Building Disclosure Risk Aware Query Optimizers for Relational Databases

Mustafa Canim, Murat Kantarcioglu[*]
Department of Computer Science
The University of Texas at Dallas
{canim, muratk}@utdallas.edu

Bijit Hore, Sharad Mehrotra
Department of Computer Science
University of California at Irvine
{bhore, sharad}@ics.uci.edu

ABSTRACT

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

1. INTRODUCTION

Most organizations today store increasing amounts of sensitive data in computer based systems. At the same time there are increasing concerns related to the security and privacy of the stored data due to many data theft incidents reported in the past. According to a recent report [15], approximately 285 Million records have been stolen from databases in 2008. It is believed that approximately 4.3 million of those records contain personally identifiable information [15]. According to another study, after a data theft incident, companies need to spend between \$90 and \$305 per lost record for various forensic, legal and IT costs [4].

To reduce the effect of various attacks and limit disclosure risks of sensitive data, organizations prefer to store the confidential data in encrypted format in databases. To support such demand, most of the commercial DBMS products in the market nowadays have built-in encryption support. One of these brands, Microsoft SQL Server 2008 [1], provides Transparent Data Encryption (TDE). TDE provides protection for the entire database at rest without affecting existing applications. In this security mechanism, a cryptographic key hierarchy is used to guarantee the secrecy of the keys. All data and index pages written to the storage device are encrypted with database master key (DMK). All DMKs under a certain service running in the system are encrypted with the corresponding Service Master Key (SMK). At the root of this key tree, service master keys are encrypted with the OS level root key. In this model, since the encryption and decryption operations are performed in memory, the cryptographic keys needed for such operations must be kept in memory as well. Unless a special hardware is used, all of these keys are kept in the main memory as long as they are in use. The implicit threat model assumed by these products is that the database server is trusted and only the disk is vulnerable to compromise. In the event that the physical storage devices are stolen this method prevents data theft effectively. On the other hand, this model is not strong enough to prevent data disclosure if the attacker has access to the main memory of the server itself. According to the Verizon's Data Breach Investigation Report [15], newer varieties of malware utilities bypass existing access controls and encryption, effectively creating vulnerable data stores that can later be retrieved from the victim environment. Examples of this include the usage of memory scrapers, sophisticated packet capture utilities, and malware that can identify and collect specific data sequences within memory, unallocated disk space and pagefile. According to this report, 85 % of the 285 million records breached in the year 2008 were harvested by custom-created malware. Clearly, a malware observing memory can easily capture the master keys used to encrypt/decrypt the sensitive information stored in the disks[2]. Once the keys are revealed, all sensitive

[*]This work was partially supported by Air Force Office of Scientific Research MURI Grant FA9550-08-1-0265, National Institutes of Health Grant 1R01LM009989, National Science Foundation Grants Career-0845803, and CNS-0964350.

Proceedings of the VLDB Endowment, Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... \$ 10.00.

[1]Similar functionality is provided by Oracle. Due to lack of space, we do not discuss it here.
[2]Memory scraping is listed among the top 15 data breach threats in [15].

data may be compromised irrespective of what encryption algorithm has been used.[3]

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

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

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

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

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

1.1 Threat Model and System Overview

Threat model: In this paper, we address a threat model where both the memory and hard disk are not trusted and only the client and HSM are trusted. Trusting a hardware component means that there is negligible probability that any malware or malicious agent will have access to (be able to tamper) the data residing on that component. Furthermore, we assume a passive adversary (e.g., malware) has access to the contents of the memory and the entire hard disk from time t_1 to time t_2. In this threat model, we try to limit what the passive adversary can learn by observing the contents of the memory by smartly designing the query optimizer.

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

In the proposed system, the decryption operation is performed in the server with the symmetric keys generated by the HSM. Based on the threat model described above, we define the disclosure risk as follows. Let S_{key} be the set of symmetric keys residing in the memory during the attack $[t_1, t_2]$. We assume that all sensitive attributes of the records (i.e., cells) encrypted with any key in S_{key} are compromised. Based on this assumption we propose disclosure models in Section 2 for different query evaluation techniques. In these models we use "**the number of cells**" as the disclosure metric[5]. To clarify the definition of disclosure cost, consider the following example. Let a table T have two sensitive and three non-sensitive attributes and say, each data page of T can accommodate 100 tuples of T. Suppose further that each data page of T is encrypted with a unique key. That is, whenever a data page of T is retrieved to the memory, the HSM generates a unique decryption key for this page and the sensitive records in this page are decrypted with this key in the main memory of the server. If a query plan p requires accessing 1000 data pages of T, then the disclosure risk of p is computed as: 1000 * 100 * 2 = 200,000. Using this metric, we compare the disclosure cost of different query plans.

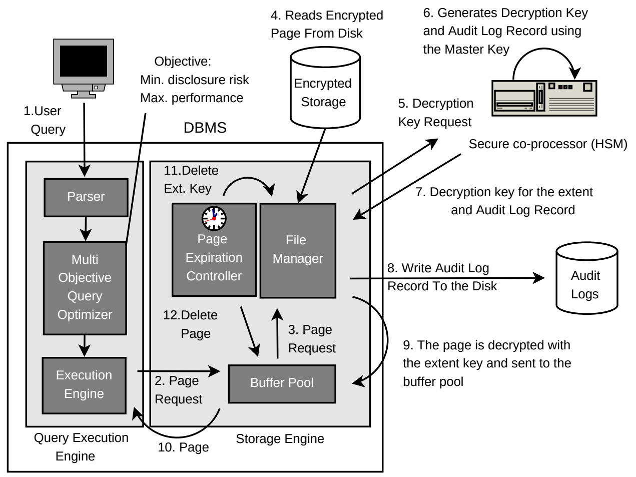
Proposed Architecture:

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

[3]The proposed techniques in this paper are also effective against the attack scenarios described in [6] where they describe how to capture cryptographic keys from the memories with cold boot attacks.

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

[5]*Cell* is described as the smallest indivisible values of the tuples.



4. Reads Encrypted Page From Disk
6. Generates Decryption Key and Audit Log Record using the Master Key
Objective: Min. disclosure risk Max. performance
Encrypted Storage
5. Decryption Key Request
1.User Query
DBMS
Secure co-processor (HSM)
11.Delete Ext. Key
Parser
7. Decryption key for the extent and Audit Log Record
Page Expiration Controller
File Manager
Multi Objective Query Optimizer
8. Write Audit Log Record To the Disk
Audit Logs
12.Delete Page
3. Page Request
9. The page is decrypted with the extent key and sent to the buffer pool
Execution Engine
2. Page Request
Buffer Pool
Query Execution Engine
10. Page
Storage Engine

Figure 1: System overview

rection for future work.

During the execution, the pages are requested from the storage engine (item 2). If the page does not exist in the buffer pool, the request is forwarded to the file manager for retrieval (item 3). After the encrypted page is retrieved from the disk (item 4), the file manager requests the decryption key from the HSM (item 5). This request includes the extent ID [6] of the page. In the proposed system, all of the pages within an extent are encrypted with the same key. One could prefer key management at the level of pages or records but there are negative performance implications of this decision. We discuss the details of why we prefer extent level key management in Appendix C.

Once the key request is received (item 6), HSM computes the hash of this ID and encrypts the output of this hash function with the master key stored within the HSM. Therefore a MAC based approach is used to generate the extent keys. Once the key is generated, it is returned to the file manager along with the audit log record (item 7). Next, the file manager writes the audit log record to the disk (item 8). The audit logs are used to keep track of extent accesses during the query execution. Since HSM needs to be used to access any encrypted data page, such logs will provide an accurate estimate of what could be leaked during an attack. Next, the file manager decrypts the page using the extent key and forwards the decrypted page to the buffer pool for processing (item 9, 10). Note that the decryption of the data page is performed in the system memory, not in the HSM. If the attacker monitors the content of the memory during these operations, all records within the extent would be compromised since the extent key resides in the memory during the attack.

To guarantee that the remnants of the sensitive information is removed from the memory, the proposed system has a mechanism called *Page Expiration Controller*. The objective of this component is to delete both the extent keys and the sensitive pages (item 11, 12) from the memory after a certain amount of time, t_{life}. In [3], Chow et al. present a technique for reducing the lifetime of sensitive data (i.e., encryption keys, sensitive records) in memory called secure deallocation so as to minimize the data disclosure from the memories. The basic idea is to "zero" the data either at deallocation or within a short, predictable period afterward in general system allocators. They show that this method substantially reduces the data lifetime with minimal implementation effort in the existing operating systems. We use a similar technique to guarantee that given an attack interval $[t_1, t_2]$ there are no remnants of the data pages or the keys retrieved to the memory before $t_1 - t_{life}$. This is essential for the accuracy of the auditing process. If a data page has to stay longer than t_{life} in the memory, an audit log record including this extension event is written to the *Audit Logs* disk.

[6]An extent is defined as a set of contiguous blocks allocated in a database tablespace.

Outline of the remaining paper: In Section 2 we discuss the relation of performance and disclosure risk in a RDBMS and derive concrete measures for the latter. In Section 3 we discuss how to jointly optimize for both performance and disclosure-risk and describe our implementation of the enhanced disclosure aware query optimizer. Subsequently, in Section 4 we describe our experimental setup and discuss the results on the TPC-H benchmark and conclude in Section 5. Due to lack of space, we summarize the related works in Section A in Appendix.

2. DISCLOSURE METRICS FOR MEMORY-BASED ATTACKS

Current database products are designed to minimize the response times of the queries issued by the clients. Estimating the query execution cost is one of the major steps in the query optimization process. Modern databases consider numerous factors while estimating the cost of alternative plans. Some of these factors are number of page I/Os and blocked I/Os, physical access speed of the storage devices, CPU costs etc. [10]. Using these parameters, the goal of the traditional optimizer is to generate a query plan that minimizes the response time of the issued queries. Typically, query optimization consists of two phases - a heuristics based query rewriting phase that generates a logical plan and a cost-based optimization phase that generates a physical plan starting from the output of the first phase. In the first phase, the optimizer employs standard heuristics to rewrite the parsed query tree where all selection and projection operators (predicates) are pushed down the appropriate branches of the tree as deep as possible. The second phase is a cost-based optimization step where predicate evaluation ordering and join ordering (for multi-relation queries) are selected. However, the sensitivity of contents of the tables are not taken into account while evaluating alternative plans. This, in turn, could lead to a dramatic change in the data exposure risk depending on the selected plan. Our goal in this study is to design an optimizer that not only maximizes the performance but also minimizes the disclosure risk. Below we first discuss this problem by analyzing a query evaluation scenario and then present the measures to estimate the disclosure risk of alternative query execution plans.

In our model since the pages are decrypted as soon as they are brought into the memory, the disclosure risk is only proportional to the number of different extent keys that are accessed. This, in turn is assumed to be proportional to the number of pages brought into the memory under the *random spread* assumption (i.e., the fact that each matching tuple can reside anywhere on the disk independent of other tuples in the set). Therefore, the disclosure risk is only dependent upon the selectivity of the least selective predicate in the query. In an alternate model where one has a choice to postpone the decryption of a page until processing the sensitive records, the optimal predicate ordering might differ significantly. A detailed discussion on this issue is provided in Appendix D.

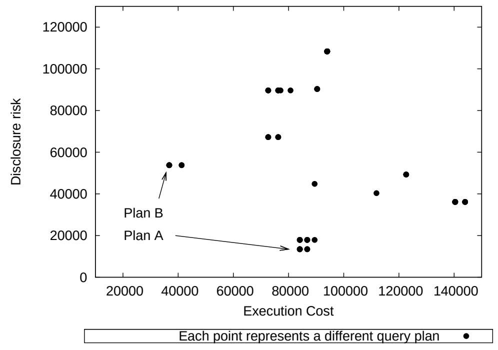


Figure 2: Possible execution plans for a given query

Disclosure risk vs. performance cost: Before deriving the disclosure metrics to be used by an optimizer, here we illustrate via an example how the disclosure cost and performance vary across different query plans for a sample query. Using the metrics to be described soon (in Section 2.1 and Section 2.2) we implemented a disclosure estimator module in MySQL query optimizer. The objective of this module is to estimate the disclosure risk of alternative query execution plans. Using the TPC-H schema [16], we created a database instance and designated four attributes of *Customer* table and eight attributes of *Lineitem* table as sensitive. Next, a query joining six tables is issued to the database (See Table 2 in the Appendix for the query). Consequently, 720 (6!) different enumeration plans are generated. Each point in Figure 2 represents an alternative execution plan (Only the top 20 query plans are plotted, the full plot is shown in Figure 8 in the Appendix). The x axis represents the query execution cost and the y axis represents the disclosure risk for the execution plans. In this figure, plan B minimizes the execution cost whereas plan A minimizes the disclosure risk. Hence, there is no single query execution plan for which both the execution time and the disclosure risk are minimized. This is a common problem in multi-objective optimization which involves the application of special methods in order to obtain a viable solution.

In Section 3, we discuss alternative approaches to solve this multi-objective optimization problem. We now present metrics to estimate the disclosure risk given different input parameters and data access paths for a query. We also refer to the disclosure risk metrics as the *cost functions*

2.1 Disclosure Risk for Single-relation Queries

Single-relation queries include only one relation in the *FROM* clause. During the evaluation of these type of queries the disclosure cost changes dramatically depending on the existence of indexes. Therefore we will analyse the disclosure issue in single relations under two categories: "Plans without indexes" and "Plans utilizing at least one index" similar to the chronology of discussions in [10]. We use the notations given in Table 1 while describing the models.

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

Table 1: Notations used for disclosure model

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

Granularity of encryption: We provide the disclosure cost models for only column level encryption. We assume that a column encryption scheme similar to one proposed in [7] can be implemented wherein each page that is brought into the memory is decrypted separately. [7] For the databases where the table level encryption is used, the cost estimations can be carried out by assuming that all columns contain sensitive information. (See [2] for a detailed discussion on table level encryption and column level encryption).

2.1.1 *Plans without indexes*

The simplest way of accessing the records of a relation is to scan the data pages in a linear order if there is no index created on it. During a table scan all data pages are retrieved to the memory. Consequently, all sensitive information stored in relation R will be disclosed. The disclosure cost in this case would be:

$$D(R) = N_{ext}(R) * E_{rec}(R) * C_{sens}(R). \tag{1}$$

2.1.2 *Plans utilizing an index*

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

Single-Index Access Path: If there is an index on a relation R which matches a range condition in the *WHERE* clause of the query, first, the initial leaf page of the index tree including the data entries (or records if it is a primary index) should be identified. During this operation some non-leaf index pages should be retrieved to the memory. Typically 2-4 I/O's are performed to find this page [10]. These non-leaf index pages may include sensitive values. The disclosure cost of these top down tree traversal is $E_{iPage}(R) * {I^i}_{nleaf}(R)$ per I/O operation.

The disclosure cost of traversing the leaf pages depends on whether the index is clustered. If it is clustered, then the records satisfying a range condition could be found in the consecutive data pages[8]. Since the consecutive pages are

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

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

stored in contiguous extents, we can compute the disclosure risk by estimating the number of extent keys that need to be retrieved from the HSM to the memory. In the worst case, $\left(\left\lceil \frac{S_{range}(R)}{E_{rec}(R)} \right\rceil + 1\right)$ keys will be requested from the HSM, including the first and last extents which includes some extra records out of the range condition. Then, the worst case total disclosure cost of a clustered index access will be:

$$\left(\left\lceil \frac{S_{range}(R)}{E_{rec}(R)} \right\rceil + 1\right) * E_{rec}(R) * C_{sens}(R) + 4 * E_{iPage}(R) * {I^i}_{nleaf}(R) \tag{2}$$

If the index is not clustered (i.e., a secondary index), the traversal of data entries will reveal some information if the key values in the data entries are sensitive. Then, the disclosure cost of the traversal will be ${I^i}_{leaf}(R)$ for each consecutive accessed leaf page. In the worst case the I/O cost would be one I/O per matching key value under the random spread assumption. Therefore, the worst case disclosure cost is estimated as $E_{rec}(R) * C_{sens}(R)$, that is, one extent per matching tuple. For average cost estimation we use a model similar to the one in [11]. Suppose that $S_{range}(R)$ records satisfy a given range condition. Then, the disclosure cost will be $E_{exp}(S_{range}(R)) * E_{rec}(R) * C_{sens}(R)$ where $E_{exp}(S_{range}(R))$ can be estimated by using the Cardenas formula [17]. If there are $|R|$ records in $N_{ext}(R)$ extents in relation R, then the number of extents touched while accessing $S_{range}(R)$ records through an unclustered index will be:

$$E_{exp}(S_{range}(R)) = N_{ext}(R) * \left(1 - \left(1 - \frac{1}{N_{ext}(R)}\right)^{S_{range}(R)}\right) \tag{3}$$

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

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

Note that, expression 3 is independent of $|R|$ (the total number of records in R). Yao mentions in [17] that the error involved in using this approximation is negligible for cases in which the number of tuples per page is not a small number (say < 10). This assumption is typically true for page sizes used in today's database systems (4KB-32KB) [11].

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

Index-Only Access Path: If all of the attributes mentioned in the query (in the *SELECT*, *WHERE*, *GROUP BY*, or *HAVING* clauses) are included in the search key for some *dense* index on the relation in the *FROM* clause, an index-only scan can be used to compute answers [10]. The disclosure cost of using an index-only plan consists of the disclosure due to the traversal of non-leaf and leaf nodes of the index tree. Similar to the single index case, the disclosure cost of top down tree traversal cost will be $E_{iPage}(R) * {I^i}_{nleaf}(R)$ for each I/O operation. As for the leaf pages, if the index is a primary index then the disclosure cost will be $\left(\left\lceil \frac{S_{range}(R)}{E_{rec}(R)} \right\rceil + 1\right) * E_{rec}(R) * C_{sens}(R)$. If it is a secondary index, then the disclosure cost will be ${I^i}_{leaf}(R)$ for each consecutive accessed leaf page.

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

2.2 Disclosure Risk for Multi-relation Queries

Join Algorithms: The most commonly used join algorithms in the conventional database systems are *nested loop join*, *block nested loop join*, *index nested loop join*, *sort merge join*, and *hash join*. Except *index nested loop join*, all of these algorithms require at least one scan over the joined relations[9]. Therefore, the disclosure cost of these algorithms except *index nested loop join* is computed similar to the single relation plans without indexes. That is, $D(R) = N_{ext}(R) * E_{rec}(R) * C_{sens}(R)$ for each joined relation. This can be effectively written as:

$$|R| * C_{sens}(R) \tag{5}$$

As for the *index nested loop join* algorithm, the disclosure cost of accessing the outer relation, and the inner relation, is estimated separately. The summation of these costs will yield the overall cost of the join operation. Let $D_o(R)$ be the disclosure cost of accessing the outer relation R and $D_i(S)$ be the cost of accessing the inner relation S. If the relations R and S are joined with index nested loop join algorithm, then the records of R is scanned at least once. Consequently, all sensitive information in R will be revealed. Then, $D_o(R)$ will be $|R| * C_{sens}(R)$. On the other hand, the disclosure cost of accessing the inner relation depends on the index type on it (i.e., primary or secondary index). For each tuple from R, the index on S will be accessed to find matching tuples. Let $D_{index}(S)$ denote the disclosure cost of accessing S via index. In Section 2.1.2 we explained how to estimate $D_{index}(S)$ for a particular equality or range condition. Using this model the disclosure cost of accessing S is the following:

$$D_i(S) = Min\left(|R| * D_{index}(S), |S| * C_{sens}(S)\right) \tag{6}$$

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

3. MULTI-OBJECTIVE QUERY OPTIMIZATION

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

3.1 Combining two cost measures

Multi-objective (MO) problems are traditionally solved by converting all objectives into a single objective (SO) function. The ultimate goal in this process is to find the solution that minimizes or maximizes this single objective while satisfying the given constraints [8]. Below, we discuss some of the classical methods and how to apply them to our problem.

Weighted Aggregation Approach: Conversion of the MO function into an SO function is usually carried out by aggregating all objectives in a weighted function. The major

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

issue in this approach is that it requires an a priori knowledge of the relative importance of the objectives [8]. In our case, aggregation can be employed by expressing the disclosure risk and execution cost in monetary terms. Next, this aggregate monetary cost can be minimized subject to the constraints. According to a new study by Forrester Research, the average security breach can cost a company between \$90 and \$305 per lost record [4]. Considering the criticality of the system, the cost of the execution time for the issued queries can also be estimated. Then, these unit weights can be multiplied by the estimates for each query plan. The objective of the query optimizer is minimizing the sum of these costs.

Constraint Approach: An MO problem with n objectives can also be solved by transforming $n-1$ objectives into constraints and minimizing or maximizing only one objective subject to the constraints. In this approach a security administrator sets a *tolerance ratio* $\alpha > 0$ on either the performance loss or the disclosure risk. Suppose κ denotes the minimum disclosure risk that can be achieved among all possible execution plans. Suppose further that the disclosure risk cannot exceed $\kappa+\kappa*\alpha\%$ is the security constraint. Then, the objective of the optimizer is to minimize the execution cost while ensuring that the disclosure risk is not higher than $\kappa+\kappa*\alpha\%$. Consider the following example. For a given query q, suppose that the disclosure risk of query plan p_i is 1000 records and this plan minimizes the disclosure risk among all possible execution plans. As for the *tolerance ratio*, the security administrator sets the value of α at 20. Given α, the optimizer is now free to select any query plan p_j as long as its execution cost is less than the execution cost of p_i and the disclosure risk is less than 1200 records. Hence the optimizer may select a query plan p* which has a disclosure risk between 1000-1200 records and an execution time less than that of p_i.

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

3.2 Modifications in MySQL Query Optimizer and InnoDB Storage Engine

Query Engine: The suggested algorithms has been implemented by modifying the source code of MySQL 5.1.38 query optimizer. The implementation is about two-three man months, but for more sophisticated optimizers this cost could be slightly higher. Given a set of query tables, *best_extension_by_limited_search* procedure in *sql_select.cc* file searches for the optimal ordering of these tables and the corresponding optimal access paths to each table. The pseudocode of this procedure is given in Algorithm 1 in Appendix F. The name of the recursive procedure in the pseudocode is denoted as *findBestPlan*. This is essentially a dynamically pruning, exhaustive search algorithm. Therefore, the complexity of the algorithm is O(N!) in the worst case where N is the number of base tables. The procedure constructs all alternative left deep trees by iterating on the number of relations joined so far, while pruning suboptimal trees. We changed three parts of this procedure. These are: *cost estimator*, *pruning conditions* and *best plan selection conditions.* The given pseudocode includes the implementation details of the *Weighted Aggregation* approach, *Constraint on Performance* and *Constraint on Disclosure*, *Max Performance*, and *Min Disclosure* approaches. *Max Performance* is the default implementation of the optimizer which aims to maximize the performance by reducing the overall execution cost. We implemented a second approach called *Min Disclosure* and the goal of this is to reduce the disclosure without considering the overall execution cost.

For each table T in a set of base tables S_{tables}, the execution cost and the disclosure cost of joining this table as the inner table with the tables in the current partial plan is estimated and added to the costs accumulated so far (line 2, 3 in Algorithm 1). Then, the pruning conditions are checked to back track the traversal of the tree when a better plan is not available (line 4-28). When *Weighted Aggregation* approach is used, a weighted cost of disclosure cost and execution cost is computed. Using this aggregate cost a decision of pruning the rest of the path is made (line 17, 18).

For *Constraint on Performance* approach, a maximum acceptable limit on the performance cost is computed using κ and α. (κ is the minimum execution cost that can be achieved among all possible execution plans and α is the tolerance ratio). If the traversal of the nodes in the path does not provide a better performance than the best plan selected so far or a better disclosure cost is not available, the path is pruned (line 22-23). Without a full traversal of the tree, the value of κ can not be estimated. To tackle this problem two methods can be applied. The first method involves traversing the tree twice. In the first traversal, the value of κ is computed and then used in the second run to find a plan which minimizes the disclosure cost while ensuring that the execution cost does not exceed $\kappa+\kappa*\alpha\%$. The second method has the advantage of traversing the tree once but this solution requires exponentially large memory space in the worst case. The details of this alternate algorithm is discussed in Appendix E.

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

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

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

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

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

4. EXPERIMENTS

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

4.1 Experiments with TPC-H workload

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

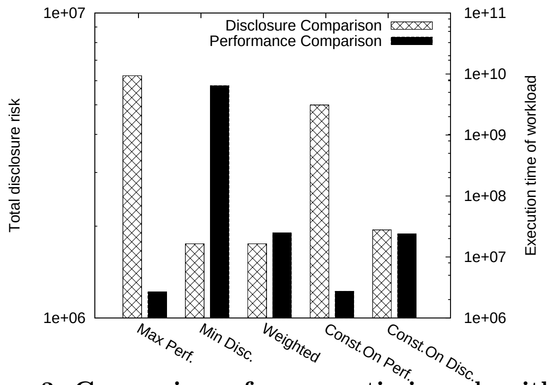


Figure 3: Comparison of query optimizer algorithms

Experiment with constant weights and constraints: In this experiment, our goal is to compare the performance and disclosure risk of running a workload while choosing alternative multi-objective optimization algorithms. We run the workload repeatedly (five times) on the modified MySQL database server and monitored the cumulative disclosure risk and the execution time estimated by the query optimizer for different algorithms. In the configuration file we designated *Customer* and *Supplier* tables as the tables including sensitive information. The results are given in Figure 3. The dashed bars in the figure represents the total disclosure risk (see left y axis) for a particular algorithm. The black bars represent the cumulative execution time of the workload (see right y axis). Each pair of bars correspond to a single run of the workload. In the first run *Max Perf.* algorithm is chosen. In this algorithm, the performance is maximized while disregarding the disclosure risk. As seen in the figure, the total execution time of the workload is minimized when this algorithm is chosen. However, this option also maximizes the disclosure risk by retrieving maximum number of sensitive information to the memory. This option represents the default choice of conventional database optimizers. The second algorithm aims to minimize the disclosure risk without considering the performance penalty. Note that the execution time of the workload in this case is about three orders of magnitude greater than the one in *Max Perf.* algorithm while reducing the disclosure risk by a factor of three. The third pair in the figure corresponds to the *weighted aggregation* approach. As we discussed earlier the average security breach can cost a company about $200 per lost record. So we set the weight of the disclosure as 200. As for the execution cost, we assumed that each delayed second of the queries costs a dollar for the company. So we set the disclosure cost weight as 1, in this particular scenario. Given these parameters, we observed that the cumulative disclosure risk of the workload in this case is similar to the one in *Min Disc.* approach whereas the execution time is reduced by about two orders of magnitude. Compared to the *Max Perf.* approach, the execution cost is increased. If one considers both the disclosure risk and execution cost of the workload this approach would yield more preferable results compared to both *Max Perf.* and *Min Disc.* approaches. As for the fourth experiment, we tuned the database with the *constraint* approach parameters. In this run, we assumed that there is 30 % limit on the performance loss. That is, the optimizer aims to minimize the disclosure risk while guaranteeing that there is no performance loss more than 30 % compared to the best running time. As seen in this figure, there is a slight increase in the overall workload execution performance (less than 3 %) whereas the disclosure risk is better than the one in *Max Perf.* approach (about 20 %). The last experiment is similar to the fourth one. In this run, we assume that there is 30 % limit on the disclosure risk. The results are similar to what we observe in *weighted aggregation* approach. Compared to the other three approaches this one provides more balanced results in terms of both performance and disclosure.

We implemented a program to analyse the audit logs generated by the DBMS. Using this program, we counted the number of SCP accesses during the execution of the workload (see Figure 4-a). We observed that the number of SCP accesses in *Max Perf.* is reduced by a factor of three in *Min Disc.*, *Weighted Aggregation* and *Constraint on Disclosure* approaches which is parallel with the disclosure estimations of the query optimizer.

Experiment with changing constraints: In this experiment, our goal is to observe the impact of constraint parameter on the disclosure and performance of a given workload. Using the database instance described above we run the same workload on MySQL while changing the constraint parameter. Each point in Figure 4-b represents the disclosure and execution time of a single run of the workload. As discussed in Section 2, a constraint could be defined either on disclosure or performance. Therefore, for various constraint ratios (see the x axis), we run the workloads for both "Constraint on Performance (COP)" and "Constraint on Disclosure (COD)". In the COP approach, as the constraint ratio on performance is relaxed, the execution time increases while the disclosure risk reduces. In the

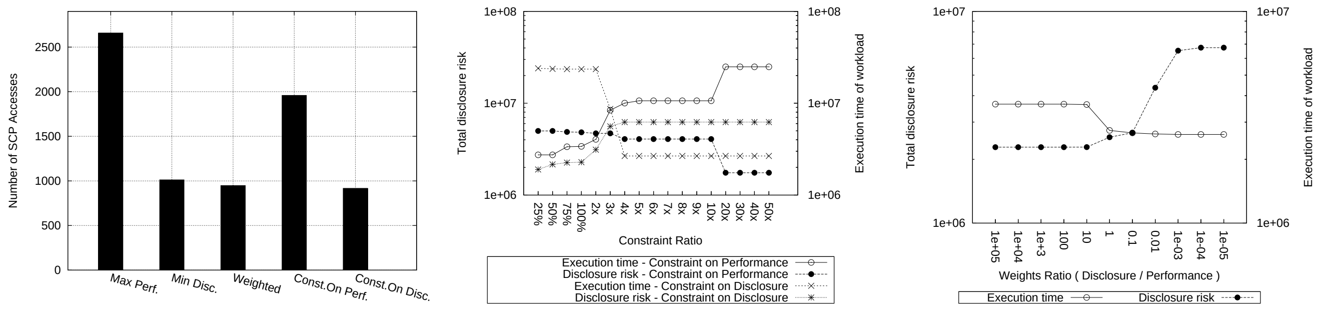


Figure 4: a) Number of SCP accesses during the execution of the workload b) Constraint on Disclosure and Constraint on Performance c) Weighted Aggregation Approach

COD approach, the opposite of this behavior is observed as expected.

Experiment with changing weights:

In this experiment, our goal is to observe the impact of different weights on the disclosure and performance of a given workload. Similar to the constraint approach experiments, each point in Figure 4-c represents the disclosure and execution time of a single run of the workload. For various weight ratios (ratio of disclosure cost to the execution cost), we run the workload. As the weight of the performance cost gets higher, the execution time starts decreasing while the disclosure risk starts increasing.

4.2 Performance Overhead of Key Generation and Audit Log Generation Mechanism

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

5. CONCLUSION

In this work, we consider the problem of estimating disclosure cost incurred in query processing over data with sensitive information. Specifically, we study the risk of sensitive data exposure when the adversary is able to access the contents in the main memory of the server during query execution. We enhanced the query optimization techniques to consider both the sensitive data disclosure risk and the execution costs and provide knobs to the administrator to select the desired tradeoff between the two. In addition, we developed methods to securely generate audit logs which is critical for forensic analysis after an attack occurs. Our results indicate that careful consideration of disclosure risk and query execution cost is indeed required to balance security and efficiency of query processing. Our work in this paper just scratches the surface of this important new direction of potential research. As a future work, we will analyze the impact of various query rewriting strategies and heuristics (e.g., pushing selections) on disclosure risk. Also, we will explore how to efficiently store and analyze the audit log for effective forensic analysis. Furthermore, we plan to extend our framework to consider active memory attacks where the attacker can modify the contents of the memory as well.

6. REFERENCES

[1] N. Anciaux, M. Benzine, L. Bouganim, P. Pucheral, and D. Shasha. Ghostdb: querying visible and hidden data without leaks. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 677–688, New York, NY, USA, 2007. ACM.

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

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

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

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

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

[7] B. Iyer, S. Mehrotra, E. Mykletun, G. Tsudik, and Y. Wu. A framework for efficient storage security in rdbms. In *International Conference on Extending Database Technology (EDBT 2004)*, 2004.

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

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

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

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

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

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

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

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

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

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

APPENDIX

A. RELATED WORK

There are numerous studies that address several problems regarding database auditing and forensic analysis and provide alternative solutions to deal with them. Most of these studies focus on detection and protection of database tampering. In [12] Schneier et al. present a general scheme that allows keeping an audit log on an insecure machine, so that log entries are protected even if the machine is compromised. In [5] a special mechanism is proposed to certify the creation and modification of digital documents such as text, video and audio files. In [14] Stahlberg et al. investigate the problem of unintended retention of data in database systems, and build a database system that is resistant to unwanted forensic analysis. They show how data remnants in databases pose a threat to privacy. In [9] Pavlou et al. focus on the problem of determining when the tampering occurred, what data was tampered with, and who did the tampering, via forensic analysis. In [13] a security mechanism within a DBMS is proposed that prevents an intruder within the DBMS itself from silently corrupting the audit logs. The solution they propose involves use of cryptographically strong one-way hash functions. Unlike ours, none of these studies addresses the problem of detecting and minimizing what is revealed in databases during a memory attack. In [1], a secure database architecture is implemented within a tamper resistent USB key. Unlike this study, we outsource the computations to the server (instead of performing the cryptographic operations within secure hardware due to the performance limitations) while trying to minimize sensitive data disclosure risks under various attacks.

Table 2: Sample query

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

B. MOTIVATING EXAMPLE FOR QUERY OPTIMIZATION

Suppose that the following query is issued to the database: *Select* $*$ *From* A, B, C *Where* $A.pk = B.pk$ *and* $B.pk = C.pk$. Table A includes sensitive information and all of the data pages of this table are stored in an encrypted format in the disk. Tables B and C include non-sensitive information and the records of these tables are not encrypted. Suppose further that two alternative join plans given in Figure 5 are considered. Without the knowledge of the sensitivity of the tables, the query optimizer selects the plan on the left assuming that this is the least costly plan. However, this plan requires accessing all of the rows of table A and therefore maximizes the risk of disclosure. To be able to access all of the rows, all of the data pages of A should be decrypted. Suppose that table A on the second plan is accessed via index only plan and retrieves less than 2 % of the data pages of A to the memory. Compared to the former plan this could be a slower one but may reduce the disclosure risk effectively. If an attacker monitors the content of the memory during the execution of this query, the amount of information leakage in the second plan would be much less than the one in the first plan. Being aware of the sensitive information, a query optimizer may reduce the risk of disclosure effectively while maximizing the performance.

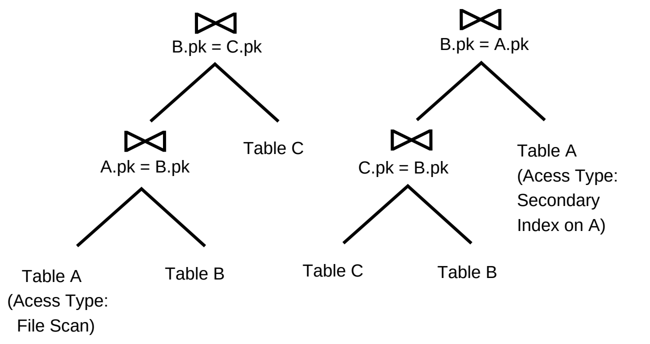


Figure 5: Query execution plans

C. GRANULARITY OF KEY MANAGEMENT

To minimize the risk of disclosure, we propose the idea of "extent level key management". In this approach a unique key is used for each extent to encrypt/decrypt the data pages. An extent is defined as a set of contiguous blocks allocated in a database tablespace. For instance, if the extent size in MySQL-InnoDB is configured as 128KB then eight contiguous data pages can be placed in one extent assuming that 16KB data pages (physical blocks) are used. For OLTP environments the typical size of an extent could vary between 4 to 16 pages whereas for DSS (OLAP) environments larger extent sizes are preferred: 8 to 64 pages per extent. All of the pages within an extent are encrypted with the same key. Whenever a page within an extent is retrieved to the memory the extent key is used to decrypt that page. In our approach extent keys are not stored physically but generated whenever they are needed. To be able to decrypt the content of a page the extent key is requested from the HSM device. This request includes the extent ID. HSM device computes the hash of this id and encrypts the output of this hash function with the master key stored within the HSM. Therefore a MAC based approach is used to generate the extent keys. Once generated the key is used to decrypt the requested page.

To minimize the information disclosure, one can suggest key management at the level of data records, (i.e., one key per record). All of these keys are encrypted with a master key which is protected by a hardware security module. Whenever a row needs to be accessed, the key of this row will be decrypted by the HSM and then used to decrypt the row. This approach would certainly reduce the amount of disclosure during the attack because only the keys used to decrypt the rows in use will be compromised. Although this scheme provides much better protection, it is not preferable due to the performance concerns. First and foremost, for each row access, the HSM needs to be accessed. Such approach would drastically slow down the query processing performance. The second problem is the burden of key initialization. Key initialization is a costly operation and can significantly affect the query processing time if it is performed for each row. The third problem is the encryption/decryption speed of HSM devices. Due to limited processing capacity of these devices, decryption speed of the keys may significantly slow down the data processing. Assuming that 128 bit keys are used, just scanning a table with six million records could require more than one minute extra processing time with a IBM Secure Coprocessor as the HSM device. The fourth problem that we need to address is the storage of encryption keys. An efficient key management strategy should be employed to tackle the issues related to storage of the keys in DBMS. Keeping a single key per row may create a non-negligible storage problem if there is high contention for the shared memory space in the buffer pool.

Key generation in the level of records may not be preferable. However, one could argue that a single key could be used for the encryption of each page. Although this approach provides better protection, we observed that extent level key management provides significant performance improvement compared to the page level key approach. In our preliminary experiments we used a 4764 IBM Secure Coprocessor as the HSM device which is connected to a server machine over PCI-X bus. The execution times of different operations for a single database page are given in Figure 6. The first bar represents the time to generate a single extent key with the HSM device. The second bar and third bar shows the time to decrypt a single data page with the HSM and CPU correspondingly. As seen in this figure decryption speed of the CPU is more than an order of magnitude faster than the HSM. Therefore performing the decryption of bulk data in the server side is much faster. The third and fourth bars show how much time is spent to retrieve a single page to the memory from the disk with a sequential and random disk access pattern correspondingly. Not surprisingly the random access cost is much greater than the sequential access cost due to the physical head movements of the magnetic disk. In OLAP type database applications the disk I/O operations are dominated by the sequential disk accesses because most of the queries require scanning the tables. If page level key generation approach is used the key generation cost would exceed the I/O cost because key generation is twice as costly as reading a page from the disk sequentially. Therefore the query execution time would be dominated by the key generation cost which is not desirable. On the other hand if extent level approach is preferred only a single key generation is required for all pages within an extent. Assuming that we have eight pages within an extent only a single key is generated per eight page accesses. Therefore key generation process does not create a bottleneck while processing the queries.

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

D. OPTIMAL PREDICATE ORDERING

In our model since the pages are decrypted as soon as they are brought into the memory, the disclosure risk is only proportional to the number of different extent keys that are accessed. In an alternate model where one has a choice to postpone the decryption of a page the optimal predicate ordering might differ significantly as illustrated by the following example - consider three selection uncorrelated predicates p_1, p_2 and p_3 with selectivity 0.3, 0.4 and 0.5 respectively. Also, let p_1 be a sensitive predicate (i.e., one that involves a sensitive attribute) and p2 and p3 be predicates over non-sensitive attributes. Then, simply using the performance criteria the optimizer would schedule p_1 to be evaluated first, but this would mean a complete disclosure assuming a table scan is used. Instead, if p_2 and p_3 are evaluated first and the resulting tuples pipelined to p_1, we can expect only 0.2 fraction of the keys to be exposed under the random spreads assumption. In multi-relational queries where join ordering is required, pushing down selections all the way in the query tree may significantly increase the exposure risk in general. A smart way to explore all possible orderings of joins and selections can be carried out by masking a selection predicate as a join with a virtual table containing a single tuple in the range of the predicate. For instance, a condition like $\sigma_{age=50}(R)$ can be modeled as natural join between R and a virtual table $V(age)$ with a single record having value of attribute age as 50. Other common selection conditions can also be represented similarly. Unfortunately, given the exponentially large number of ordering, this technique is likely to degrade the optimizer severely. In a related problem regarding expensive predicate placement in query plans, it might be possible to develop an approach similar to what was done in Hellerstein [sigmod93]. They consider the problem of expensive predicate ordering in query plans and use an optimal constraint aware sequencing algorithm to order the operators optimally be defining a notion of a stream in a query plan. Anyway, we do not go into the details of this alternative model in this paper and this remains an important direction of our future work.

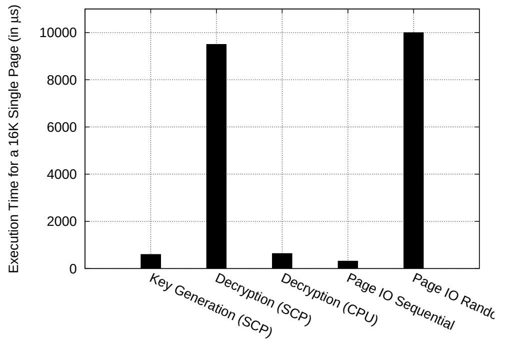


Figure 6: Execution times of operations

E. AN ALTERNATIVE IMPLEMENTATION METHOD FOR CONSTRAINT APPROACH

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

F. ADDITIONAL DETAILS OF THE EXPERIMENTS

F.1 Hardware & Software Specifications

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

As for the tamper resistant hardware we used IBM 4764 PCI-X Cryptographic Coprocessor (See Figure 7). For the cryptographic operations in both the server side and the secure co-processor we used AES in CBC mode with 16 byte key size as the encryption algorithm.

Figure 7: IBM 4764 PCI-X Cryptographic Coprocessor

F.2 Preparation of TPC-H database instance and query workload

The TPC-H benchmark is a decision support benchmark widely used in the database community to assess the performance of very large database management systems [16]. The benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and provide answers to critical business questions. Using the TPC-H schema and queries, we prepared an Operational Data Store (ODS) environment to compare the effectiveness of proposed query optimization approaches. Below, we explain the details of how we prepared the experiment bed and the workloads.

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

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

The major difference between an ODS and a data warehouse (DW) is that the former is used for short-term, mission-critical decisions while the latter is used for medium and long-range decisions. The data in a DW typically spans a five to ten years horizon while an ODS contains data that covers a range of 60 to 90 days or even shorter. In order to simulate an ODS environment, more predicates are added to the "where" clause of the TPC-H queries. This in turn, reduces the number of rows returned. As a result, we obtained a workload comprising of queries which scan both small and large ranges of data. The query given in Table 2 provides an example for this modification.

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

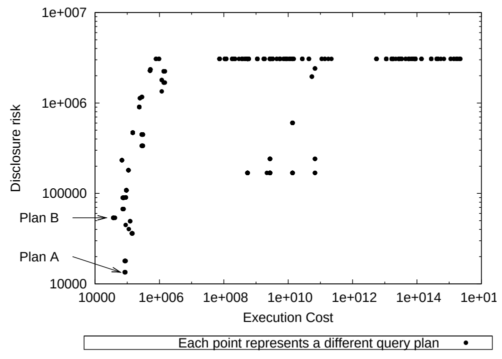


Figure 8: All possible execution plans for a given query

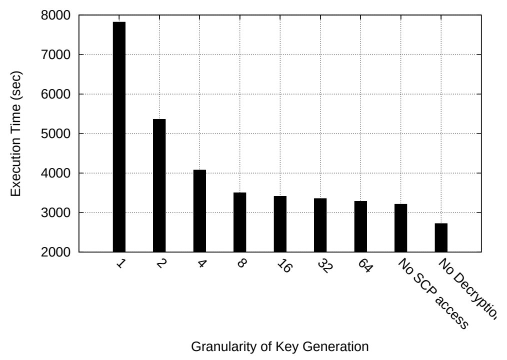


Figure 9: Performance overhead of key generation for different granularities

F.3 Performance Overhead of Key Generation and audit log generation mechanism

Using a similar technique described in Section 4 we prepared a workload of 50 TPC-H queries and run on the modified MySQL database server. In this experiment setting, we created a new TPC-H instance where we assume all attributes of the tables are sensitive and therefore stored in encrypted format in the disk. Each bar in Figure 9 represents the execution time of the workload for different key generation granularity. The left-most bar shows the overall execution time when the HSM device is accessed for each page retrieved from the disk. As the granularity increases the execution time declines. After a certain number of pages (single key per 8 pages) the overhead of key generation remains steady. These results show that key generation per extent would be a good design choice in terms of both performance and disclosure risk. The second right-most bar in this figure shows the execution time when a single key is used for all page decryption operations. Therefore there is no HSM accesses. Note that the execution time in this case is almost similar to the case where the HSM device is accessed per each extent. This observation shows that using extent level key generation incurs no extra cost compared to using a single master key for the database. This, in turn, reduces the disclosure cost while having no extra performance cost. The last bar in the figure shows the execution time when there is no decryption at all. Hence, one can observe the overhead of the decryption operations by comparing this one with the other execution times. We observe 18 % increase in the execution time of the workload when we assume that all accessed tables include sensitive information.

```
Recursive Procedure: findBestPlan
Input: execCost, discCost, bestExec, bestDisc, $S_{tables}$, bestJoinOrder
// $S_{tables}$ is the set of tables to traverse
Output: bestJoinOrder
foreach table $T$ in $S_{tables}$ do
    cummDiscCost = discCost + estimateDiscCost($T$) ;
    cummExecCost = execCost + estimateExecCost($T$) ;
    if OptAlgorithm = "Max. Perf" then
        if cummExecCost > bestExecCost then
            backTrack ;
            // Prune sub-optimal plan

    else if OptAlgorithm = "Min. Disc" then
        if cummDiscCost > bestDisc then
            backTrack ;

    else if OptAlgorithm = "Weighted Aggregation" then
        bestWeightedCost =
        (discWeight * bestDisc) + (perfWeight * bestExec) ;
        cummWeightedCost =
        (discWeight * cummDiscCost) + (perfWeight * cummExecCost) ;
        if cummWeightedCost > bestWeightedCost then
            backTrack ;

    else if OptAlgorithm = "Constraint On Performance" then
        // $\kappa$: minimum execution cost among all possible query plans
        // $\alpha$: tolerance ratio on performance
        maxAllowedCost = $\kappa + \kappa * \alpha\%$ ;
        if cummExecCost > maxAllowedCost and cummDiscCost > bestDisc then
            backTrack ;

    else if OptAlgorithm = "Constraint On Disclosure" then
        // $\kappa$: minimum disclosure cost among all possible query plans
        // $\alpha$: tolerance ratio on disclosure
        maxAllowedCost = $\kappa + \kappa * \alpha\%$ ;
        if cummDiscCost > maxAllowedCost and cummExecCost > bestExec then
            backTrack ;


    // Recursively expand the current partial plan
    if $S_{tables} - T$ is not empty then
        findBestPlan(cummExecCost, cummDiscCost, bestExec,
        bestDisc, $S_{tables} - T$, bestJoinOrder) ;

    if OptAlgorithm = "Max. Perf" then
        if cummExecCost < bestExec then
            // A better path is available
            bestExec = cummExecCost ;
            updateBestJoinOrder(T, bestJoinOrder) ;

    else if OptAlgorithm = "Min. Disc" then
        if cummDiscCost < bestDisc then
            // A better path is available
            bestDisc = cummDiscCost ;
            updateBestJoinOrder(T, bestJoinOrder) ;

    else if OptAlgorithm = "Weighted Aggregation" then
        bestWeightedCost =
        (discWeight * bestDisc) + (perfWeight * bestExec) ;
        cummWeightedCost =
        (discWeight * cummDiscCost) + (perfWeight * cummExecCost) ;
        if cummWeightedCost < bestWeightedCost then
            // A better path is available
            bestExec = cummExecCost ;
            bestDisc = cummDiscCost ;
            updateBestJoinOrder(T, bestJoinOrder) ;

    else if OptAlgorithm = "Constraint On Performance" then
        // $\kappa$: minimum execution cost among all possible query plans
        // $\alpha$: tolerance ratio on performance
        maxAllowedCost = $\kappa + \kappa * \alpha\%$ ;
        if cummExecCost < maxAllowedCost and cummDiscCost < bestDisc then
            // A better path is available
            bestExec = cummExecCost ;
            bestDisc = cummDiscCost ;
            updateBestJoinOrder(T, bestJoinOrder) ;


    else if OptAlgorithm = "Constraint On Disclosure" then
        // $\kappa$: minimum disclosure cost among all possible query plans
        // $\alpha$: tolerance ratio on disclosure
        maxAllowedCost = $\kappa + \kappa * \alpha\%$ ;
        if cummDiscCost < maxAllowedCost and cummExecCost < bestExec then
            // A better path is available
            bestExec = cummExecCost ;
            bestDisc = cummDiscCost ;
            updateBestJoinOrder(T, bestJoinOrder) ;


end
```

Algorithm 1: Pseudocode of the modified query optimization algorithm (with additional pruning and decision steps)