Finally, Chapter 6 presents the conclusion and suggestions for future works.

Chapter 2

Background and Related Work

In this chapter, we first analyze the problem and then give three models of StegDB. The latter part of this chapter review related work on various security problems in StegDB environment.

2.1 Problem Statement

2.1.1 Hidden Partition

Let us first introduce the relevant definitions below. A relational schema is represented as $RS(Attr)$ where $Attr$ is a set of attributes $\{A_1, A_2, ...A_n\}$. Data may be hidden or open. The data in hidden partitions which is denoted as h, are called *hidden data*; the data outside of any hidden partitions are called *shared data*. The whole set of shared table containing shared data is denoted as SS.

Each hidden partition is associated with a distinct Hidden Partition Key (HPK), only HPK key holders are given access to the data in the corresponding hidden partition. Without a HKP, an unauthorized user could not access the data in a hidden partition. Each hidden partition is stored as a hidden file in the StegFS system [5]. A hidden partition has two states: 1) online which means it is activated and accessible to its HPK holder; 2) offline which means it is currently not activated by any users.

There are three types of hidden partitions: 1) Horizontal partition: $\exists t \in SS, t.Attr = h.Attr$, e.g., there is a shared table that has the same attributed set; 2) Vertical partition: $\exists t \in SS, h.Attr \wedge t.Attr \neq \Phi$, e.g., there is a shared table that has common subset of columns with h; 3) Standalone partition which does not belong to the above categories. Tables 2.1 and 2.2 show an example of a horizontal partition and its corresponding shared table.

TID	Duration	Objective
T001	001	Exploration
T002	002	Exploration
T003	002	Spying

Table 2.1 Shared task table

TID	Duration	Objective
T054	001	Spying
T058	007	Assassinatio

Table 2.2 Hidden task partition.

2.2 Security Definition of StegDB

In StegDB, protected sensitive data can be hidden securely so that without the corresponding access keys an attacker would not be able to deduce their existence or at least not be able to estimate the amount of hidden data of any particular user, even if he has the administrator privilege, gaining full access to physical storage media and understanding the software completely. In other words, StegDB prevents an attacker from observing the existence of sensitive data and verifying whether the user acting under compulsion actually discloses all the data. We next give two security definitions used to measure the security level in storing and querying hidden partition.

Definition 1 (Indistinguishability). *For every query pair Q_s, Q_h querying on the same set of shared data. Suppose Q_h also queries on data in hidden partition. The Q_h is indistinguishable if for every polynomial-size circuit family $\{C_n\}$, every polynomial p, all sufficiently large n, and each occurrences of Q_s, Q_h*

$$|Pr\{C_n(Q_h)=1\}-Pr\{C_n(Q_s)=1\}|<\frac{1}{p(n)}$$

This definition says that if we try to construct a polynomial circuit for distinguishing any given Q_h (i.e., the circuit will output one if Q_h queries hidden partition, else it will output zero), the circuit will have a success probability that is at most slightly better than a random guess. This guarantees that the adversary could not detect whether the target query accesses hidden partition by comparing observations of this query and observations of the same query not involving hidden partitions.

Definition 2 (Plausibility). *For a query, Q_h involving data in hidden partition, the Q_h is plausible if for every polynomial-size circuit family $\{C_n\}$, every polynomial p, all sufficiently large n,*

$$|Pr\{C_n(Q_h)=1\}-\frac{1}{2}|<\frac{1}{p(n)}$$

The probability in the above terms is taken over the all occurrences of Q_h sent to a StegDB. This definition states that if we try to construct a polynomial circuit for distinguishing any given Q_h (i.e., the same circuit in Definition **1**), the circuit will have a success probability that is at most slightly better than a random guess. This guarantees that the observation of the internal operations in database engine and disk activities could not convince the adversary that current query is accessing hidden partitions.

Definition 1 assumes a more powerful adversary who is able to intercept the original query. A query that is plausible when executed alone might be distinguishable by comparing this query with the same query statement but not involving hidden partitions. For example, after an adversary observes a suspicious query, he reissues the exact query statement to the database server, not involving any hidden partition this time. By comparing his observations, the adversary might find different system behaviours between the suspicious query and his query, which reveal the existence of hidden partition. We will discuss the different behaviours in later sections, and we shall assume design of StegDB in this chapter follows the plausibility definition.

2.3 Three System Models

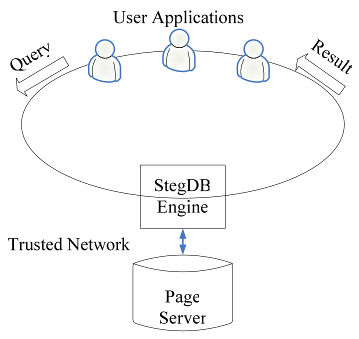


Figure 2.1 Model 0

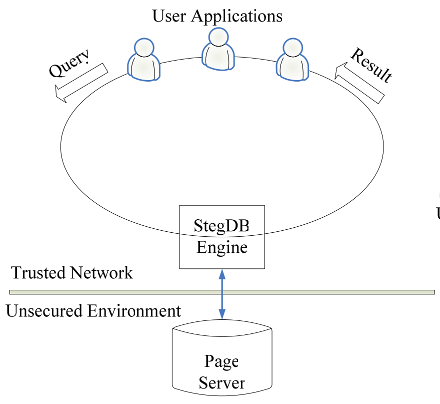


Figure 2.2 Model 1

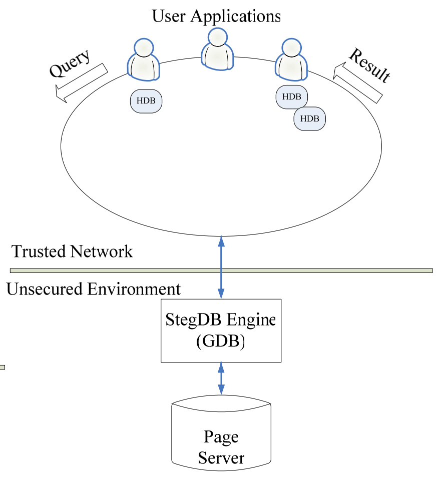


Figure 2.3 Model 2

The Steganographic Database Model consists of three entities: (1) the users(s), (2) the database engine and (3) the page storage. The database users create, modify and query the contents of the database. The database engine serves the users with query results. The page storage hosts the physical storage and response to block access requests. There are three system models according to different locations of database engine and page storage. 1) In model 0, as shown in Figure 2.1, all components are situated in a secured environment; 2) In model 1, as shown in Figure 2.2, the users and the database engine are all situated within a trusted network, and the page storage is in an unsecured environment; 3) In model 2, as shown in Figure 2.3, the database engine is moved from trusted environment to untrusted environment.

The three models present three cases with different attackers that present a major increase in the capabilities.

- In model 0, the adversaries are curious system users such as database administrators that may want to see if the target user has hidden data by examining the content of database. The major counter-measure in StegDB is to hide the sensitive data together with its associated schema data and transaction log. Meanwhile, the StegDB runs like as a MLS database so no operations on the sensitive data could be detected by administrators, without authorization of the owner of hidden partitions. As this model is similar to Multilevel Secure Database System (MLS), we will not consider this model any further.
- In model 1, there are several security issues: 1) privacy of data during transmission; 2) privacy of static stored data; and 3) privacy of data during access. The first issue,

privacy during network transmission, has been studied widely in Internet security and addressed by the Secure Socket Layer protocol (SSL) [6] and Transport Layer Security (TLS) protocol [7]. The second issue, protecting static data from illegal copying, is addressed through data encryption and proper key management. In this thesis, we focus on the third issue—security breaches from the data traffic at the page storage. In other words, the adversary could passively observe the disk activities and scout the disk for evidences of hidden data, e.g. through the sequence of page locations exposed during the access of pages. The attacker might observe the patterns in page access sequences. The regular access patterns in private storage where sensitive data locate might disclose their existence. In my work [8], we used probability-based relocation and random prefetching strategies to transform the access sequences into a random-looking one, which prevents the adversary from analysing passively access patterns by monitoring the access activities.

- In model 2, the codes and internal operations during runtime are under the eyes of attackers or compromised administrators. Even if the database engine is well protected through the signatures on the binary codes, an attacker can still observe passively the internal operations of database engine. Hence, transforming traffic alone is insufficient as the attacker now can observe the logic page number by probing the kernel of the database engine. The major problem is then how to prevent the attacker from inferring the existence of hidden data by monitoring the internal operations. In this thesis we will show a design of a StegDB which countermeasures against the above mentioned threats, especially the third threats.

All three system models can be found in real-life applications which have different security requirements. For example, common practical scenarios for model 1 where the storage server is not completely trusted include shared storage area networks (SAN) over TCP/IP network accessed through iSCSI, a block-level access protocol. The model 2 is similar to “Outsourced Database Model” in which organizations outsource their data management needs to an external service provider that hosts client's databases and offers seamless mechanisms to create, store, update and access (query) their databases.

2.4 Literature Review

2.4.1 Multilevel Secure (MLS) Database

The rationale behind StegDB is similar to a Multilevel Secure (MLS) database [9, 10] in that an unauthorized database user cannot query or even infer the existence of high level sensitive data. In a MLS database management system, database components are classified by specific security labels, called classifications and are representing the sensitivity of the classified information. Users and applications that have access to database components have a security label called clearance. The list of labels is partially ordered and forms a lattice and the ordering is called dominance. Let *cd* be a classification of data item *d* and *cu* be a clearance of user *u*. For a successful read access from user *u* to data *d* the mandatory access control requires that $cu \geq cd$ (*cu* dominates *cd*), or for a successful write access that $cu = cd$ (restricted *-property of [11]). One can envision a StegDB with activated hidden partition as a MLS database, where the shared data are unclassified, and data in hidden partition are classified.

To close signaling channel which arises when an unauthorized user insert a tuple that has the same primary key values as an existing but hidden tuple, the polyinstantiation technique is introduced into MLS database system [11]. It means more than one tuple with the same apparent key value but different attribute values for users at different classification levels. StegDB also adopt polyinstantiation to prevent similar signaling channel.

However, there are two key differences: 1) MLS databases are based on trusted hosts which are often absent from outsourced environment. The untrusted host of a StegDB, thus, causes major problems in data storage and query which will be discussed in later sections. 2) Hidden partitions are only available for authorized users. Without a HPK, the StegDB could not locate the hidden partitions; while in a MLS system the sensitive data are always known for trusted components of MLS system.

2.4.2 Secured Database Storage

Incorporating encryption into database seems to be quite a recent development among the industry database providers [12] [13]. Database encryption consists in encryption data stored within a database in order to protect it from being comprised [14]. If the information managed by a hosted database is encrypted, a hacker who breaks into will not be able to access it; furthermore (and perhaps more importantly) a powerful database user who either intentionally or accidentally displays critical data will not be able to understand them. The encryption operation can be performed at various levels of granularity. In general, finer encryption granularity affords more flexibility in allowing the server to choose what data to encrypt. The obvious encryption granularity choices are:

1) field value: each attribute value of a tuple is encrypted separately; 2) row: each row in a table in encrypted separately; 3) column: certain sensitive attributes are encrypted; 4) page: whenever a page is stored on disks, the entire block is encrypted.

Some recent research has focused on an untrusted server model in which a client does not event trust the serve with cleartext queries [2, 3, 15]. Damiani et al proposed a framework for the management of encrypted databases, within which the server performs encrypted queries over encrypted data; then the client decrypts and further processes returned results. The framework cannot cope well with range queries. Note that this model is similar to ours.

In all the above solutions, even the owner of the data controls the whole encryption procedure including encryption key and shield DBA from seeing sensitive data in plain state, by examining the scheme of database, the adversary or a comprised DBA still know the existence of sensitive data and on which pages these data are located. The adversary or DBA can then decrypt the data page with brute force or coerce the owner into extorting a confession. For high data privacy environments, letting unconcerned people know the existence of sensitive information is security time bomb.

In [16], pang et al introduces a revised B+-tree, HACCS (Hierarchical Authentication cum Correction Coding Scheme), to enforce data integrity for databases that are hosted on open servers, which may be susceptible to attacks on their operating systems or physical devices.

2.4.3 Query on Encrypted Database

In database-as-service model [2, 3], data are encrypted before stored at the server provider, which can only be decrypted by the owner. It deploys a “coarse index” which allows partial execution of an SQL query on the provider side without the need to decrypt the stored data at provider side. In particular, the attribute values in question are mapped into several partitions. Then a SQL query procedure is split into two phases. First, the sql statement is modified according to those attribute partitions. For example, the SQL “select * from T where attribute1 = 123” on server side may look like “select * from T where attribute1 = map(123)” where map(123) is the partition that 123 fall in. The result is sent to the client, where the correct result of the query is found by decrypting the data and executing a compensation query to filter the received result removing redundant records, because it might be three attribute values mapped into that partition.

In summary, the model needs to split a SQL query into a server query and a client query. The service provider retains the responsibility to manage the persistence of the data. There are two constraints: 1) the mapping function determines the security level of model. A poor mapping function give more chance for the adversary to deduce the distribution of encrypted data. 2) Clients need to maintain those value-partition mappings locally.

2.4.4 Traffic Analysis

Even with a steganographic file system, an attacker who is monitoring the storage might be able to analyze the patterns of the update or data traffic activities, and from there deduce the existence of hidden files. This is the traffic analysis problem [17].

Traffic analysis has been studied extensively in the context of privacy-providing systems, such as the MIX networks [18]. While all these techniques serve to prevent private information from being released to adversaries through the data traffic or access patterns, different mechanisms are adopted according to the peculiar objectives and requirements of individual systems. Two privacy protection mechanisms that could be adapted to solve our problems are oblivious RAM [19] and private information retrieval (PIR)[20].

PIR enables users to privately retrieve their information from a secondary storage system, such as a database. With such a mechanism, user data are stored into multiple databases that are not aware of each other, so that a user can retrieve data without revealing them. However, all the existing schemes of PIR [21, 22] only concentrate on reducing the communication complexity, but ignore the I/O overheads. Specifically, most of them need to scan the entire storage volume for every query, and are too expensive for a generic file system. Oblivious RAM is a tamper-resistant cryptographic processor that serves to protect code privacy and prohibit software copyright violation. Even an attacker who can look into the memory and monitor the memory accesses (reads or writes) cannot gain any useful information about what is being computed and how it is being computed. In [19], the oblivious RAM's processing overhead is reduced to O((log t)3) where t is the number of computation steps of the RAM.

Assuming an attacker knows that a page server is hosting a (relational) database that exhibits regular, predictable page access patterns, he can try to identify the patterns that are repeated over a period of time. From the sequences of pages, he can reconstruct parts of the database, before using dictionary attacks to recover the content. In [8], we propose two methods to counter traffic pattern analysis in the context of StegDB: 1) relocating the pages in certain possibility during accessing, which disturb the potential traffic patterns; 2) inject the dummy traffic patterns to mimic the real traffic patterns to prevent the attacker from knowing the exact traffic patterns who may then further deduce the existence of hidden pages. The experiment results show that, by applying the above counter measures, most of actual traffic patterns are disturbed and indistinguishable. In order to justify that the attacker can distill useful information from the traffic analysis, we developed an effective sequence mining algorithm to find the frequent episodes meeting predefined constraints in an access sequence [23].

2.4.5 Steganographic Data Storage

While access control and encryption can safeguard the content of protected folders, an unauthorized observer can still establish their existence and coerce the owner(s) into unlocking them.

Steganography provides a countermeasure against this vulnerability, by preventing an attacker from verifying whether user acting under compulsion actually discloses all of the data. Derived from a Greek word that literally means "covered writing", steganography is about concealing the existence of messages and encompasses a wide range of methods like invisible ink, microdots, covert channels and character arrangement. This contrasts with cryptography, which is about concealing the content of messages. While the practice of steganography dates back many centuries, the modern scientific formulation was first given in [24]. Since then, many studies have investigated ways of embedding a secret message, be it an electronic watermark, a covert communication or a serial number, within still images [25], text [26], audio [27] and video [27].

In [28], Ross Anderson et al presented the first schemes for a steganographic file system that hides data directly on a raw disk volume. However, this is achieved at the expense of incurring high processing overhead and/or risks of data loss. In [5], Pang etc proposed a practical steganographic solution, StegFS, that overcame those limitations. In a StegFS, a number of randomly selected blocks are initially filled with random data and abandoned by the system. After that, the data blocks of useful files, which are encrypted under the files' access keys, are scattered across the storage space in such a way that they can only be located through the access keys. Therefore, an attacker without the files' access keys cannot distinguish between useful blocks of hidden files and abandoned blocks, and thus cannot deduce the existence of the files. As StegFS was designed for local storage devices, it does not address the additional risk of traffic analysis that shared network storage must contend with.

Chapter 3

Mining Access Patterns in Untrusted Page Storage

With our system model 1, we must assume that in the worst case an attacker is able to monitor all access activities on the page storage. In particular, to observe the storage I/O activities, an attacker could take either of the following two methods: 1) scan the storage volume repeatedly to look for changes in the raw content, which could be done through remote access; 2) trap the I/O requests between the DB engine and the page server, either by installing a malicious file system filter driver or by spreading viruses to intercept system calls. Both ways could be achieved through inserting a filter driver in a stackable file system [29], as shown in Figure 3.1. The installed filter driver logs the page number and reference time for further analysis, where the reference time refers to the time that the page is accessed.

In this chapter, we first review the types of page reference patterns that a DBMS generates in the course of executing relational operators. Then, we provide some examples in that an attacker might launch attacks according to those patterns. Before presenting our solution, in this chapter, we will look at how easy it is to design frequent episodes mining algorithms to locate repeated patterns.

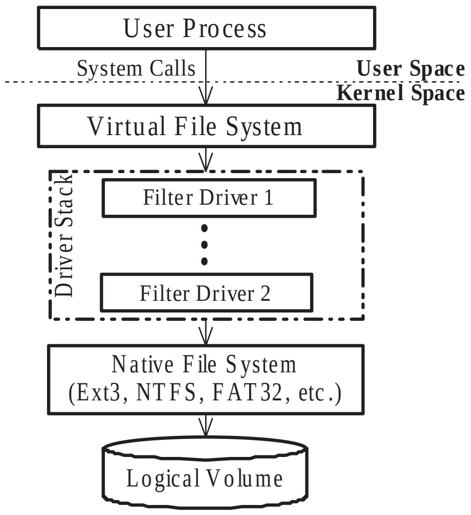


Figure 3.1 Architecture of Stackable File System.

3.1 Database Reference Patterns

In [30], Chou and DeWitt first observed that the pattern of page references exhibited by database operations in relational database systems are very regular and predictable. These reference patterns fall into three categories: sequential, random and hierarchical.

- Sequential references: In a sequential scan, pages in a table are retrieved and processed one after another. Operations like a selection on an unordered relation involve only one scan, whereas other operations like nested loop join and merge join may scan (parts of) a table repeatedly.
- Random references: These access patterns are commonly observed in retrieving the leaf nodes of a non-clustered B+ tree index. Such a pattern could be repeated within the same operation, for example during an index nested loop join.
- Hierarchical references: A hierarchical reference is a sequence of page accesses that form a traversal path from the root down to the leaves of a tree index.

Based on the above classification, the authors proposed a DBMIN algorithm that carries out buffer allocation and replacement according to the reference pattern for each operation. Nevertheless, even where DBMIN is implemented, parts of the access patterns will appear at the server if an operation is not given its full buffer allocation.

In addition, the data access sequence for a query is usually predictable after it is optimized. This information on which data pages are likely to be accessed next is commonly exploited in database buffer management and prefetching to improve performance [31, 32]. In this chapter, we exploit the same predictive information to re-move recurrent patterns in the server's I/O traffic.

3.2 Threat from Page Reference Patterns

As explained above, recurring reference patterns in the disk access sequence provides hints on repeated database operations, and thus the logical links among physical pages. One way for an attacker to locate repeated patterns is to run a sequence mining algorithm on the disk activity log. For this work, we developed two sequential mining algorithms (EPT and PPS) in Chapter 3, based on the minimal occurrence method that counts the distinct occurrences of each detected pattern. Running over the page access sequence from the storage server's activity log, the algorithms find all sequential patterns, e.g. lists of sets of page references, with user-specified minimum number of occurrences and constraints which defines the durations or structures of patterns. Through fine-tuning the constraints, we captured, on average, over half of the actual patterns in conducted experiments. Having obtained those informational patterns, additional attacks could be conducted. The following are some typical examples.

- Database operations present different reference patterns. For example, as shown in Figure 3(a), a continuous repeated reference pattern might hint at a loop join operation on a table; reference patterns sharing a common prefix might suggest that those involved pages are part of an index. Piecing together that information, a powerful attacker could infer some of the operations currently being executed in the database engine.
- Figure 3(b) shows another example. Suppose that the attacker could get the information about the user sessions of individual users (e.g. when they start and end) for instance through social engineering. The attacker then is able to mine for access patterns associated to any target user. Specifically, he first identifies those intervals when the user is logged on to the DB engine, then mines the combined sequences for repeated reference patterns. If a repetitive pattern (28274) appears rarely outside of this user's sessions, it would imply that the data in those pages is critical to this particular user. This could prompt the attacker to attempt to decipher the pages or, worse, coerce the user into disclosing the data.
- In addition, the frequencies of access patterns may also indicate which parts of the database are more useful than others, so that the attacker may pay more attention to them.

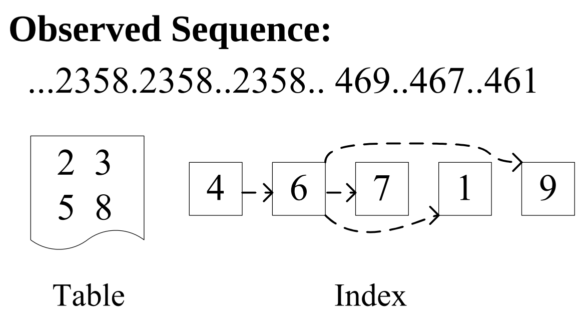


Figure 3.2 Hint about physical structure.

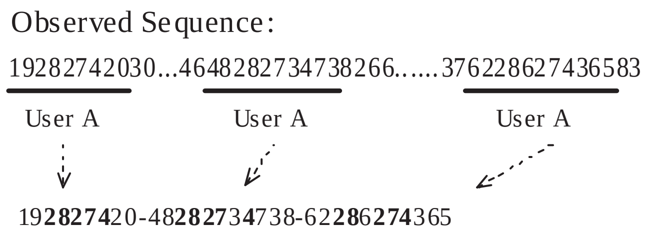


Figure 3.3 Mining Reference Patterns of Particular User.

We developed two sequential mining algorithms (EPT and PPS), based on the minimal occurrence method that counts the distinct occurrences of each detected pattern. Running over the page access sequence from the storage server's activity log, the algorithms find all sequential patterns (episodes), e.g. lists of sets of page references, with user-specified minimum number of occurrences and constraints which might define the durations or structures of patterns. Through fine-tuning the constraints, we captured, on average, over half of the actual patterns in conducted experiments. Having obtained those informational patterns, additional attacks could be conducted.

3.3 Problem Description

Like most existing episode-related research, we adopt the notation presented in [4] . For the sake of completeness, we summarize the relevant definitions below.

In the episode model, an event sequence *S* is a history of events. Each event has at least a type and an occurrence time. Several events may occur at the same time, and there is no guarantee that events would happen at any given time unit. Given the set $R = \{A_1, A_2, ...A_m\}$ of event attributes with domains $D_{A_1}, D_{A_2}, ..., D_{A_m}$ respectively, an event e over R is a (m + 1) tuple $(a_1, a_{2,}, ..., a_m, t)$, where $a_i \in D_{a_i}$ and *t* is a real number, the time of *e*. We denote the occurrence time of e by *e.T*, and an attribute $A \in R$ of *e* by *e.A*.

An episode (pattern) is defined to be a collection of events that occur relatively close to each other in *S*. There are three kinds of episodes: 1) serial, defined as an ordered list of events; 2) parallel, defined as a set of events; and 3) composite, defined as an ordered list of sets of events. Constrained frequent episodes are a particular type of episodes that satisfy certain constraints, noted by *C*, and their occurrences in *S* satisfy a specified minimum support, noted by min_sup. The mining task is to find the complete set of constrained frequent episodes.

For simplicity, the episodes as mentioned in the rest of this chapter refer to serial episodes comprising consecutive events; support for gaps in adjacent events and parallel and composite episodes will be discussed in sections 3.4 and 3.5.

Example 1 *An example of event sequence is shown in Figure 3.4. With a minimum support of 2, the event sequence contains two possible episodes,* $< ED >$ *and* $< BC >$*. Note that the episodes comprise consecutive events. Episodes that contain gaps in adjacent events will be supported in constraints (Section 3.4).*

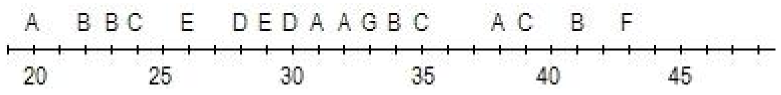


Figure 3.4. A sequence of event.

3.4 Episode Prefix Tree (EPT) Algorithm

In this section, we describe the data structure underlying EPT (Episode Prefix Tree) algorithm before presenting the associated episode mining algorithm.

Definition 1(EPT-tree). *An EPT-tree stores the episode prefixes that are at most k in length, where k is a constant called Depth Threshold.*

- It consists of one root labeled as "null".
- Each internal node registers two pieces of information: the label of an event, and an occurrence count.
- For any node N in the tree, the nodes in the path from root to N(inclusive) form a prefix of episode ending with the event associated with N. The count of N registers the number of occurrences of the corresponding episode.

Example 2 *For the event sequence S in Example 1, the corresponding 2-depth EPT-tree is shown in Figure 3.5. In this EPT-tree, every path from the root to a descendant node represents a sub-sequence of S. Associated with each node is a counter that records the frequency of the corresponding subsequence. For instance, the subsequence $<BC>$ occurs twice in the event sequence, while $<AC>$ occurs only once. Note that this EPT-tree looks like a truncated suffix tree, while the difference will become fundamental when an EPT-tree supports composite episodes and constraints which will be discussed later.*

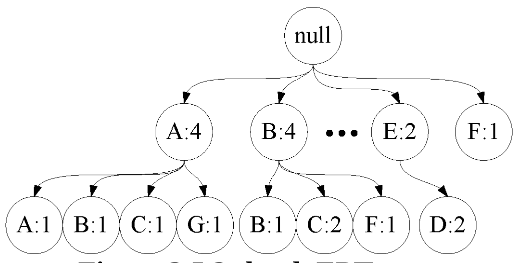


Figure 3.5 2-depth EPT-tree.

When the user-specified depth threshold k, is larger than the length of longest episode in the event sequence, the procedure of constructing the EPT-tree can be regarded as a mining procedure. However, such a mining procedure is inefficient because it grows the non-frequent paths along with the frequent paths. We observed some properties of the EPT-tree structure that facilitate the mining task.

Property 1 Non-growth path: *A non-growth path ends at a leave node with a counter below the minsup. Frequent episodes cannot have a prefix that constitutes a non-growth path.*

Property 2 Depth threshold: *The process of building an EPT-tree with depth threshold k and removing non-growth paths can be accomplished in two steps: 1) build an EPT-tree with depth threshold k_1 ($0 < k_1 < k$) then remove non-growth paths; and 2) extend remaining branches by k_2 such that $k_1 + k_2 = k$, then remove non-growth paths.*

This property can be proved easily as the branches pruned in the first step are non-growth paths that do not contribute to episodes longer than k_i. This property can be generalized so that the EPT-tree is constructed in more than two steps.

Based on the above properties, we have Algorithm 1, which takes as input a series of depth thresholds, $\{k_1, k_2, \ldots k_n\}$. The basic idea of the algorithm is to grow the EPT-tree in multiple phases. In each phase, we grow the EPT-tree to the next depth threshold, k_i, and prune all non-growth paths, which entails a scan through S. The algorithm stops when there is no more path to grow.

Algorithm 1 EPT

```
$i = 1$
repeat
  scan the event sequence $S$.
  for all events $e_i$ in $S$ do
    find the node $nd_j$ at level $\sum_{l=1}^{i-1} k_l$ in the tree associated with $e_j$ by re-
    recognizing the prefixes.
    call extendnode ($nd_j, S, C, k_i$)
  cache the frequent parts of those non-growth paths.
  prune all non-frequent and non-growth paths.
  $i = i + 1$
until there is no path left in the EPT-tree.
output episodes from the cache.
```

Algorithm 2 extendnode**(**et, nd, S, C, k_i**)**

```
Input: $et$ : the exploring event. $nd$ : the node to be expanded.
  if $\text{depth}(et) \geq \sum_{l=1}^{i} k_l$ then
    return.
  $next\text{-}event\ set$ ={$e_j$ | $e_j$ follows et and satisfies $C$}
  for all events $e_j \in next\text{-}event\ set$ do
    create a child node $node_j$ of $nd$ associated with $e_j$ if it not resent.
    call extendnode($e_j, node_i, S, C, k$)
```

Example 3 *Given the depth series {2, 3, ...}, the minimum support 2, and the 2-depth EPT-tree in Example 2. All the extensions from (A) are non-growth paths, e.g. (A)(B), we then add (A) into the result set and prune this non-growth branch. Next we prune the non -frequent paths (F). The resulting 2-depth EPT-tree after pruning is shown in Figure 3.6.*

After scanning the sequence again, the EPT-tree grows to 3-depth as shown in Figure 3.7. Here we output the frequent prefixes of the non-growth paths, i.e., (B)(C) and (E)(D). After pruning the non-growth paths this time, there are no paths left and the mining process stops. The mining result thus contains $< A >, < BC >$ *and* $< ED >$*. Note that the algorithm produces only the longest episodes.*

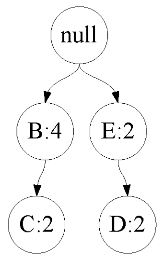


Figure 3.6 2-depth EPT-tree after pruned.

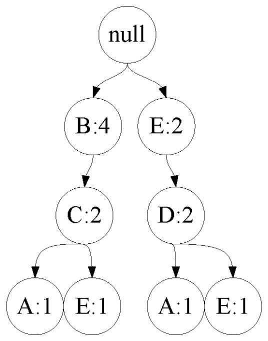


Figure 3.7 3-depth EPT-tree.

Correctness Analysis: let α be a length-l path, and $\{\beta_1, \beta_1, ..., \beta_m\}$ be the set of all frequent length-(l + 1) paths having prefix α. Considering the enumeration of event patterns in the above algorithm, the complete set of paths having prefix α can only be composed of elements in the β set and those non-frequent pruned paths, which means the prefix space is searched completely. Hence, EPT returns the complete set of episodes.

Space Complexity: for any frequent episode in S, there exists a unique path in the EPT-tree starting from the root such that all labels of the nodes in the path are exactly the same as the events in the episode. This ensures that the number of distinct leaf nodes as well as paths in an EPT-tree cannot be more than the number of distinct frequent subsequences in S, and the height of the EPT-tree is bounded by one plus the length of longest frequent episodes. If all episodes are continuous, in worse case the space requirement is $O(n \cdot m / min\_sup)$, where min_sup the minimum support, n is is the length of S and m is the types of events. Otherwise, the average space requirement depends on the content of the constraints, and the worst case is $O(n^2 \cdot m / min\_sup)$.

Depth Threshold: the depth threshold plays an important role: a low depth threshold helps to save space, while a high depth threshold helps grow the tree more quickly. A finely-tuned threshold series should balance between speed and space consumption, which at present can only be preset by users through trial and error. One quick way to get a satisfactory depth series is through mining a sample set of slices from the event sequence.

3.5 Position Pair Set (PPS) Algorithm

Although the EPT does not generate candidates by growing the episode prefixes, it still leaves ample room for improvement. First, it is tedious to repeatedly scan the sequence and re-recognize the episode prefixes. Moreover, during the next scan a lot of effort is wasted in growing non-frequent branches to *k+1* in depth. The PPS algorithm is designed to overcome those costly operations by caching the position of the beginning and ending events of an occurrence of an episode prefix, called position-pair.

Property 3 Position-pair: *If occurrences of all prefixes of an episode follow the definition of minimal occurrence, then, for each prefix, a position-pair uniquely locates its items in the event sequence. For example, given an event sequence,* $<$CABEBDE$>$*, the position-pair of episode* $<ABD>$*, is (2, 6), where A is at 2, B at 3 and D at 6.*

Like EPT, PPS works by growing frequent prefixes. However, the strategy that PPS employs is quite different. Whenever an episode prefix is found, all position-pairs of that prefix are cached. During subsequent growth of this episode prefix, only the slices immediately following those position-pairs need to be examined instead of the whole sequence. In other words, position-pairs are used to effectively split S into smaller slices, so that when growing the prefixes, only those slices need to be examined.

Prefix	Position-pair sets	Prefix	Position-pair sets
<A>	(20), (31), (32), (38)	<E>	(26), (29)
<B>	(22),(23),(34),(41)	<BC>	(23,24),(34,35)
<C>	(24), (35), (39)	<ED>	(26, 28), (29, 30)
<D>	(28), (30)		

Table 3.1 The prefixes and their position-pairs.

Example 4 *Consider the sequence shown in* **Figure 3.4** *and suppose min_sup = 2. The set of events is {A, B, C, D, E, F, G}. The frequent episodes in S can be mined in following steps.*

Step 1. Find length-1 episodes and their associated position-pairs. Scan S once to find all frequent episodes. They are A:[(1), (9), (10), (14)], B:[(2), (3), (12), (16)], C:[(4), (13), (15)], D:[(6), (8)] and E:[(5), (7)], where prefix: [position-pair set] represents the prefix and its associated position-pair set. Since the two parts of the position-pair are the same here, in order to save space we only record one of them.

Step 2. Divide the search space. The complete set of episodes can be partitioned into the following five subsets according to the five prefixes: (1) those having prefix A; ...; and (5) those having prefix E.

Step 3. Grow each prefix separately and recursively. We look for episodes beginning with event E. By checking the events after position 26 and 29, we obtain a frequent episode <ED>. We then continue to grow the prefix <ED> by examining the events after position 28 and 30, there is no further growth for <ED>. **Table 3.1** *lists all frequent prefixes and their position pairs.*

The PPS algorithm is shown in Algorithm 3; it explores the search space in a depth-first search (DFS) manner.

Algorithm 3 **PPS**

```
scan S once for 1-length episodes; add them to active-set.
scan S again; find their associated position-pairs.
while active-set is not empty do
  get next prefix $pf_i$ from the head of active-set and its associated position-pairs.
  for $pp_i \in$ position-pairs do
     next-events set = {$e_k \mid e_k$ is in the following slice of $pp_j$ and
     satisfies C}.
     for all events $e_k \in$ next-events set do
        append $e_k$ to $pf_i$ and record the position-pairs of the new formed prefixes.
  if frequency(new formed prefixes) > min_sup then
     add the new formed prefixes to active-set.
  else
     cache $pf_i$.
  output episodes from the cache
```

The correctness of PPS can be proved the same way as for EPT. Hence, PPS returns the complete set of episodes. Due to the introduction of position-pair, PPS requires the entire sequence to be hold in main memory. Since all algorithms for finding frequent episodes are CPU-bound, this assumption is not very limiting in practice.

3.6 Constraint Support

To support constraints in EPT and PPS, only those events that satisfy the constraints are appended when growing the episodes prefixes, which ensure that the prefixes satisfy the constraints all the time. It also leads to an easy implementation of the numeric prefix constraints such as *time constraints, regular expression, length constraint, duration constraints* and so on.

In this section we take the time constraints for example, which mainly include: 1*) max-gap*: the maximum allowed time difference between two successive occurrences of events; 2) *min-gap*: the minimum required time difference between two successive occurrences of events; 3) *max-during*: the maximum allowed time difference between the latest and earliest occurrences of events. Note that as the satisfied events are not limited to succeeding events, the *max-gap* and *min-gap* constraints could introduce gaps between adjacent events. The below is an example involving PPS; EPT operates in a similar way.

Example 5 *Consider the sequence shown in Figure 1, and suppose max-gap=3, min-gap=1. The satisfied events after the first B at position 22 is {B:[(23)], C:[(24)]}, thus two prefixes < BB > and < BC > are found comparing to one prefix in the no-constraint case. The 2-length and 3-length prefixes are listed in* **Table 3.2**.

Prefix	Position-pair sets	Prefix	Position-pair sets
<AB>	(20,22),(32,34),(38,41)	<ED>	(26,28),(29,30)
<AC>	(32,35),(38,39)	<ABC>	(20,24),(32,35)
<BC>	(23,24),(34,35)		

Table 3.2 The prefixes and their position-pairs after applying time-constraints.

3.7 Parallel Episode and Composite Episode

Up till this point, we have discussed only serial episodes, while in real life applications the events in some episodes are often not necessarily ordinal. Those kinds of episodes are parallel or composite episodes. The two proposed algorithms can also be extended to handle parallel and composite episodes. As serial and parallel episodes are specialized forms of composite episodes, we will only describe the process of mining composite episodes.

To mine composite episodes, we extend the definition of episode prefixes to include parallel components. That is, we allow this forms of prefix, A(BC)D where the relative order between B and C is immaterial. The mining process is the same as that for serial episodes, except that appending an event from next-events to a prefix leads to two extended prefixes — one is appended as a separate component of the prefix, while the other is merged into the last component of the prefix. For example, after appending E to <A(BC)D>, we get <A(BC)DE> and <A(BC)(DE)>. To avoid an exponential increase in the number of prefixes, we introduce another time related constraint called window-size, the maximum allowed time difference among the parallel events in the episodes.

Example 6 *Continuing with Example 5, suppose we have an additional constraint window-size=2. The new prefixes and their position-pairs are listed in Table 3.3, i.e., parallel episode* <(BC)> *and composite episode* <A(BC)>*. Note that the introduction of window-size actually relaxes the time constraints in Example 5.*

Prefix	Position-pair sets
<(BC)>	(23,24),(34,35),(39,41)
<A(BC)>	(20,24),(32,35),(38,41)

Table 3.3 The prefixes and their position-pairs after applying window-size.

3.8 Discussion

Finally, we highlight the differences between our proposed algorithms and existing techniques, and motivate some of the design choices made in EPT and PPS.

Both proposed algorithms utilize an episode-growth strategy, thus no candidate episode is generated. That is, they only need to count episodes instead of testing whether they exist in the event sequence first. Therefore, EPT and PPS search a much smaller space compared to Apriori-like algorithms. The resulting performance difference is much more significant in situations where the minimum support is low or there are a lot of event types.

Our algorithms do not have sliding windows, so there is no restriction on the length of discovered episodes. While MINEPI does not apply sliding windows either, it cannot identify some occurrences of episodes due to algorithm limitation. For example, in slice <ABBC> , as the minimal occurrence of $< AB >$ and $< BC >$ do not overlap, MINEPI cannot recognize the candidate ABC. By contrast, our proposed algorithms can correctly identify those episodes.

For mining composite episodes, we also introduce another form of constraints—window-size—to reduce the search space, which imposes a limit on the maximal number of parallel events that can occur in a component of an episode.

3.9 Implementation and Evaluation

In this section, we present a performance comparison of EPT and PPS with MINEPI that is the state-of-the-art algorithm for mining frequent episodes and uses the minimal occurrences too. We implemented the MINEPI algorithm to the best of our knowledge based on the published reports [8]. Nevertheless, in most cases, the mining result of MINEPI is slightly different from our proposed algorithms due to its own limitation (see Section 3.6). In particular, MINEPI cannot identify some occurrences of episodes (see Section 3.6), which causes 1) the support of some episodes (*1% - 5%*) to fall below the minimum support so that they are filtered out of the final result; 2) *20% - 50%* of the frequencies of resulting episodes are slightly below that of their counterparts in our proposed algorithms. All experiments are performed on an Intel PC with a 2 GHz Pentium 4 CPU, 512 MB memory. All programs are written in Microsoft/Visual C++7.1.

The experiments are conducted on both synthetic and real sequences. The synthetic single sequences are generated by a modified version of the data generator from IBM AssocGen [33] and labeled with the parameters in Table 3.4, e.g. T500N100L5S200k indicates a sequence generated with $T = 500,\ N = 100,\ L = 5,\ S = 200,\ 000$. The synthetic sequences are very dense, in the sense that almost every event in a sequence is involved in one frequent episode. The real sequences are derived from a collection of text documents. Each word is an event and is indexed consecutively to give it a "time". Sentence boundaries cause a gap. In this context, the mining results can reveal the frequent phrases in the documents. Due to the consistent results we report here only results for L2 which has 153799 events with 8345 event types.

Parameter	Description	Range
$S_{sequence}$	Size of the event sequence	20,000~2,000,000
T_{event}	Number of event type	300~10,000
N_{fe}	Number of frequent episodes	100~5,000
L_{avg_fe}	Average length of episodes	5~100

Table 3.4. Synthetic database parameter.

Besides the event sequences, the parameters required by the mining algorithms are the minimum support *min_sup* and constraints such as *min-gap*, *max-gap, max-span* and *regular expression*. The default series of depth thresholds for EPT is *{1,1,1, ...}*, i.e., the EPT-tree is grown by one level each time. The primary performance metric is the overall runtime of each mining task inclusive of both CPU time and I/O time. To evaluate the effect of a constraint, we define the selectivity of a constraint as the ratio of the number of episodes satisfying the constraint against the total number of episodes. Therefore, a constraint with *100%* selectivity filters out no episode, while one with *0%* selectivity filters out all the episodes.

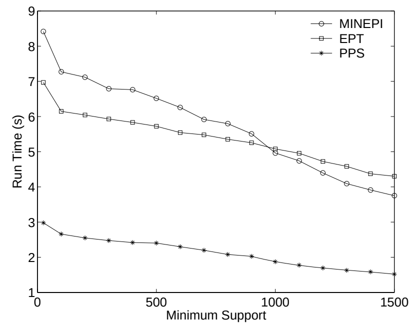


Figure 3.8 Minimum support on T500N100L5S400k.

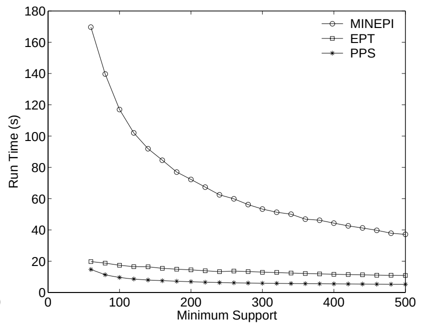


Figure 3.9 Minimum support on L2.

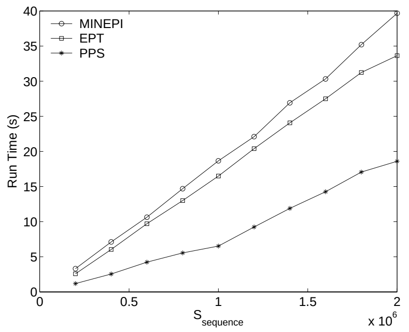


Figure 3.10 Sequence length on T500N100L5.

3.9.1 Varying Minimum Support

The execution times of the three algorithms with different minimum thresholds are shown in Figure 3.8 and Figure 3.9. On synthetic sequences, PPS is much more efficient than EPT and MINEPI, while EPT is faster than MINEPI at low support thresholds. On real-data sequences, L2, EPT is roughly 100% slower than PPS. Moreover, unlike MINEPI, EPT and PPS degrade much slower as the minimum support decreases. Hence, PPS is a clear winner in both experiments. The result is expected as EPT needs to scan the data sequence repeatedly until the longest episodes are found, and within each scan the prefixes of episodes are repeatedly recognized. The costs are especially high on episode-dense sequences. MINEPI performs the worst because it generates too many candidates. PPS only needs two full scans of the event sequence, and its use of position-pairs avoids recognizing prefixes.

3.9.2 Varying Sequence Length

Figure 3.10 plots the execution times of the three algorithms with respect to the length of the event sequence. EPT and MINEPI still have similar performance here. As the data sequence becomes longer, the execution time difference between PPS and MINEPI becomes more pronounced. As the effect of increasing sequence length is similar to that of decreasing the minimum support, this experiment shares the same explanation as the previous experiment.

3.9.3 Varying Average Length of Episodes

This experiment aims to investigate the performance of the algorithms in mining long episodes. As shown in **Figure 3.11**, EPT and PPS cope better with longer episodes, whereas MINEPI cannot handle long episodes well. This is because, in MINEPI, a long episode must grow from an exponential number of short episodes. In contrast, EPT and PPS do not generate short episodes that never occur in the sequence.

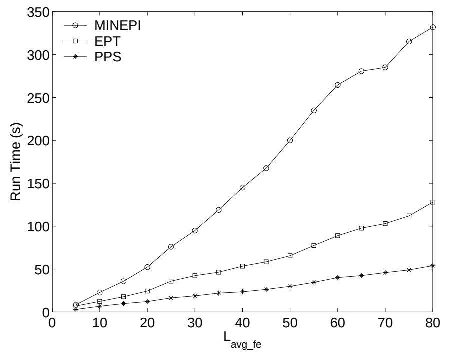


Figure 3.11 Episode length on T500N100S400k.

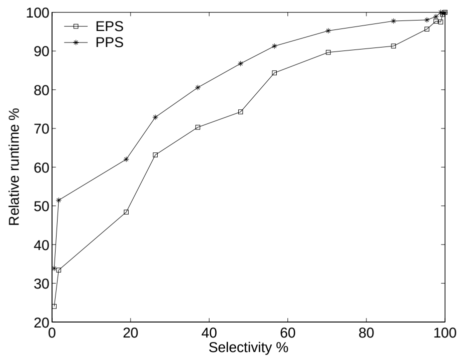


Figure 3.12 Capability of PPS and EPT on pushing max-during constraint.

3.9.4 Effectiveness in Using Constraints

In this experiment, we use three representative constraints — time, aggregate and regular expression — to examine their impact on performance. As expected, the performance impact caused by time constraints (except max-during) depends heavily on the duration of the constraints, as they create an exponential number of new prefixes during the mining process. For aggregate, regular expression and max-during constraints, the performance impact is pegged to the selectivity of those constraints. The results of max-during constraints are shown in Figure 3.12. When the constraint selectivity is high, not much time can be saved since most prefixes have to be generated and tested. However, when the selectivity is low, i.e., many episodes do not satisfy the constraints, significant gains can be observed for both PPS and EPT.

To summarize the above experiments, we can safely conclude that PPS outperforms EPT and MINEPI by a large margin. PPS achieves good execution times with varying minimum support, length of sequence and length of frequent episodes. PPS also occupies the least memory due to the DFS search strategy despite the entire sequence in memory. The memory consumption of PPS can be further improved by storing unused position-pairs on disk. The combination of lower memory requirement and faster execution time would make PPS even more suitable for mining huge amounts of sequence data.

3.10 Summary

In this chapter, we have shown two algorithms on StegDB, Episode Prefix Tree (EPT) and Position Pairs Set (PPS), to mine repeated patterns in the page access sequence on outsourced storage efficiently. Both algorithms allow a systematic way to push various constraints into the mining process, which help to locate suspicious patterns more quickly. Through running our mining tools on real page access sequences, we captured, on average, over half of the actual patterns which are useful for an adversary to locate the hidden partitions.

Chapter 4

Masking Access Patterns in Untrusted Page Storage

As stated in Section 2.1.1, hidden patterns are stored as hidden file on the StegFS file system[5]. Blocks of hidden patterns are marked as dummy blocks and scatter around the encrypted storage. In Chapter 3, we describe data mining algorithms that the attacker might mine frequent patterns on access sequences. The regular patterns on dummy blocks might hint that those dummy blocks are parts of a hidden partition.

Those reference patterns could be masked by transforming the access sequence. Following that, we define the security metric which evaluates the effectiveness of the transformations. We then proposed two masking schemes to transform the access sequence. The experiment section shows how effective our schemes are.

To raise the difficulty of isolating reference patterns, we propose two countermeasures: 1) relocating pages with the aim of reducing the support of patterns, and 2) prefetching random parts of reference sequences with the aim of disturbing the sequentiality in reference sequences. We will discuss them in detail later.

4.1 Security Metrics

A countermeasure against traffic analysis is to reduce informational patterns. Our proposed schemes achieve this by transforming the original reference sequences so that there will be few or even no valuable patterns left in the transformed reference sequences. To evaluate the effectiveness of the transformations, two quantities are measured.

First, we mine the transformed reference sequence to examine whether any informational reference patterns could be found by an attacker. For this purpose, we use the concept of *recall* from Information Retrieval. It is defined as the number of patterns in the transformed sequence over the number of patterns in the original sequence, where the patterns are obtained through running the sequential mining tool that we reported in Chapter 3. A better countermeasure should expose fewer patterns, i.e., has a lower recall. The recall is, however, not an objective measure as it depends on the different constraints and minimum frequency specified in the mining algorithms.

For a more objective measure, we calculate the correlation between the transformed sequence and the original sequence. Ideally, the transformed sequence is always independent of the original sequence, and no information will be leaked by monitoring the transformed sequence. The concept of correlation here, however, cannot be obtained using the Pearson's correlation coefficient that measures only the linear dependence among numerical sequences. Instead, the *mutual information* function from Information Theory, which measures correlation by calculating the information shared between two symbolic sequences [34], is more appropriate here. The *mutual information* vanishes if either 1) the sequences have no entropy or information to share, or 2) the variables in the sequences are statistically independent. The first case means that the sequence is fully predictable. As an example, the page server reads the whole disk from start to end repeatedly; no matter serving which pages, the page server always reads pages in this fixed sequence. This example achieves independence between the sequences, at the price of high I/O overhead. In general, transforming an irregular symbolic sequence to a fully predictable one would inevitably incur the penalty of high I/O overhead. The second case requires that sequences be statistically independent. Unfortunately, this is not easy to determine precisely given the large number of different symbols (disk pages). Considering that a truly random sequence is statistically independent of any other sequence, an alternative way is to measure how random the transformed sequence is. Specifically, the more random the transformed sequences, the fewer patterns could be found. For this purpose, we use the *conditional entropy* from information theory to quantify the randomness of a sequence [35].

Definition 1 Conditional Entropy*. The conditional entropy tells how much uncertainty remains in a sequence of events E after we have seen subsequence s. The conditional probability of an event e based on a context s is written as p(e|s). For an event set E, the conditional entropy of a sequence is:*

$$H(E \mid C) = \sum_{s \in C} p(s) \sum_{e \in E} p(e \mid s) \log_2 \frac{1}{p(e \mid s)}$$

where C is the set of all possible contexts. The maximal length of subsequences in C is the order of conditional entropy. According to the definition, the presence of patterns increases $p(e|s)$, thus decreasing the overall conditional entropy. In other words, the higher the entropy, the higher the uncertainty contained within the sequence, which means less discernible patterns. For simplicity, the entropy as mentioned in the rest of this chapter refers to conditional entropy.

4.2 Counter Measures for Page Reference Analysis

Having highlighted the security threats from database page reference patterns, we now present two countermeasures to protect an encrypted DBMS against such attacks. Since we expect an attacker to look for repeated page patterns at the storage server, we aim to raise the difficulty of isolating genuine reference patterns.

4.2.1 Sequentiality-Aware Page Relocation

Conceptually, if data pages are relocated on every read or write access, there would be no repeated patterns in the server access activity. Moreover, if the attacker cannot track the location changes, no pattern would be inferred from the reference sequences. Yet this is impractical from a performance perspective because it doubles the number of I/Os. As a compromise, we relocate data pages probabilistically - each time a page is accessed, it has a probably p of being shifted to a new location within the storage volume. Since longer patterns may contain more information than shorter patterns and are easier to discover (even if part of it has been relocated), pages in longer reference patterns should have higher p than those in shorter patterns. For this reason, we introduce the concept of subsequence masking to indicate that a pattern is entirely relocated.

Definition 2 Subsequence Masking. *A subsequence is masked if at most one of its items will appear the next time the same subsequence is retrieved. For example, a subsequence (abc) is masked if only one of {(a), {b),{c}}, or even none of them, appears in the relocated reference sequence when the same logical pages are requested again.*

Suppose the probability of a subsequence with length n being masked is S_r. According to the definition, the below inequality must be satisfied:

$$\binom{n}{1} \cdot p_s^{(n-1)} \cdot (1 - p_s) + p_s^n \geq S_r$$

where p_s is the relocating probability of pages in this n-length subsequence, and thus depends on S_r and n. Inequality 2 means that, to mask a subsequence with a probability of S_r, pages in this subsequence should be relocated with a probability of p_s. Furthermore, even if a page is not in a subsequence, a single highly referenced page will also raise suspicion. Hence, on top of sequentiality-aware relocation probability p_s, we also relocate single page with probability P_r to level the frequency distribution of individual pages. The final relocation probability p is the probability of p_s or P_r, i.e. $p = 1 — (1 — p_s) \bullet (1 — P_r) = p_s + P_r — p_s \text{-} P_r$. Note that S_r and P_r are tunable parameters and may be increased to satisfy higher security requirements. Combining with the above inequality, we have the following inequality:

$$n\left(\frac{p - P_r}{1 - P_r}\right)^{n-1} - (n-1)\left(\frac{p - P_r}{1 - P_r}\right)^n \geq S_r$$

The final relocation probability of a page, namely *p,* in subsequences with length *n* is shown in Figure 4.1. One can observe that pages in longer subsequences have higher overall relocation probability *p.* The overall relocation probability *p* depends on the single page relocation probability P_r and the subsequence masking probability S_r. To illustrate, suppose P_r is 0.2 and S_r is 0.3, 40% of pages are in subsequences with average length 3 while the remaining 60% are single pages. The average relocation probability is 0.49 * 40% + 0.2 * 60% = 31.6%. That is, on average, a page would be accessed 3.165 times before being moved, and the I/Os for writing data pages to their new locations are expected to add 31.6% overhead to the server load. Note that, for any given page, the relocation rate may vary each time according to which subsequences it is in.

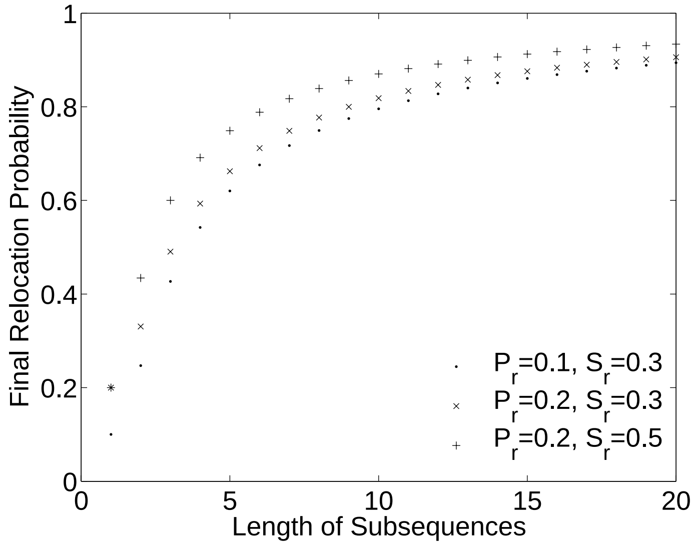


Figure 4.1 The relocation probability with various subsequence length.

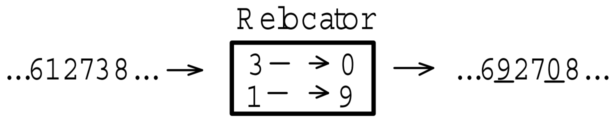


Figure 4.2 Relocating.

To relocate a data page, a free page is randomly selected within the storage volume, and swapped with the data page. The table header and page allocation map are then updated accordingly. Since the header of active tables and the page allocation map are always placed in the cache, the overhead in updating them will not add significantly to the response time. Figure 4.2 shows an example of how page relocation could change the page access patterns between consecutive retrievals, on a table occupying pages (P6, P1, P2, P7, P3, P8). After the table is scanned the first time, P1 and P3 may be relocated, so the page sequence becomes (P6, P9, P2, P7, P0, P8) when the DBMS scans the table again. After that, the next page sequence could be (P6, P9, P3, P7, P0, P5). Hence it is unlikely that an attacker would be able to spot the correlation between the 3 sequences from the activity log on the page storage. The above probabilistic page relocation mechanism achieves the twin effect of breaking a long database reference into shorter page sequences, and reducing the number of repetitions for each page sequence. The experiments in the next section will show that this lowers very significantly the recall of transformed sequences. The downside of page relocation is that it is no longer possible to tune system performance through clustering, as data pages in a table get scattered over time.

The page swapping operation treats page-read and page-write requests differently. For a page-write request, the swap operation writes the content of the original page to the new location directly. In the case of a page-read request, the swap operation reads in the original page, and then writes the content out to its new location. During the swap operation, the following two facts make sure that it is impractical for the attacker to track locations changes. First, the pages are re-encrypted before being written back to the storage volume. Each page contains an initial vector (IV) and a data field. The data field contains real data in the case of a data page, and random bytes if it is a free page. For each page, its data field is encrypted using a CBC (Cipher Block Chaining) block cipher with the IV as seed. Whenever a page is re-encrypted, its IV is reset so that the content of the whole encrypted page changes. For an attacker without the encryption key, it would be impractical to identify the location changes by comparing the binary content of the pages. Second, a relocated page may not be written out immediately after the read operation that causes the relocation. Instead, it is likely to occur several page operations away as the DBMS rarely sends page requests one by one. For example, from the following sequence: 'read(Pl),read(P3),write(P2),write(p4)' the attacker can only deduce that the page P1 may be relocated to P2 or P4. As pages get relocated continually, it would be impractical for the attacker to trace all the possible relocations.

To reduce the performance impact of page relocation, our relocation strategy can also be integrated with the database buffering strategy. Specifically, pages are written back to disk only when they are evicted from the buffer. The benefits of such a delayed write-back strategy are two fold. First, some of the pages would be dirty when they are written back. Thus, delayed write-back avoids writing back those pages twice as with an eager write-back strategy (i.e., pages that are read and immediately written out could be modified in the buffer and hence need to be written back again). Second, it raises the difficulty of tracing location changes because the relocation is delayed until the page is evicted.

4.2.2 Out-of-Order Page Prefetching

The method of relocating pages incurs significant overhead in I/O throughput. In addition, they cannot change the sequentiality of the reference sequences the first time they are issued. To complement page relocation, knowledge of impending I/Os could be exploited to mix up the reference patterns before they are issued to the page server. In this section, an out-of-order prefetching technique is introduced to disturb the sequentiality of reference sequences further.

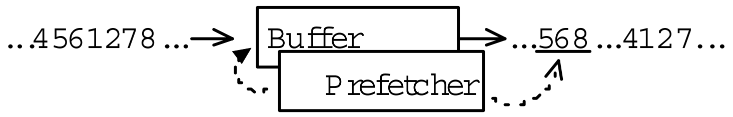


Figure 4.3 Prefecthing.

The out-of-order prefetching technique splits a predicted page sequence into several parts, and randomly prefetches some of them into the database buffer beforehand. Later, as parts of this sequence are already in the buffer, the database buffer subsystem would issue only the unbuffered parts of the sequence to the page server. Thus the page sequence appears in the final masked reference sequence as several separate parts instead of as a whole sequence. For example, as shown in Figure 4.3, P5, P6 and P8 are prefetched when the page sequence {PA, P5, P6, P1, P2, P7, P8) is predicted. The actual issued reference sequence to the page server thus becomes {PA, P1, P2, PI). At a later time, when the same page sequence is predicted, P6 and P2 may be prefetched. Hence, it is unlikely that an attacker would be able to determine that the same page sequence has been requested twice.

The out-of-order prefetching technique effectively splits a page reference pattern into several parts even if it is issued for the first time. It leaves few clues about repeated reference patterns with negligible impact on I/O performance. However, prefetching is not without restrictions. Limited by the size of database prefetching buffer, it cannot prefetch page requests of a predicated reference page sequence too early, or else the prefetching buffer may overflow. In addition, the hint from the query optimizer may not be available very early before the access sequence is issued. As a result, a powerful mining tool may still observe that the reference sequences are clustered. Continuing with the aforementioned example, the attacker may observe that pages in the set {P1,P2, P4, P5, P6, P7, P8} are often referred near to each other. To alleviate these problems, it is necessary to combine the prefetching countermeasure with the relocation strategy.

4.3 Implementation and Evaluation

This section begins by describing a prototype implementation of the proposed countermeasures, and then discusses representative results from an experimental study conducted with the prototype.

4.3.1 System Implementation

To evaluate the effectiveness of the two proposed schemes, we have implemented them in MySQL [30] as depicted in Figure 4.4. The page I/Os of MySQL are passed to a buffer module which returns hit pages immediately. The OOP (out-of-order prefetching) module randomly prefetches segments of a sequential page sequence into the buffer when the page sequence is predicted by the database engine. All normal and prefetched page requests are then redirected to a page relocation module, which decides whether to relocate the page or not. The relocation module, which keeps a map from original page locations to relocated locations, maps pages to their real locations and forwards the requests to the external page storage which is situated on a remote machine.

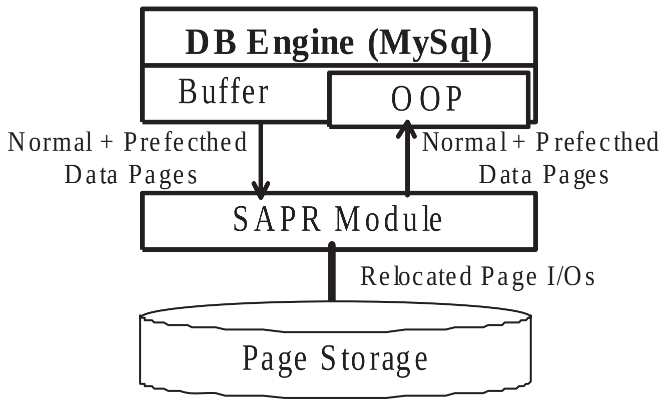


Figure 4.4 System implement.

When a requested page is received, the page relocation module maps the relocated page back to its original page. The buffer updates its content according to the page, so that the DBMS receives only legitimate pages. There is thus no need to modify internal mechanisms of the DBMS except that its query optimizer needs to provide hints to the prefetching and relocation modules when a sequential reference sequence is predicted. The three components—buffer, OOP and SAPR can be enabled/disabled separately; when the SAPR detects the existence of a buffer, it automatically uses the buffer as a write-back buffer (see Section 4.1).

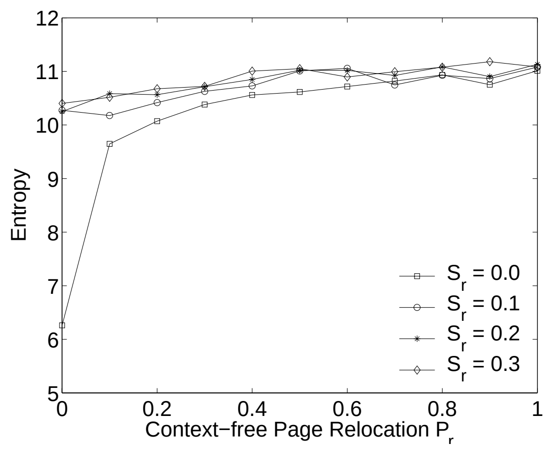
4.3.2 Experiment Set-up

To evaluate the performance of the proposed algorithms, we install the modified MySQL DBMS and the page storage server on two Intel PCs. The logical resource and workload are modeled after a TPC-C benchmark [36], and each trial in the experiment consists of 60000 transactions issued from 10 remote terminals. The database contains 9 base tables, and the workload is made up of a series of 5 different types of business transactions - entering new orders, delivering orders, recording customer payments, monitoring the stock level in a warehouse, and checking the status of an order. To induce sequential IOs, we introduce queries on unindexed fields which lead to full table scans.

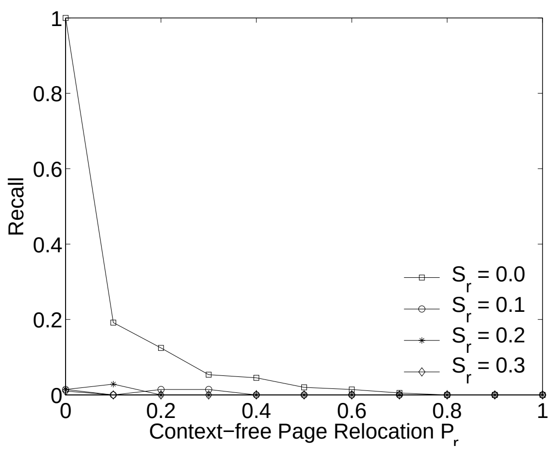
Besides the security metrics, recall and entropy measured in the third order (see Section 3.3), I/O overhead is used to measure the efficiency of the schemes. The latter is defined as the total number of page I/Os from page relocations, divided by the number of data pages requested by the DBMS. As sequential I/Os are much cheaper than random I/Os, and sequential I/Os are the kind of I/Os that need to be masked most, the experiments are based on two sequential workloads.

4.3.3 Experiment with High Sequentiality Load

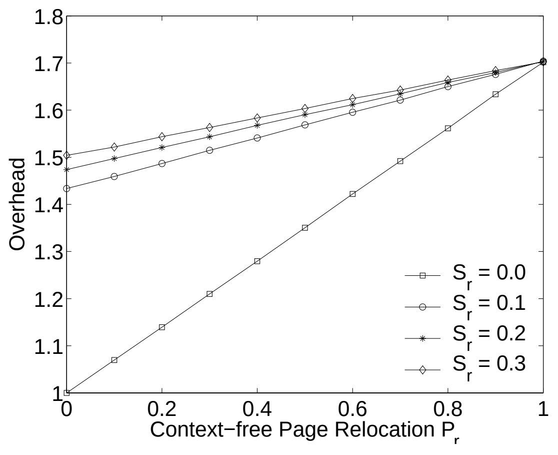
The first experiment is designed to study the various schemes under high sequentiality load conditions. We remove some indice in the standard TPC-C benchmark to make about 80% of queries causing table scans in the system. This represents the worst case scenario as there are lots of sequential references and hence it is much easier for an attacker to detect repeated patterns from the page I/O activity log.



(a) Entropy with page relocation rate.



(b) Recall with relocation rate.



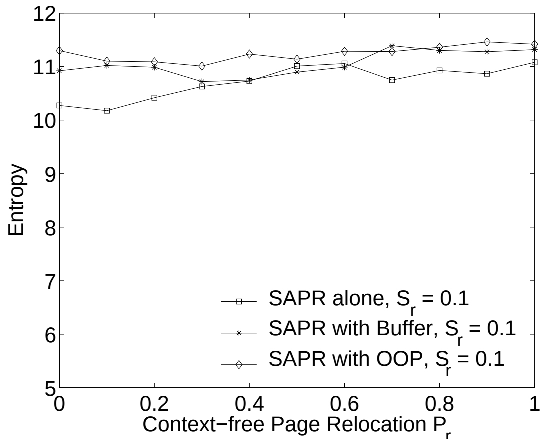
(c) Traffic overhead with page relocation.

Figure 4.5 Effect of SAPR without Buffer.

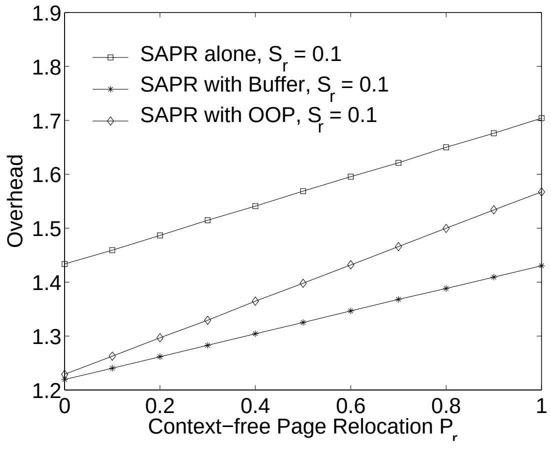
Figure 4.5 (a)-(c) plot the entropy, recall and overhead with the SAPR scheme alone. Note that, as illustrated earlier in Figure 4, the final relocation rate for a single page is the sum of context-free page relocation rate (Pr) and sequence relocation rate (Sr). To illustrate the effect of sequence relocation we use four different Srs ranging from 0 to 0.3. Figure 8(a) shows that when Sr = 0, the entropy increases quickly for Pr smaller than 0.3, and then it rises slowly after that point, while the entropies with Sr > 0 start with a high value, around 10.2, and grows slowly over time. Note that the upper bound of entropies depends on how many different pages occurred in the transformed sequences. In Figure

8(b), the recall with Sr = 0 decreases drastically with the increase in page relocation and reaches 0 when page relocation rate exceeds 0.8; while the recalls with Sr > 0 start low and approach 0 quickly. The above two results confirm our expectation that the relocation scheme effectively removes reference patterns contained in the page I/O activity log. Comparing the context-free page relocation and sequentiality-aware relocation method, the latter is more effective in removing repeated patterns. The overhead result is shown in Figure 8(c). Schemes with Sr > 0 have a higher overhead than that with Sr = 0 because, in high sequentiality case, the final page relocation is dominated by Sr which is high especially when handling long sequential patterns. Note that the maximum overhead does not reach 2 when Pr reaches l. The reason is that, unlike the relocation for page-read requests, relocating a page-write request does not cause two disk I/Os—the page is just redirected to another location in the page storage volume.

To study the effect of SAPR with a write-back buffer and SAPR plus prefetcher, we compare them with SAPR in Figure 4.6(a)-9(b). As shown in Figure 4.6(a), the entropies of SAPR with buffer and SAPR plus prefetcher are on average a little higher than that of SAPR alone, which means the introduction of buffer and prefetcher increases uncertainty in the traffic. Figure 4.6(b) shows the traffic overhead is slightly reduced with a buffer size that is around 2% of the entire disk pages. We do not show here the comparison on recall which is near to zero in each scheme.



(a) Entropy with relocation rate.

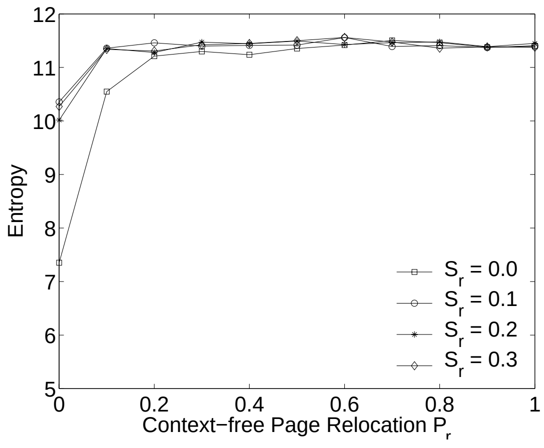


(b) Traffic overhead with page relocation.

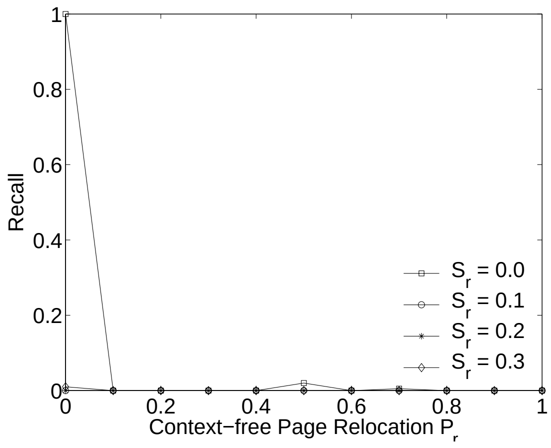
Figure 4.6 Effect of Buffer and Prefetcher.

4.3.4 Experiment with Low Sequentiality Load

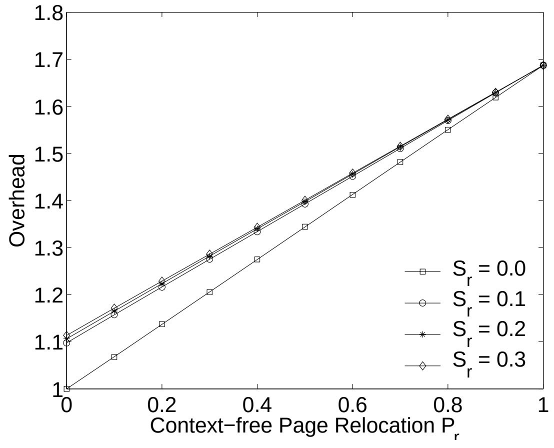
Next, we experiment with low sequentiality loads by removing only one index from the standard TPC-C benchmark, which causes a small fraction of sequential queries. The results, given in Figure 4.7 and Figure 4.8, exhibit similar trends as those in the high sequentiality load scenario. Therefore, we focus only on the differences next. In Figure 4.7(a), the average entropy is a bit higher than that for the high sequentiality workload as the ratio of random I/Os is much higher here, which leads to higher uncertainty. In addition, entropies for $S_r > 0$ increase quickly when P_r goes from 0 to 0.1. A possible reason is that random IOs may also present a weak correlation which is disrupted when $P_r > 0$. In Figure 4.7(c), the overhead of the SAPR scheme is less than that in the high sequentiality load scenario because the sequential reference patterns, which lead to high S_r, only occupy a small fraction of the entire reference sequence. Figure 4.8(a) and Figure 4.8(b) convince us that the write-back buffer and OOP add uncertainty in the page traffic, although they contribute less here as compared to the high sequentiality workload scenario in the previous experiment.



(a) Entropy with page relocation rate.

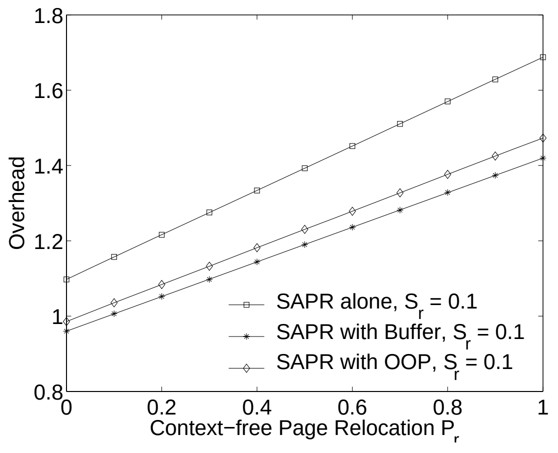


(b) Recall with relocation rate.

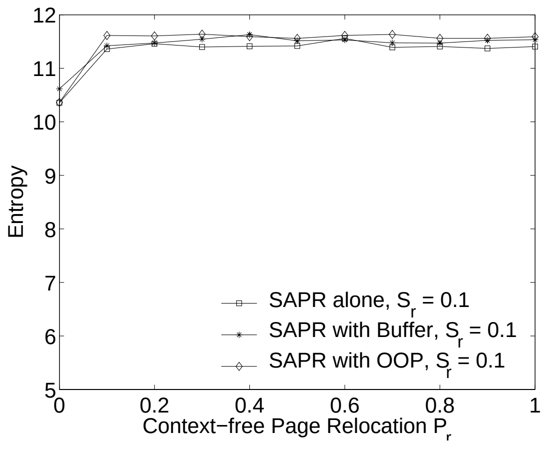


(a) Traffic overhead with page relocation.

Figure 4.7 Effect of SAPR without Buffer.



(a) Entropy with page relocation rate.



(b) Traffic overhead with page relocation.

Figure 4.8 Effect of Buffer and Prefetching.

Summary of Experiment Results. From these series of experiments, we arrive at the following observations:

- Natural page I/Os from concurrent jobs do not disperse the page reference patterns adequately; the buffer alone cannot wipe out all patterns either. Hence a DBMS that supports encryption needs to implement specially-designed mechanisms to mask its page references.

- The SAPR strategy is very effective in breaking long reference patterns into segments and in reducing the number of repetitions of the segments, thus preventing an attacker from assembling all the pages of a database object (e.g. a table) in the correct order. A set of recommended parameters based on our experiment is $S_r = 0.1$ and $P_r = 0.1$. Comparing sequentiality-aware relocation with context-free page relocation, the former targets sequential sequences specifically and is thus more cost-effective.

- SAPR with write-back buffer increases the difficulty in tracing the page location changes; OOP then further weakens the transformed sequence, making it more random.

4.4 Summary

In system model 1, the storage of StegDB migrates from devices that are local to individual computers, to shared storage volumes that are accessible over open network. This exposes the data to heightened security risks, because databases exhibit regular page reference patterns that can easily reveal the logical links among blocks which contains the hidden partition. With this knowledge, an attacker can easily spot the location of a hidden partition, and thus disclose its existence.

To mitigate the risk of attacks initiated through analyzing the shared storage server's activity for page patterns, we introduced algorithms that employ data relocation and out-of-order page prefetching techniques. Experiments showed that data relocation is effective in breaking long reference patterns and thus reducing the number of times that a pattern is repeated. We also studied the interplay between data relocation, buffering and prefetching, and observed that buffering and prefetching further increase the difficulty for the attacker in tracing page relocations and spotting the reference patterns. Note that there are minor differences for different database buffer replacment in masking page patterns.

Chapter 5

Secure Coprocessor-Based StegDB in Untrusted Database Engine

In system model 2, the untrusted environment where the database engine and page storage locate poses severe security challenges of protecting and hiding the potential sensitive data, as the adversary could break-in and compromise the system from several places. In this chapter, we propose the use of secure coprocessors to address these issues and thus make model 2 more attractive and practical. In this chapter we introduce first the secure coprocessor, and then give an architecture built upon secure coprocessors. Under this framework, we re-examine the security requirement of a StegDB and proposed a query decomposition algorithm. Finally, we study the traffic pattern threat, and give a two-zone solution which is based on probability model.

5.1 Secure Coprocessor

Secure coprocessors are tamper-proof sealed general-purpose computer. They are protected so that any attempt to penetrate them will result in all critical memory being erased. Smith and Weingart showed how to build a generic secure coprocessor platform [37]. This research culminated in the family of commercially available devices, the IBM

Cryptographic Coprocessor [38]. These devices feature—in a PCI form factor—physical and logical security protection validated at FIPS (Federal Information Processing Standard) 140-1 Level 46, as well as hardware 3DES and SHA.

A SC can be installed on a standard computer to provide a secure perimeter wherein sensitive data may be stored and processed undisturbed, even if the adversary has direct physical access to the device. In StegDB context, SCs are placed around the database server to provide a physically secure (tamper-detecting/-responding) premises where security-critical operations such as the execution of cryptographic protocols and access control decisions are executed.

Since a SC has access to the encrypted page storage, it can decrypt the client's query and run it over the client's data with no real assistance from the server. However, as a SC usually has rather small storage and a limited processor, it is impractical to fit the whole database in. Additionally, the shared data that are queried by different protected queries might not need the same level of protection as that in hidden partition. Therefore, it still appears necessary to engage the untrusted server in query execution. This chapter demonstrates our efforts to bootstrap security properties from the secure coprocessor to the entire unsecured system. The challenge is in hiding from the untrusted server the existence of hidden partition while providing transparent access to it for the clients. Also, the performance of query processing becomes a concern. It is important to avoid the relatively high latency associated with each client query. The secure co-processor has limited resources, which is why it has to farm out as much of the query plan to the open processor as possible, and even curtail the types of query operators that can be executed in the co-processor. That being the case, the co-processor is not likely to be able to run concurrent queries.

5.2 StegDB Architecture Based on Secure Coprocessor

Having highlighted the security threats from the untrusted environment, we now present a framework to protect hidden partitions against such attacks. **Figure 5.1** depicts our efforts to bootstrap security properties from the SC to the entire unsecured system. In this framework, database engine are divided into ODE (Open Database Engine) which takes full charge of the shared data and HDE (Hidden Database Engine) which mainly operates on hidden partitions. A HDE run on a SC to prevent the adversary from observing its internal operations. To hide the existence of hidden partitions, they are stored as hidden files in a StegFS file system [5] which restrict the access only to authorized users, e.g. HPK holders. Note that the shared data and hidden data are separately stored. Arrows in **Figure 5.1** present the dataflow when an user sents a query. We will describe them shortly.

The basic workflow of the framework is as follows. A SC receives a query request from a user. If the query does not involve any online hidden partition, it is directly routed to ODE; otherwise, the query might be split and executed in HDE and ODE separately. The SC receives intermediate results from ODE and HDE and returns the combined and post-processed results. For ease of explaining the framework, in the following section we will show how to access hidden partition and discuss queries processing.

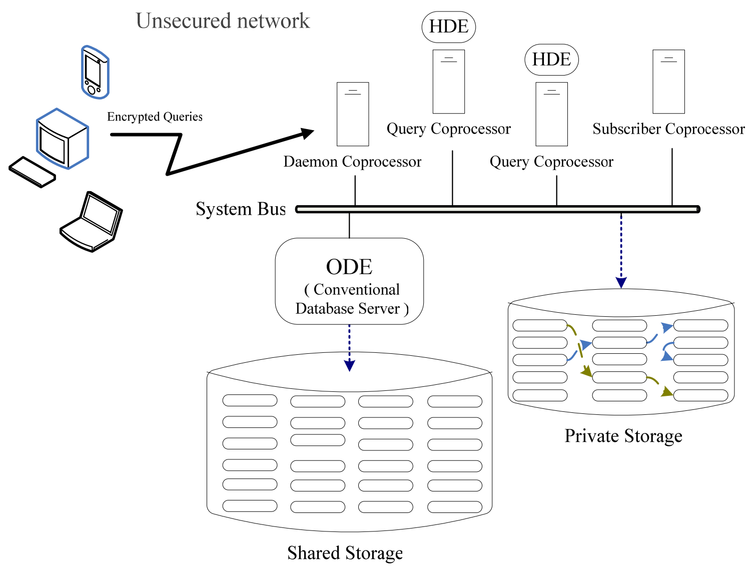


Figure 5.1 Architecture.

5.3 Accessing Hidden Partitions

When a user activates his hidden partition, he first requests the certificate of public key of a coprocessor which is digitally signed by a key management center. The user's HPK is then encrypted with the public key and transferred to the coprocessor. The HPK is stored only within the secure non-volatile memory of the coprocessor. The coprocessor then negotiates with the user for a session key which is used to encrypt the query statement and query result.

The HPK is finally passed to the StegFS system [5] for accessing specified hidden partition. This kind of file system grants access to a hidden partition only if a correct HPK is supplied. Without them, an adversary could get no information about whether the protected partitions ever exists, even though he understands the file system completely, and is able to scour through the content on the raw storage. In concrete terms, the data blocks of hidden partitions, which are encrypted, are scattered across the storage space in such a way that they can only be located through hashing HPKs. Therefore, an attacker without the HPKs cannot distinguish between data blocks of hidden partitions and dummy blocks, and thus cannot deduce the existence of the hidden partitions.

Meanwhile, a traffic masking system proposed in our previous work is also deployed on the top of the StegFS system to mitigate the risk of attacks initiated through analyzing the access sequences for access patterns. The masking system has two strategies: 1) the first relocates data pages according to which page sequences they are in; 2) the second enhances the first by randomly prefetching pages from predicted page sequences. The StegFS and traffic masking system both run inside SCs.

Proof of plausibility. Under the StegFS, data blocks are the scattered around the storage and look the same as dummy blocks. Accessing a hidden partition is thus almost the same as accessing a dummy sequence. Even if a hidden partition is repeatedly accessed, an adversary is not able to observe repeated patterns in the access sequence because of the relocation strategy. In particular, pages currently accessed are more likely relocated to other locations so that the access subsequence on the hidden partition is probably entirely different next time. Therefore, when a hidden partition is being accessed, there is no convincing evidence for an adversary that current accessed blocks are parts of a hidden partition. The chances of an adversary to successfully spot the hidden partition are thus the same as random guess, which satisfies Definition **2**. In other words, queries on the hidden partitions are plausible.

Note that, after the users activate their hidden partitions, all updates go to those hidden partitions instead of their shared counterpart. Otherwise, an unauthorized user may observe the sensitive data caused by incautious updates (see also *-property of [11]).

5.4 Reconciliation Operation

Before delving into the decomposition problem, let us first examine the polyinstantiation in StegDB. There are two types of polyinstantiation: polyinstantiated tuple (or polyinstantiated entity) and polyinstantiated attribute. The polyinstantiated attribute is used to model a same real-world entity, an attribute of which has different values in shared data and hidden partition. The polyinstantiated tuple was introduced to model two or more different real-world entities with the same primary key value. For example, an authorized user inserts a tuple into his hidden partition before an unauthorized user inserts a new tuple into the share database without knowing the existence of the duplicated one. In this case, the two tuples probably present two different entities in reality. In another case, the authorized user insert a tuple into his hidden partition to record the secret attributes of tuples which are already in shared database. Here, the tuple in shared database is only a cover story of its counterpart in hidden partition. To distinguish the two cases, a simple solution is to allow the user to specify which types of polyinstantiation the tuples are when creating a hidden partition.

As the existence of polyinstantiation, there is a possibility that conflicting tuples are found when combining the intermediate results from ODE and HDE. To this end, we introduce the reconciliation operation ⌂ between a hidden partition and its associated shared table.

Reconciliation Operation: a operation used to reconcilate the conflicted tuples from shared table and hidden partitioin. Concretely, if the conflicted tuples are polyinstantiated tuples, the reconciliation operation gives user both conflicting tuples; if the conflicting tuples have polyinstantiated attributes, it shows the user the tuple from the hidden partition. To find conflicted tuples, the reconciliation operation needs to compare the primary keys of potential conflicted tuples.

Distributive property: a reconciliation operation is able to distribute over join operations, e.g. A⊳⊲(B⌂C) ≡ (A⊳⊲B)⌂(A⊳⊲C). *Proof.* Let A_k denote the primary key of B and C. Suppose that a set of tuples CT_B in B conflict with tuples CT_C in C on primary key values $CK = \{k_1, k_{2,} ..., k_n\}$. CT_B develops into the tuple set $CT'_B = \{t \mid t.A_k \in CK\}$ in join result of A ⊳⊲ B while CT_C develops into the tuple set $CT'_C = \{t \mid t.A_k \in CK\}$ in join result of A ⊳⊲ C. By checking A_k of join results, a reconciliation operation can still find that CT'_B and CT'_C conflict with each other. □

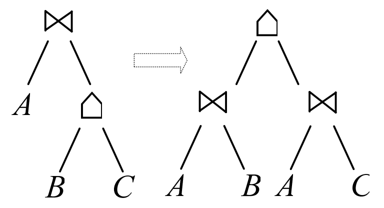


Figure 5.2 Distributive property of reconciliation operation.

5.5 Query Decomposition

There are two major security considerations for queries sent to ODE during the query decomposition procedure. 1) The queries are standalone, not appealing to be parts of compound queries. For example, queries should only access the shared data. 2) The queries should avoid alerting an adversary to the potential sensitive information. For example, given the same table as Tables 2.1 and 2.2, the query "*select tid from task where objective = 'Assassination'*" might not decomposed as two same queries sent to ODE and HDE separately as the predicate "*objective = 'Assassination'*" will reveal the intention of users who query information contained only in hidden partition.

In addition to those security considerations, it would also be better to decompose a query in a way that as few parts as possible are executed on a SC as its processing limitation. In the following, we will provide insights to the query decomposition process through an example. The complete query decomposition algorithm is provided in Appendix A.

Example 1. *Consider the following schemas: 1) employee(eid, name, age, address), 2) task(tid, objective, duration) and 3) group (eid, tid, role). Among the three tables, the task and group are associated with horizontal hidden partitions. Given the user query, "select name, role from employee, task, group where task.tid = group.eid and group.eid = employee.eid and task.objective = 'Spy'", it can be decomposed in the following steps.*

Step 1. ***Identify hidden partitions***. *During this step, each involved table is examined whether they are associated with hidden partitions. If so, a reconciliation operation is applied on data in shared tables and data in hidden partitions.*

Step 2. ***Process join, selection and projection list.*** *Regard the reconciled result as a base table and apply conventional query optimization techniques, which would lead to a query tree. The result of this step is shown in Figure 5.3.*

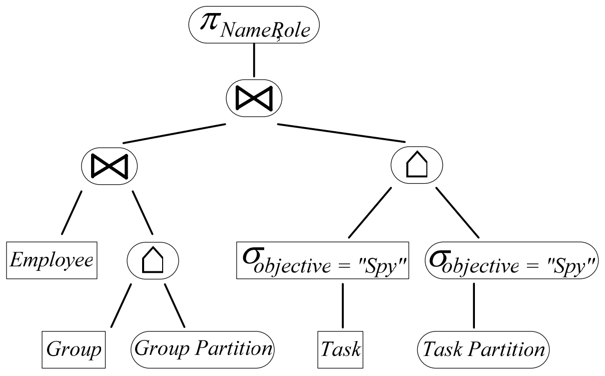


Figure 5.3 Query tree in Step 2 and 3.

Step 3. ***Determine the execution hosts of operation nodes.*** *As the ODE could not see any intermediate result from SC, so every branch of the query tree consists only of two parts: the upper part executed in SC, and the lower part executed in ODE. This requirement gives rise to the following rules:*

- Reconciliation operations are only executed in SC.
- Operations directly accessing hidden partition are executed in SC.
- An operation is executed in HDE if it has a descendant operation node executed in SC.
- The rest operators which have not explicit execution place are executed in ODE.

The result is shown in Figure 5.3 in which nodes within normal rectangles are executed in ODE, while nodes within round-corner rectangles are executed in SC.

Step 4. ***Transform the query tree to shift as much workload as possible to ODE***. *One can observe, from Figure 5.3, that all costly join operations are executed in the resource limited SC. To improve the query performance, the distributive property of reconciliation operations, as shown in Figure 5.2, is applied to transform the query tree so that some parts of join operations are executed in ODE. The transformation is a top-down approach. It keeps pushing down the join operations until some join operations could be executed in ODE without violating the execution host rules mention earlier. The final result is shown Figure 5.4.*

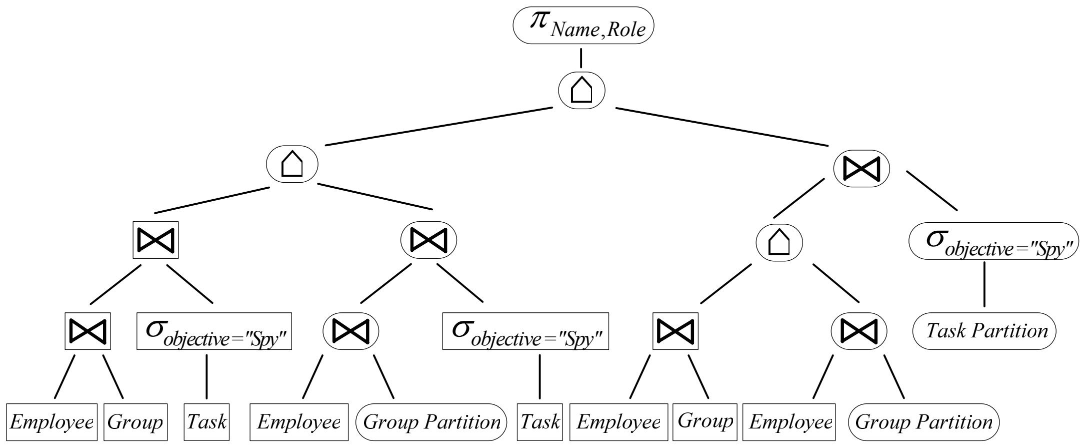


Figure 5.4 Query tree in step 4.

Step 5. ***Adjust the query tree to avoid leaking sensitive selection predicate.*** *In particular, the sensitive operation nodes are moved up in the query tree to right below its lowest ancestor node executed in HDE. Then the operation nodes are changed to be executed in SC. For example, if the select predicate in example query is "objective='Assassination'", we would move the most left selection node in Figure 5.4 one-level up and execute the three predicates in SC.*

Step 6. ***Export the subqueries from the query tree.*** *In this step, we transfer all sub-trees executed in ODE back into query statement. Note that all subqueries sent ODE should be rewritten to include the primary key for reconciliation purpose. There are:*

- Q1: select eid,tid,name,role from employee, task, group where employee.eid = group.eid and task.tid = group.tid and task.objective='Spy'.
- Q2: select eid from employee.
- Q3: select tid from task where task.objective='Spy'.
- Q4: select eid, tid from employee, group where employee.eid = group.eid.

The remaining part of the query tree is executed in a SC while waiting the intermediate result from ODE.

Proof of plausibility. According to the decomposition procedure, queries sent to ODE are complete and standalone. Besides, as an adversary could not probe internal operations in SCs, he is not able to confirm his suspicions that some queries are parts of compound queries. For some compound queries involving multiple tables, the decomposition procedure generates a group of queries than one queries which often cluster together. In addition, as the result of decomposition is fixed for a compound query, an adversary might observe some queries in ODE often appear repeatedly and cluster together. Those observations however could not support the inference of the existence of hidden partition, as many database applications often issue fixed groups of queries to the server. So the decomposition procedure is plausible.

Note that the decomposition procedure does not satisfy the indistinguishability requirement in Definition **1**. In particular, after the adversary observes a suspicious query, he reissues the exact query as a normal unauthorized database user, not involving hidden partitions this time. As the second query is directly routed to ODE, he will observe queries executed in ODE is different from previous observation, which discloses that the previous query is a compound query.

Discussion. Parts of the query decomposition are similar in that of a multidatabase system [39]. One can envision the ODE and HDE are autonomous local databases. The plausibility requirements of StegDB, however, make the difference. According to the requirement, we analyze the execution host of each operation. In addition, as the limited resources of a SC, we transform the query tree to optimize the execution cost. The transformation procedure, however, may result in heavier data communication between the SC and the ODE. For instance, the intermediate result size in the final step of above example is bigger than that in Step 2. Taking the high data bandwidth between the SC and ODE—often a pci bus—into account, it still profitable to shift parts of join operations to ODE.

5.6 Hiding the Interplay between Open Partition and Private Storage

In previous sections, we discussed how individual components of StegDB provide plausibility for client queries. Let us continue with the task table in Tables 2.1 and 2.2. If a user issues a query "*select * from task*" repeatedly, from the side of private storage an adversary can only see a random-looking access sequence because of the relocation strategy; from the side of ODE he can only see a normal query because of the query decomposition procedure.

However, things become complex if the adversary combine his observations at both sides. At the open processor, as well as the private storage, an adversary can see which SC issues the query requests. That enables the adversary to co-relate a query to the open processor to I/O operations within private storage. He might be able to infer the existence of hidden partitions through co-relating the queries in ODE with access sequences on private storage. To be specific, the adversary records a group of accessed blocks in private storage, called *access set*, when a query or a group of queries in question are being executed in ODE. Note that it could be possible that the start and end of this group of accessed blocks are not exactly the same as that of the queries in ODE since the existence of dummy accesses. It is, however, enough for the adversary to identify the block sequence. As all queries need to be responded as soon as possible because of the nature of a database system, the SC has not much buffer time to delay or advance the access on hidden partitions. In addition, the SC needs to send back query result which also hints the end of a query.

5.6.1 Repeated-access-attack on Relocation Schemes with Random Dummy Blocks

Existing practical schemes for hiding access patterns are based on block relocations. In concrete terms, they relocate accessed blocks to new locations so the old location would not appear in access sequence when the same block is accessed again. With access sets, the effect of relocation schemes is described in Figure 5.5, where the big empty circles present access sets, and the small solid circles are requested blocks which are relocated after they are accessed.

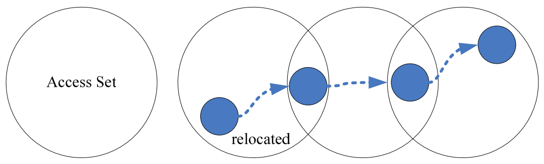


Figure 5.5 Intersection of continues read/write sets.

The intersection between two consecutive access sets of the same query (or same group of queries), defined as IAS, however, may reveal the underlying hidden partitions. The reasons are as follows. The size of IAS might be relatively large when a hidden partition is accessed repeatedly during each run of queries. In those relocation schemes all blocks are however marked dummy and accessed in a random manner, so that large IASs do not occur often in an assumed random access sequence.

To check whether a particular IAS is caused by two random access sequences which has length *L*, an adversary first assumes that the two access set are entirely caused by dummy random accesses. That is, each block in the storage has equal chance to be accessed. Considering that there are repeated accesses during each run, the expected size of the access set is smaller than the access times during this run. In general, access blocks *m* times randomly within a disk volume with size n, the expected size of access set is a function of *m* and *n*, depicted as

$$\mathrm{E}(n,m)=\sum_{i=0}^{\min(n,m)}\left(i\cdot\binom{n}{i}\cdot\sum_{j=0}^{i}\left((-1)^{j}\cdot\binom{i}{j}\cdot(i-j)^{m}\right)\right) \qquad (1)$$

Then he would calculate the probability function of the size of intersection as follows:

$$P(I=i)=\begin{cases}\dfrac{\dbinom{K}{i}\cdot\dbinom{D-K}{K-i}}{\dbinom{D}{K}} & i\geq 2K-D \\ 0 & i<2K-D\end{cases} \qquad (2)$$

where D is the total number of blocks in private storage, *K=E(D,L)* is the size of suspicious access set. If the size of IAS, as shown in Figure **5.5**, are far large than that in assumed random case, which is highly unlikely to happen, the adversary would reject his random hypothesis and infer that the intersections are probably caused by repeated access on the same hidden partitions. In another words, he infers that the current user with a high probability has hidden partitions, which violates the plausibility requirement of StegDB. Consequently, the possibility of the attacker infers the existence of hidden partitions is directly related to the size of IAS. In the following sections, we will check the viability of random hypothesis against several relocation-based schemes, and take a closer look at the IAS problem.

5.6.1.1 Sequentiality-Aware Page Relocation

The basic idea of our page relocation schemes in Chapter 4 is to relocate data pages according to which page sequences they are in. Pages in longer page sequences are more likely to be found in next occurrence so they will be relocated more frequently. Besides relocation, in Chapter 4, we also proposed a disordered prefetching scheme to further diminish the patterns in access sequence. The prefetching scheme is, however, of little help in reducing the size of IAS because prefetching occurs within each run of queries and does not alter the access set.

The actual size of its access set is $E(D, L + L \times p)$, where p denotes the final relocation rate for each page and n is the number of noise pages caused by inaccurate calculation of the start and end of S. The size of its IAS is a bit bigger than N because the blocks within the hidden partition will be re-accessed in the next run page sequence (some may be relocated), and there might be a few repeated accesses among noise blocks. We now show how an adversary infers the existence of hidden partition through a concrete example. Suppose the total size of private storage D=2000; the relocation rate for pages in a page sequence with length 10 is 0.8. The size of access set is thus $\mathrm{E}(2000,10{+}10{*}0.8{+}10) \approx 28$. The size of IAS is roughly equal to the size of the page sequence 10. With the adversary's random hypothesis and Formula (2), the chance that the IAS is caused by entirely random access is 5.4E-13, which means that it is almost impossible that the intersection is caused by random access. The attacker thus rejects his random hypothesis and immediately infers that the two access sets include hidden partitions, which violates the plausibility requirement in StegDB.

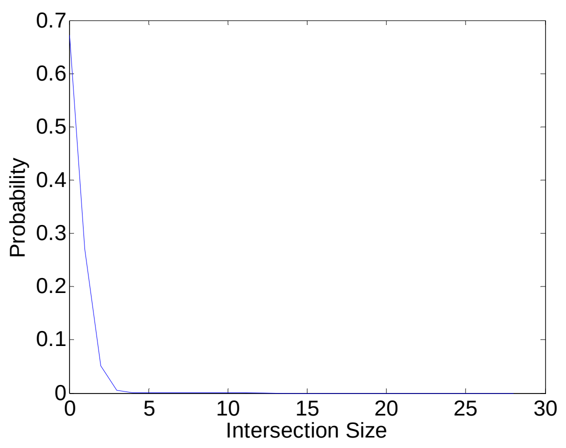


Figure 5.6 Probability distribution with intersection size.

Discussion. Introduction of buffer in this scheme could alleviate the IAS problem by reducing intersection size. With a buffer, parts of page sequence are not written back immediately in the previous run, so less blocks are re-accessed in the next run. Figure **5.6** depicts the probability distribution with intersection size. Continuing with the previous example, if the size of IAS reduces to 2, the chance that two consecutive access sets is caused by entirely random page access is 0.06, which is more likely consistent with the random hypothesis. Increasing the relocation rate *p* might result in an increase in the size of the access set, which might also increase the possibility. It would, however, not increase the possibility much if the volume of private storage is far bigger than the size of the current access set.

5.6.1.2 M-block Relocation Scheme

The basic idea of page relocation in [40] is that each time a block is read, $m-1$ additional blocks are randomly selected, at least one of which is known to be empty. When the blocks are written, the original is written into the empty. All blocks other than the target and the empty one must be randomly chosen from the data storage space so that maximum independence is achieved. m is an adjustable security parameter.

Let us denote the length of a particular page sequence S, which accesses a hidden partition, N. The actual size of its access set is $E(D, mL)$. We now still show how the adversary checks his random hypothesis against this scheme using an example. Suppose the total size of private storage D is 2000, m is 5, and n is 10. For a particular page sequence with N equal to 10, the size of its access set is $\mathrm{E}(2000, 5*10) \approx 49$, and the size of its IAS is roughly 10. With the adversary's random hypothesis, the chance that the IAS is caused by entirely random access is a sufficiently low value, 6.4E-6. The attacker thus rejects his hypothesis and immediately infers that the two access sets include hidden partitions, which violates the plausibility requirement in StegDB.

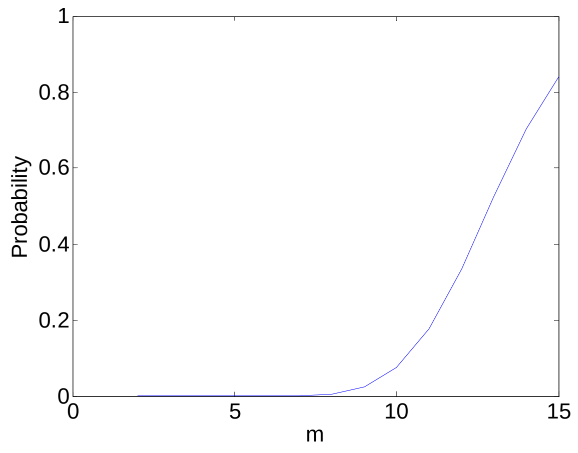


Figure 5.7 Probability distribution with m.

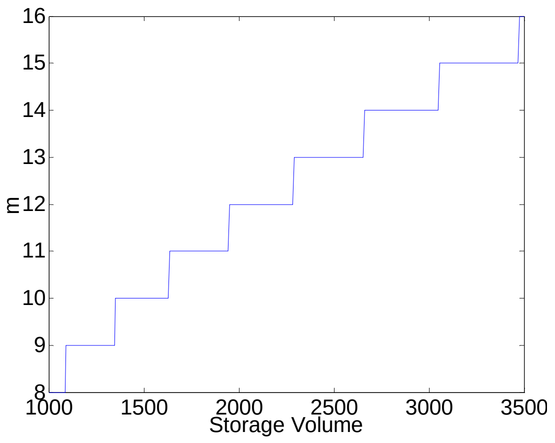


Figure 5.8 m increases with volume.

Discussion. In this scheme, m which denotes the redundancy set size plays a very important role. Increasing m is indirectly increasing the size of access set. When m is big enough, the adversary is not able to convince that the observed access sets are from dummy access or not. Continuing with the previous example, Figure 5.7 depicts the change of probability with m, from which we can see that when m is increased to 15 from

5 the probability rises to 0.84 that is consistent with random hypothesis. However, in order to provide consistent protection, *m* is proportional to the storage volume, as shown in Figure 5.8. As the overhead of this scheme is clearly *m* times more traffic, a big *m* will makes this scheme undesirable in scalability.

5.6.2 Repeated-access-attack Problem in Oblivious Storage

In this section, we will examine the IAS problem against the oblivious storage scheme in [41]. The scheme based on the oblivious ram scheme [19], is made up of a hierarchy of memories. The first level is twice as large as the database buffer, and each subsequent level doubles in size until the last level is big enough to accommodate all the data blocks that could be read by users. To read a data block, the system first looks in its buffer. If the block is not there, the system retrieves it from the highest level in the oblivious storage where it can be found. At the same time, it reads a randomly selected block from each of the other levels. After a data block is read, it is added to the system buffer until it becomes full, at which time all its blocks are flushed into the first level of the oblivious storage, then all the blocks in that level are re-encrypted and re-ordered (shuffled) to an arbitrary permutation. Similarly, when level i of the oblivious storage is full, all its data blocks are flushed into level i+1, and the blocks there are then re-encrypted and re-ordered. Consequently, within each level of the oblivious storage, any given data block will be read at most once before the blocks in that level is re-ordered.

Under this scheme, when a block is written back to top level on disk it is shuffled with other blocks of the same level. When a hidden partition is accessed again, its data blocks must be shuffled (accessed) in the previous run. Therefore, the size of IAS is appropriate to the length of page sequences accessing that hiding partition. Different from previous described schemes, accessing dummy blocks in oblivious storage is not completely random. In particular, as blocks (dummy or data) are selected from each level and the blocks in that level are shuffled in the previous run, the intersection of two different dummy page sequences is also appropriate to their length. The adversary thus is not able to infer whether the current IAS is caused by accessing hidden partition or only random access. So the oblivious storage is free of the IAS problem.

The biggest problem of oblivious storage is that it requires many additional redundant access cycles and frequent storage shuffles. For a normal file system with 20GB disk 20GB and 80MB buffer, the average cost is about 70 times that of a read operation in a conventional file system. In addition, the frequent shuffle operations, especially those occurred in the lower level, which might occurs during querying a hidden partition, leads to an intolerable response time.

5.7 Two-zone Relocation Scheme

Formula (2) shows that reducing the size of the total number of blocks *N* will get a bigger probability. So if we can reduce *N* in some way, we could get a larger IAS for unrelated page sequences. When the IAS of different page sequences has roughly similar probability with the IAS of same page sequences, it is hard for an adversary to distinguish the two. In addition, the design of this scheme should avoid shuffle operations on mass blocks which might be intolerable for a database user.

The bottom line of this scheme is to select random blocks in a much smaller group instead of the entire disk volume. The size of this group is proportional to the length of current access sequence. As the IAS of different page sequences with the same length are comparable within the hot zone, it cannot convince an adversary which IAS is caused by repeated access on same partition. In concrete terms, we split the blocks into two zones: hot zone and cold zone. The length of the hot zone is proportional to the length of the current access sequence. To read a data block, the system first looks in the hot zone. If the block is not there, the system retrieves it from the cold zone. At the same time, it reads *c* blocks randomly from hot zone. To diminish the intersection size, the data block is swapped into the hot zone. When blocks from the hot zone are all data blocks of the current hidden partition, we perform a random swap among those blocks. Otherwise, we switch the data block to one of the block in the hot zone that does not belong to the current hidden partition. If the block is in the hot zone, the system retrieves randomly *c-1* blocks from the hot zone and 1 block from the cold zone. The *c* blocks from hot zone are then randomly swapped.

The page swapping operation treats block-read and block-write requests differently. For a block-write request, the swap operation writes the content of the original block to the new location directly. In the case of a block-read request, the swap operation reads in the original block then writes the content out to its new location. To make sure that it is impractical for attacker to track locations changes, blocks are re-encrypted before being written back to the storage volume. Each block contains an initial vector (IV) and a data field. The data field contains real data in the case of a data block, and random bytes if it is a dummy block. For each block, its data field is encrypted using a CBC (Cipher Block

Chaining) block cipher with the IV as seed. Whenever a page is re-encrypted, its IV is reset so that the content of the whole encrypted page changes. For an attacker without the encryption key, it would be impractical to identify the location changes by comparing the binary content of the blocks.

Assume that the size of the hot zone is H . The size of the cold zone is then $D-H$. The length of the current page sequence is *L*. The expected size of access set in the hot zone is $\mathrm{E_H} = \mathrm{E}\left(H, cL\right)$, and the expect size of access set in the cold zone is then $\mathrm{E_C} = \mathrm{E}\left(D\text{-}H, L\right)$. The probability function of the size of IAS is the sum of two intersections:

$$P(I=i) = \sum_{h=0}^{i} \left(\frac{\binom{E_H}{h} \cdot \binom{H-E_H}{E_H-h}}{\binom{H}{E_H}} \cdot \frac{\binom{E_C}{i-h} \cdot \binom{D-H-E_C}{E_C-i+h}}{\binom{D-H}{E_C}} \right) \tag{3}$$

As $H \ll D$, so the above formula could write as

$$P\left(I=i\right) \approx \frac{\binom{E_H}{i} \cdot \binom{H-E_H}{E_H-i}}{\binom{H}{E_H}} \tag{4}$$

If $P(I > L) > \alpha$, where α the confidential level of an adversary, the adversary is is not able to convince that the current IAS is caused by repeated access on hidden partition.

From the other side, we need to satisfy that the intersection size in cold zone is not suspicious. As the code zone might be very large, the intersection size should be as small as possible, which give the rise of the inequality $\left(1-\left(\frac{L}{H}\right)^{C}\right)^{L} < 1/2$. The solution of this scheme then changes to find the pair of (H,c) to satisfy the two inequality simultaneously and at the same time with the minimum c.

We now show how this scheme improves the IAS problem through a concrete example. Suppose the total size of private storage D=2000, and the noise blocks for each access set is 10. For a page sequence with length 10, and the size of its IAS is 10. With H=22, and c=3, under the two-zone hypothesis, the chance that, in this case, the IAS is caused by entirely random access is 0.1, which is big enough so that an adversary could not deduce any information form current access sequence. According to the definition, those parameters are plausible for all access sequence which length equal to 10.

Discussion. As describe in the scheme, we make the large part of repeated access on data blocks occurred within hot zone. By using the above mentioned probability model, the IAS caused by data sequence is comparable to the one caused by random dummy sequence, which is hard for an adversary to induce which one is caused by data sequence. At the same time, IAS in the cold zone is kept as small as possible, which still give no clue of data sequence. The scheme is also has better scalability compared with m-block scheme because that the parameter c is not proportional to the total number of blocks.

5.8 Summary

In system model 2, the database engine and page storage are both migrated to untrusted environment, which poses severe security challenges of protecting and hiding the potential sensitive data, as the adversary could break-in and compromise the system from several places. Through the use of secure coprocessors, we proposed an architecture built on this secure hardware. Under this architecture we examine the security threats and give out solutions. In concrete terms, we proposed a query decomposition algorithm to separate client users so that only the parts of queries on shared data are sent to opened processor, which the attack might see. Then, we study the interplay between operations in open processor and access act ivies on private storage. To cover the intersections of repeated access on the same hidden partition, we provide a two-zone solution which is based on probability model and guarantee that the size of intersections caused by actual data access sequences is comparable to the one caused by dummy sequences. It thus raised the difficulty for an adversary to infer which access sequences are currently accessing hidden partitions.

Chapter 6

Conclusion

The database community is witnessing the emergence of two recent trends set on a collision course. On one hand, outsourcing of data management has become increasingly attractive for many organizations with the advances in the network technologies. On the other hand, escalating concerns about data privacy, recent governmental legislation, as well as high-profile instances of database theft, have sparked keen interest in enabling secure data storage and access. The two trends are conflict with each other. A client using a database service is required to trust the service provider with potentially sensitive data, leaving the door open for damaging leaks of private information. Outsourcing database server put the contained sensitive data at a high risk.

A steganographic database management system (StegDB) could resolve the conflict by providing steganography in data storage and query. In particular, it grants access to a protected database partition only if the correct access key is supplied; without it, an adversary could get no information about whether the protected partition ever exists. The protected partitions are hidden not only logically but physically, which ensures that a system intruder cannot detect the existence of those sensitive data even if he understands the hardware and software completely.

In this thesis, we presented two different StegDB design, i.e. model 1 and 2. In model 1, only the data storage is outsourced; in model 2, the database server entirely expose to untrusted environment. For model 1, we propose two data mining tools to mine informational patterns from access sequence on data storage which might be intercepted by an adversary. To spot quickly specious patterns which might hint the existence of hidden partitions, the mining tools can support various constraints. To counter the security threat exposed by the mining tools, we then provided two sequence transforming schemes which are aimed at diminishing access patterns in sequences. The experiment showed that at a marginal cost, the patterns in access sequence could be masked to the level that most patterns never repeat twice. For model 2, we proposed a framework based on secure coprocessor. In accordance with the security requirement of a StegDB, we gave a query decomposition algorithm and hide the interplay between operations in open processors and access activities on private storage. The work under model 2 guarantees that a StegDB server which is outsourced in a hostile environment still provides stenography feature without sacrificing its security standard.

This work is pioneering in the sense that it fills the gap between the high level security requirement and ubiquitous computing environment. It's the first step towards outsourcing multi-level secure database. The work can also find application in safeguarding user privacy in conventional DBMSs, such as masking access patterns in an encrypted database. For future work, we intend to investigate the performance of model 2 in real-life application. In addition, processing queries within the limited resources of a secure coprocessor might present new challenge. We will also extend the proposed model to incorporate existing protection schemes for outsourced encrypted database.

Appendix A

Query Decomposition Algorithm

Input: Global Query Q.
Output: A groups of queries $\{Q_{s1}, Q_{s2}, ...Q_{sn}\}$ executed on GDE, and an query plan executed in HDE.

Step 1. Identify hidden partitions.

```
For each different table $G$ in the from clause
    If a shared table $S$ with the same name of $G$ is found
        Create a new subquery $Q_{si}$.
        Add related projection items of Q to the projection list of $Q_{si}$.
        If a online hidden partition $H$ associated with $S$ is found
            If $H$ is a horizontal partition
                Create a new subquery $Q_{hi}$ for this partition.
                Find projection items $L_{select}$ of $Q_{hi}$ (projection item of $Q$, polyinstantiation).
                Create a consolidation operation $C_i$ to unite the intermediate result from $Q_{hi}$ and $Q_{si}$
            Else if $H$ is a vertical partition && the "where" clause or projection list includes attributes only in $H$
                Create a new subquery $Q_{hi}$ for this partition.
                Find projection items $L_{select}$ of $Q_{hi}$ (projection item of $Q$, search conditions, polyinstantiation) to the projection list of $Q_{hi}$).
                Create a consolidation operation $C_i$ to join the intermediate result from $Q_{hi}$ and $Q_{si}$.
    Else if G is a standalone hidden partition
        Create a new subquery $Q_{hi}$ for this partition.
        Add related projection items of $Q$ to the projection list of $Q_{hi}$.
```

Step 2. Process projection and selection list.

```
For each predicate P in the where clause
    If P involves attributes only in one global table $G$
        Push the predicates into related subqueries.
        For those cannot push down to the subqueries. Add the predicates to the Consolidation Operation above them.
```

```
Else If the $P$ involves several attributes not belong to the same table.
    Organize the predicate Ps into a lettuce, making each node includeing predicates that could apply the same group of subqueries.
    From the bottom of lettuce, for each node and level by level do the following
    Find the involved subquery or consolidation operation list and move queries in GDE to the head of list.
   Use a join-order selection algorithm to find the join order list $\{Q_1, Q_2, ...Q_n\}$.
        Create a consolidation operation $C_i$ to join two related table, then add applicable predicates to $C_i$.
        For remain $Q_j$ (or related $C_j$ above the subqueries) in the order list
            Create a consolidation operation $C_k$ to join $C_i$ with $Q_j$ (or related $C_j$), then add applicable predicates to $C_k$.
```

Bibliography

[1] K. John, B. David, C. Yan, E. Patrick, G. Dennis, G. Ramakrishna, R. Sean, W. Hakim, W. Westly, W. Christopher, and L. O. Ben Zhao, "OceanStore: An Architecture for Global-scale Persistent Storage," in *Proceedings of ACM ASPLOS*, 2000.

[2] H. Hacigumus, B. Iyer, and S. Mehrotra, "Providing Database as a Service," presented at International Conference on Data Engineering (ICDE), 2002.

[3] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra, "Executing SQL over encrypted data in the database-service-provider model," presented at Proceedings of the 2002 ACM SIGMOD international conference on Management of data, Madison, Wisconsin, 2002.

[4] H. Mannila, H. Toivonen, and A. I. Verkamo, "Discovery of frequent episodes in event sequences," *Data Mining and Knowledge Discovery*, vol. 1, pp. 259--289, 1997.

[5] H. Pang, K.-L. Tan, and X. Zhou, "StegFS: A Steganographic File System," presented at International Conference on Data Engineering (ICDE), Bangalore, India, 2003.

[6] A. Freier, P. Karlton, and P. Kocher, "The {SSL} Protocol Version 3.0, Internet-Draft," 1996.

[7] T. Dierks and C. Allen, "The {TLS} Protocol - Version 1.0," 1999.

[8] X. Ma, H. Pang, and K.-L. Tan, "Masking Page Reference Patterns in Encryption Database on Untrusted Storage," *Data & Knowledge Engineering*, 2005.

[9] T. F. Lunt, D. E. Denning, R. R. Schell, M. Heckman, and W. R. Shockley, "The SeaView Security Model," *IEEE Trans. Softw. Eng.*, vol. 16, pp. 593-607, 1990.

[10] P. D. Stachour and Thuraisingham, "Design of LDV: A Multilevel Secure Relational Database Management," *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, pp. 190-209, 1990.

[11] D. E. Denning, "The Sea View Security Model.," presented at IEEE Symposium on Security and Privacy, 1988.

[12] Oracle Corporation, "Database Encryption in Oracle 9i," 2001.

[13] "IBM Data Encryption for IMS and DB2 Database, Version 1.1.," 2003.

[14] G. I. Davida, D. L. Wells, and J. B. Kam, "A database encryption system with subkeys," in *ACM Trans. Database Syst.*, vol. 6, 1981, pp. 312-328.

[15] E. Damiani, S. D. C. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati, "Balancing confidentiality and efficiency in untrusted relational DBMSs," presented at ACM conference on Computer and communications security, 2003.

[16] H. Pang, R. H. Deng, and F. Bao, "Tamper Detection and Erasure Recovery in Resilient DBMS," in *Submitted for publication*, 2003.

[17] J. F. Raymond, "Traffic Analysis: Protocols, Attacks, Design Issues and Open Problems," presented at Proceedings of Workshop on Design Issues in Anonymity and Unobservability, 2001.

[18] M. Abe, "Mix-network on permutation networks," presented at In Advances in cryptology - ASIACRYPT'99, 1999.

[19] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM*, vol. 43, pp. 431-473, 1996.

[20] B. Chor, O. Goldreich, E. Jushilevitz, and M. Sudan, "Private Information Retrieval," in *Journal of the ACM*, vol. 45, 1998, pp. 965-982.

[21] D. Asonov and J.-C. Freytag, "Almost optimal private information retrieval," presented at Workshop on Privacy Enhancing Technologies (PET), 2002.

[22] B. Chor and N. Gilboa, "Computationally private information retrieval," presented at Annual ACM Symposium on Theory of Computing, 1997.

[23] Xi Ma, H. Pang, and K.-L. Tan, "Finding Constrained Frequent Episodes Using Minimal Occurrences," presented at IEEE International Conference on Data Mining, 2004.

[24] G. Simmons, "The Prisoners' Problem and the Subliminal Channel," presented at Proceedings of the CRYPTO '83, 1984.

[25] N. F. Johnson and S. Jajodia, "Exploring Steganography: Seeing the Unseen," *Computer*, vol. 31, pp. 26-34, 1998.

[26] M. Chapman and G. Davida, "Information and Communications Security -- First International Conference," 1997.

[27] M. D. Swanson, B. Zhu, and A. H. Tewfik, "Audio Watermarking and Data Embedding -- Current State of the Art, Challenges and Future Directions," presented at Multimedia and Security -- Workshop at ACM Multimedia '98, Bristol, United Kingdom, 1998.

[28] R. Anderson, R. Needham, and A. Shamir, "The Steganographic File System," presented at International Workshop on Information Hiding, D. Aucsmith, Ed., Portland, Oregon, USA, 1998.

[29] J. S. Heidemann and G. J. Popek, "File-System Development with Stackable Layers," *ACM Transactions on Computer Systems*, vol. 12, pp. 58-89, 1994.

[30] H. Chou and D. J. DeWitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems," presented at International Conference on Very Large Databases (VLDB), Stockholm, 1985.

[31] A. J. Smith, "Sequentiality and Prefetching in Database Systems," *ACM Trans. Database Syst.*, vol. 3, pp. 223-247, 1978.

[32] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *Proceedings of the fifteenth ACM symposium on Operating systems principles*, 1995, pp. 79-95.

[33] R. Agrawal and R. Srikant, "Mining Sequential Patterns: Generalizations and Performance Improvements," presented at International Conference on Extending Database Technology (EDBT), 1996.

[34] W. Li, "Mutual Information Functions versus Correlation Functions," *Journal of Statistical Physics*, vol. 60, pp. 828-837, 1990.

[35] D. P. F. L. Feldman, "A brief introduction to: Information Theory, Excess Entropy and Computational Mechanics," 2002, pp. 13-15.

[36] "{TPC} Benchmark C Standard Specification, Revision 5.1," 2002.

[37] S. W. Smith and S. Weingart, "Building a high-performance, programmable secure coprocessor," *Comput. Networks*, vol. 31, pp. 831-860, 1999.

[38] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. v. Doorn, S. W. Smith, and S. Weingart, "Building the IBM 4758 Secure Coprocessor," *IEEE Computer*, vol. 34, pp. 57-66, 2001.

[39] W. Meng and C. Yu, "Query processing in multidatabase systems," in *Modern database systems: the object model, interoperability, and beyond*: ACM Press/Addison-Wesley Publishing Co, 1995, pp. 551-572.

[40] P. Lin and K. S. Candan, "Hiding Tree Structured Data and Queries from Untrusted Data Stores," *Information Systems Security Journal*, 2004.

[41] X. Zhou, H. Pang, and K.-L. Tan, "Hiding Data Accesses in Steganographic File System," presented at International Conference on Data Engineering (ICDE), Boston, USA, 2004.