Engineering Security and Performance with Cipherbase

Arvind Arasu   Spyros Blanas   Ken Eguro   Manas Joglekar   Raghav Kaushik
Donald Kossmann   Ravi Ramamurthy   Prasang Upadhyaya   Ramarathnam Venkatesan

Abstract

Cipherbase is a full-fledged relational database system that leverages novel customized hardware to store and process encrypted data. This paper outlines the space of physical design options for Cipherbase and shows how application developers can implement their data confidentiality requirements by specifying the encryption method for static data storage and the acceptable information leakage for runtime data processing. The goal is to achieve a physical database design with the best possible performance that fulfills the application's confidentiality requirements.

1 Introduction

Data confidentiality is one of the main concerns of users of modern information systems. Techniques to protect data against curious attackers are particularly important for users of public cloud services [7]. For instance, a biologist who has carried out in-vitro experiments over several years and stores the results for further analysis in a public database cloud service (e.g., [9]) wants to make sure that her competitor does not have access to her results before she was able to publish them. Data confidentiality, however, is also critical in private clouds in which competitors may pay database administrators of a company to steal confidential business secrets.

Cipherbase is an extension of Microsoft SQL Server, specifically designed to help organizations leverage an efficient "database-as-a-service" while protecting their data against "honest-but-curious" adversaries. In particular, Cipherbase can protect the data against administrators that have root access privileges on the database server processes and the machines that run these processes. Administrators need these access privileges to do their job such as creating indexes, repartitioning the data, applying security patches to the machines, etc. However, these administrators do not need to see and interpret the data stored in the databases of those machines. Cipherbase features a novel hardware / software co-design that leverages customized hardware (based on FPGAs [5]) to securely decrypt, process, and re-encrypt data without giving externals who have access to the system a way to sniff the corresponding plaintext. One particular feature of Cipherbase is that it supports *configurable security*. Not all data stored in a database is sensitive. For instance, a great deal of Master data such as names of countries or exchange rates are public knowledge and need not be protected against curious attackers. Some data such as vendor addresses might be confidential, but a weak encryption might be sufficient. Information leakage of that data is embarrassing, but not a disaster. Other information such as customer PII (personally identifiable information) needs to be strongly encrypted. Cipherbase allows the specification of the confidentiality requirements of all data in a declarative way while preserving the general purpose functionality of a database system.

Copyright 2012 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

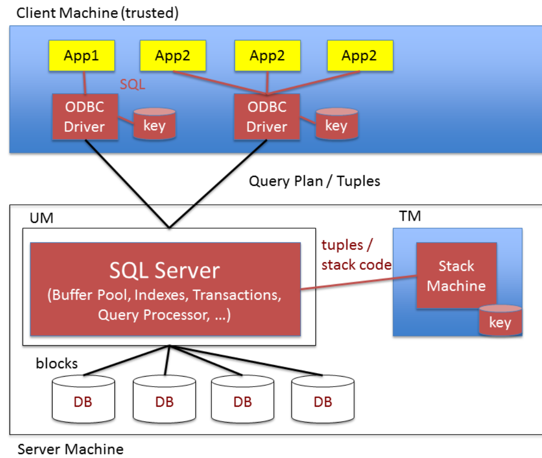

Figure 1: Cipherbase Architecture

1.1 System Overview

In this section, we give a brief overview of the Cipherbase system and contrast it with related work [10, 3](for more details on the system refer to [2]). The architecture of Cipherbase is shown in Figure 1. In a nutshell, Cipherbase extends all major components of Microsoft SQL Server. It extends the ODBC driver to keep encryption keys; this way, all encryption and decryption is transparent and applications only see clear text values so that they need not be rewritten. The SQL Server transaction and query processors at the server side are extended to make use of the trusted machine (referred to as TM in Figure 1) to process strongly encrypted data. The bulk of the SQL Server code is unchanged and runs on commodity servers (e.g. x86 processors), referred to as UM for untrusted machine in Figure 1. The TM is equivalent to an "oracle" that can compute on encrypted data without revealing the cleartext. The TM is implemented as a stack machine (Figure 1) which supports the ability to evaluate arbitrary predicates on encrypted data. The encryption keys are also stored in the TM in a tamper-proof manner and thus plaintext corresponding to encrypted data is not visible in the UM. This way, Cipherbase is able to support any kind of SQL query or update and preserve transactional semantics using the well-tested SQL Server code base. This allows us to maintain confidentiality by processing operations on encrypted data in the trusted machine in a fine-grained way whenever necessary.

In order to enable an efficient database-as-a-service, we need the ability to operate on encrypted data in the cloud. While generic homomorphic encryption techniques that enable arbitrary computation on encrypted data [6] are not yet practical, there are encryption techniques that enable executing different database operations directly on the encrypted data. For instance, if two columns are encrypted using deterministic encryption, the join can be evaluated by evaluating the join on the corresponding encrypted columns. CryptDB [10] is a recent system that exploits such partial homomorphic properties of various encryption schemes to support a subset of SQL queries. However, this approach has important limitations. First, CryptDB cannot handle generic and ad-hoc SQL queries. For instance, there is no known partial homomorphic encryption scheme that can handle even simple aggregates such as $price * (1.0 - discount)$. Thus, any ad-hoc query that computes such aggregates over a table needs to ship the entire table to a trusted proxy in order to decrypt the data to evaluate the aggregate (which would diminish the cost-benefit trade-offs in using the cloud in the first place). Second, even for the class of operations supported in CryptDB, the security is not configurable. For instance, if a query requires to sort a particular column, CryptDB would reorganize the column to be stored using order preserving encryption [4]. In contrast, Cipherbase provides *orthogonality* - it allows an application developer to choose an encryption policy independent of the query workload - a query can perform operations on strongly encrypted data (such as sorting a column) by leveraging the TM.

Our fine-grained integration of the UM/TM is also fundamentally different from the loosely coupled design

of TrustedDB [3], that combines an IBM secure co-processor (SCP) and a commodity server (TrustedDB runs a lightweight SQLite database on SCP and a more feature-rich MySQL database on the commodity server). The fine-grained integration in Cipherbase enables a smaller footprint for the trusted hardware module as well as novel optimizations. For instance, if all columns were encrypted, the loosely coupled approach is constrained to use the limited computational resources at the TM for functionality that does not depend on encryption (e.g., disk spills in hash join). In contrast, Cipherbase adopts a tightly coupled design in which only the core primitives from each module of the database system that need to operate on encrypted data are factored and implemented in the TM. Interestingly, a small class of primitives involving encryption, decryption, and expression evaluation (see [2] for more details) suffices to support query processing, concurrency control, and other database functionality. For the hash join example, the hash join operator uses the TM for computing hashes and checking equality. The rest of hash join logic including memory management, writing hash buckets to disk, and reloading them all run in the UM (for a more detailed comparison with Trusted DB see [2]).

In terms of security, the TM is equivalent to an "oracle" that can compute on encrypted data without revealing the clear-text. Therefore, the information revealed is the results of operations used during query processing - a filter operator reveals identities of records satisfying the filter predicate, a sort operator reveals the ordering of records and a join operator reveals the join graph. The security of our basic system is similar to the security offered by CryptDB [10]. However, there are key differences. Even when computation is performed on a column, we only reveal information about the subset over which computation happens. For instance, if a subset of a table is sorted, we only reveal the ordering for the subset; in contrast, CryptDB reveals the ordering of the whole column. Second, we do not change the data storage, whereas CryptDB changes the data on disk to use weaker encryption and hence reveals information even to a weaker adversary who only has access to the disk.

As mentioned earlier, Cipherbase supports configurable security - it allows the specification of the confidentiality requirements of all data in a declarative way. More importantly, Cipherbase exploits this information to optimize query and transaction processing. For instance, public data can be processed in the untrusted machine (using conventional hardware) in the same way and as efficiently as with the existing SQL Server engine. Queries and transactions that involve public and confidential data are executed in both the UM and TM, thereby exploiting the UM as much as possible, minimizing use of the expensive TM, and minimizing data movement between the UM and TM. However, the act of query processing can itself leak information for an adversary who can monitor the "access-patterns" of query evaluation. In Cipherbase we take first steps in addressing this problem and provide mechanisms that can be used to mitigate this dynamic information leakage.

The purpose of this paper is to give details of the language extensions that allow users to specify the confidentiality requirements of their data (for more details on the system architecture and implementation refer to [2]). It turns out that confidentiality needs to be defined along two dimensions: (1) *static security* for encrypting the data stored in the disk (Section 3) and (2) *runtime security* to constrain the choice of algorithms to process data in motion because different algorithms imply different kinds of information leakage (Section 4). Furthermore, this paper outlines a number of tuning techniques (Section 5) that help to improve the performance of processing confidential data in a system like Cipherbase. The goal is to achieve the confidentiality that is needed at the lowest possible cost.

2 Static Data Security

Cipherbase supports a variety of different encryption techniques in order to meet a broad spectrum of privacy needs and allow users to tune their physical design for better performance. If privacy were the only concern, then all data should be encrypted in a strong and probabilistic way (e.g., using AES in CBC mode). While Cipherbase is able to support such a setting and execute any SQL query and update statement on such strongly encrypted data, the performance would be poor in many situations and the overhead would be much worse than necessary to fulfill the confidentiality needs (as mentioned earlier, for instance, some data need not be encrypted

because it is public). We are currently integrating the following encryption techniques into Cipherbase:

- AES in ECB mode: This is a deterministic encryption technique which allows processing equality predicates on the encrypted data without decrypting the data. That is, any equality predicate can be processed in the UM entirely without shipping and processing any data in the TM.
- ROP [4]: This is an order-preserving encryption technique which allows processing range predicates, sorting, and Top N operations entirely in the UM. Aggregation or any kind of arithmetic or SQL intrinsic functions, however, must be carried out in the TM.
- Homomorphic: For certain arithmetic operators (e.g., addition and multiplication), practical encryption techniques are known. Generic homomorphic encryption [6] that, for instance, supports both addition and multiplication, however, is still not practical.
- AES in CBC mode: This is a strong, probabilistic encryption technique. Any kind of operation on this data needs to be carried out in the TM.

Our approach is based on the observation that data confidentiality needs are a *static* property of an application and do not depend on the query and update workload. For instance, compliance regulations might be the reason to encrypt some data strongly (e.g., the disease of patients in a health care domain) and some data not at all (e.g., cities of patients). These regulations are known statically and do not change as a result of executing the application. As a result, the data confidentiality needs should be specified in the schema as part of the SQL DDL by annotating the columns of a table with the encryption scheme that should be used to protect that data. The SQL query language and DML need not be changed. This design to specify confidentiality statically is in contrast to the design of CryptDB [10], a related database system for managing confidential data which adapts the encryption technique dynamically to the needs of the query workload. As discussed in Section 2, if processing of a query requires an order-preserving encryption technique and the column is currently strongly encrypted, then CryptDB will degrade the encryption scheme as a side-effect of processing this query. This design may result in performance jitter (reorganizing a whole table while processing, e.g., a fairly simple range query) and in unintended information leakage (degrading the encryption).

Figure 2 gives an example of a simple schema for patient and disease information. All patient information is considered to be confidential. The patient names are strictly confidential so that the schema of Figure 2 specifies that they be encrypted using AES. As *names* are unique in the *Patient* table, deterministic encryption (ECB mode) is sufficient. *Patient.age* is less critical; in particular, if the age cannot be associated to a name. As a result, *Patient.age* can be encrypted using a weaker, order-preserving technique such as ROP [4]. Disease information is public and the diagnosis information is again highly confidential because it belongs to patients.

As shown in Figure 2, any kind of integrity constraint can be implemented, independent of the encryption technique. Just as for regular query processing, however, the choice of the encryption technique impacts the performance of maintaining integrity constraints. In the schema of Figure 2, for instance, the `check` constraint of *Patient.age* can be validated by the UM without decrypting the patient information whenever a new patient is inserted or updated because an order-preserving encryption technique was chosen. If the age were encrypted using AES, then the TM would have to check the predicate of the integrity constraint.

A number of scenarios are conceivable to encrypt primary and foreign keys and test for referential integrity. Figure 2 shows two scenarios. First, *Diagnosis.patient* is encrypted in a stronger way than *Patient.name*; i.e., probabilistic encryption in CBC mode vs. deterministic encryption in ECB mode. The motivation for using a probabilistic encryption technique for *Diagnosis.patient* might be that some patients are sick more often and we would like to hide this fact from a potential attacker. Doing so comes at a cost: The predicate evaluation of every *Patient* $\bowtie$ *Disease* needs to be carried out by the TM. Only if key and foreign key are encrypted using the same technique and this technique is deterministic (e.g., AES in ECB mode), a join can be carried out entirely in the UM without decrypting the join keys. As a result, application developers should carefully choose the encryption

```
create table Patient (
 name      :  VARCHAR(50)  AES_ECB primary key,
 age       :  VARCHAR(50)  ROP check >= 0);

create table Disease (
 name      :  VARCHAR(50)  primary key,
 descr     :  VARCHAR(256));

create table Diagnosis (
 patient   :  VARCHAR(50)  AES_CBC references Patient,
 disease   :  VARCHAR(50)  AES_ECB references Disease,
 date      :  DATE check not null);
```

Figure 2: Specifying Confidentiality Needs

techniques of foreign keys and only diverge from the encryption technique of the primary key if absolutely necessary.

The second scenario shown in Figure 2 involves the *Diagnosis.disease* / *Disease.name* foreign key / key relationship. In this case, the foreign key is encrypted and the referenced primary key is not encrypted. Again, the motivation for encrypting the foreign key here might be that no information of the occurrence of certain diseases should be leaked to a potential attacker. Again, this situation of having different encryption techniques for join keys results in expensive *Diagnosis* $\bowtie$ *Patient* joins because the join predicate needs to be evaluated in the TM, thereby decrypting *Diagnosis.disease*. Curiously, in this example, better performance can be achieved by encrypting *Disease.name* using AES in ECB mode. That is, it might be beneficial to encrypt data for performance reasons even though the data is public and does not need to be protected. In general, there are many different ways to encrypt foreign keys and primary keys with different performance implications. [8], for instance, discusses an approach to probabilistically encrypt primary keys and foreign keys in concert in order to effect efficient joins without decrypting the keys.

3 Runtime Data Security

The previous section discussed encryption techniques and their performance implications to protect the data stored in the disk. This section discusses the performance implications in *runtime data security*. To motivate this discussion, the following simple example illustrates how the processing of data can leak information even if the UM never sees any unencrypted data.

Patient	*Disease*
Alice	!@#$xyz
Bob	@%^abc
Chen	*&#pqr
Dana	(p#z~94

Figure 3: Sample Diagnosis Table

Example 1: Figure 3 shows an example *Diagnosis* table. For brevity, the *date* column is not shown. To illustrate this example, it is assumed that the *patient* column is not encrypted and the *disease* column is strongly encrypted. That is, patient names are public information, but it should not be revealed which patients have been diagnosed with which disease. Now, consider a query that asks for the number of patients who have been diagnosed with a particular disease (e.g., AIDS). The predicate of this query needs to be executed by the TM because the *disease*

column is encrypted. Nevertheless, a naïve execution of this query in which each tuple is checked individually by the TM can reveal information to a system administrator who is able to monitor the query execution. The naive execution of the filter operator would pass each encrypted tuple to the TM, which would decrypt the tuple and evaluate the predicate and return either true or the tuple again (which contains the patient name in cleartext). Thus, a system administrator who can monitor the communication between the TM and UM can: 1) infer that the multiple patients output by the filter operator have the same disease 2) infer the disease of the patients output by the filter operator (if he has additional background information that the disease in question is AIDS). Note that this is despite the fact that the column is stored strongly encrypted and never decrypted outside the TM.

The key observation is that query processing techniques and algorithms leak certain kinds of information (particularly when the schema involves a mix of both cleartext attributes and encrypted attributes). Cipherbase allows application developers to specify which kind of information may be leaked as a part of query processing by annotating the schema in the same way as specifying static data security. For instance, the designer of the schema of the *Diagnosis* table could specify that no dynamic information leakage is allowed for the *disease* column. As a result, Cipherbase would apply an appropriate algorithm to evaluate a query that asks for the number of AIDS patients.

Figure 4 lists the options that Cipherbase supports for runtime data security (similar to static security these knobs are specified on a per-column basis). The higher the confidentiality requirements, the more constrained the choice of query processing techniques and the more work needs to be carried out in the TM. In other words, the higher the confidentiality requirements, the worse the performance in many situations. We briefly describe the different levels and their performance implications in the remainder of this section (see [2] for a more detailed discussion on the implementation of the levels).

Dynamic Leakage	*Computation in TM*
Default	Expressions
Card	Expressions, defer & encrypt output
Blob	Whole operators, process blocks

Figure 4: Confidentiality Levels for Runtime Data Security

Default: This is the default level of runtime security that protects data on disk and in addition, it protects the data against attackers who can read the main memory contents of the database server machine. As a result, this level requires that data be kept in main memory in an encrypted form and may only be decrypted on the fly (in the TM) for query processing. This level requires the use of a trusted hardware (i.e., TM) to process, say, predications or aggregates on encrypted data, but it allows the use of the naïve algorithm to evaluate predicates of tuples. In other words, it does not protect against the monitoring attack described in Example 1.

Cardinality: This level only reveals the cardinality of intermediate and final query results (that include the sensitive column). So, if a query asks for all patients that have been diagnosed with AIDS, the attacker would be able to learn the total number of patients who have been diagnosed with AIDS, but the attacker would not be able to infer the name of a single AIDS patient. To implement this level, Cipherbase *defers* the evaluation of a predicate for a tuple [2]. In Example 1, for instance, if Alice was diagnosed with AIDS and the TM is asked to process Alice, then the TM would not immediately return Alice as a match when probed with Alice. Instead, the TM would remember all matching tuples and start returning matching tuples (in which all columns including any plain text columns are encrypted) at a later point. In addition, the order of the returned tuples is randomized. For instance, it could return Alice (in an encrypted form) when processing the tuple in the TM corresponding to Dana.

Blob: This level is essentially equivalent to storing the column as a blob. To implement this level, Cipherbase needs to implement whole operators of the relational algebra in the TM. To process queries, blocks of tuples that are encrypted as a whole are shipped to the TM. The TM decrypts these blocks and then carries out bulk operations (e.g., partitioning for a hash join) on these tuples. Furthermore, the TM returns only encrypted blocks of tuples (possibly even the empty block of tuples) so that no information can be inferred by observing the data returned from the TM. This option could have significant performance penalties for more complex queries (and could also involve more significant processing in the client). Even in the case of Example 1, this option has to return a constant number of blocks as output independent of the selectivity of the predicate (to ensure that the cardinality of the filter operator is not revealed) and thus, this option must be only used only for columns that require high confidentiality.

As with the case of static security, the options for runtime data security can also be specified declaratively as DDL annotations. For the example database shown in Figure 3, the annotation would be as follows.

```
create table Diagnosis (
name    :  VARCHAR(50) references Patient,
disease :  VARBINARY  AES_CBC  BLOB references Disease);
```

4 Other Tuning Options

Specifying the encryption method and the degree of information leakage as described in the previous two sections are the most critical decisions for the physical design of a database like Cipherbase. This section discusses additional considerations for a good physical design in Cipherbase.

Horizontal clustering: Some schemas involve a great deal of flags (e.g., *is vegetarian*) or small fields that can be represented using a few bits (e.g., *status*). Encrypting each of these fields individually is wasteful. For instance, if a flag (1 bit) is encrypted using AES (128 bits), then the size of the database can grow by two orders of magnitude. One way to reduce this space overhead without sacrificing confidentiality is to cluster a set of attributes of a row and encrypt them together into a single (128 bit) cipher.

Vertical clustering: An alternative to the horizontal clustering and encryption of several fields within a row is the vertical clustering of the values of the same column of several rows. For instance, the *status* of, say, 16 records could be clustered and encrypted into a single cipher. This approach resembles the PAX storage layout proposed in [1]. Vertical clustering can be applied even if there is only a single small column in a table, but it is more difficult to integrate into a query processor if the query processor is not already based on a PAX storage layout.

Indexing and Materialized Views: Indexes and materialized views can be defined using Cipherbase. These indexes and materialized views inherit the encryption scheme from the referenced data and take the restrictions for dynamic information leakage into account. For instance, the entries of a *Patient.name* index (for the schema described in Figure 2) would be encrypted using AES in ECB mode. Point lookups (i.e., equality predicates) could be processed using that index entirely in the UM without decrypting any keys because AES in ECB mode is deterministic. Thus, a hash index on the Patient.name column can run completely in the UM. However, processing range predicates using a B-Tree index on the column(e.g., *Patient.name LIKE "Smith%"*) would have to be processed using the TM. Currently we do not support indexes for columns that include knobs for runtime security (this would require the integration of oblivious RAM techniques in the storage subsystem). Query processing with indexes in Cipherbase is discussed in more detail in [2].

Setting Isolation Levels: The isolation level is an important tuning knob in any database system. This observation is also true for Cipherbase. In Cipherbase, however, it is particularly important because it can impact which operations can be carried out in the UM and which operations need to be carried out in the TM. Using serializability, for instance, the lock manager must interact with the TM in order to check predicates on encrypted data for key range locking. Using lower isolation levels such as snapshot isolation, the actions of the lock manager can be fully executed in the UM.

Statistics: In general, the presence and maintenance of statistics such as those needed for query optimization can be a source for information leakage. In the current design of Cipherbase, however, this kind of information leakage is precluded: All statistics are kept in an encrypted form at the server and query optimization which requires these statistics in cleartext is carried out at the (trusted) client machines (Figure 1).

5 Conclusion

The goal of the Cipherbase system is to help developers to protect their confidential data against curious attackers (e.g., database administrators) and achieve high performance at the same time. Just as in any other database system, the physical database design is crucial to achieve high performance. This paper discussed the most critical physical design decisions that are specific to a system like Cipherbase. Concretely, this paper discussed the performance implications of static data security (choice of encryption scheme), runtime data security (kind and amount of work that needs to be carried out by trusted hardware), and other tuning considerations such as clustering rows and columns. We are currently prototyping Cipherbase and in the process of evaluating and quantifying the trade-offs in using the different physical design options outlined in this paper. We believe that Cipherbase is particularly suited to be used as the infrastructure for a secure database-as-a-service where the goal is to achieve the confidentiality that is needed at the lowest possible cost.

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, pages 169–180, 2001.

[2] A. Arasu et al. Orthogonal security with cipherbase. In *CIDR*, 2013.

[3] S. Bajaj and R. Sion. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *SIGMOD*, 2011.

[4] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *EUROCRYPT '09*, 2009.

[5] K. Eguro and R. Venkatesan. FPGAs for trusted cloud computing. In *FPL*, 2012.

[6] C. Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3), 2010.

[7] H. Hacigumus, S. Mehrotra, and B. R. Iyer. Providing database as a service. In *ICDE*, pages 29–38, 2002.

[8] S. Hildenbrand, D. Kossmann, T. Sanamrad, C. Binning, F. Faerber, and J. Woehler. Query processing on encrypted data in the cloud. In *Technical Report No. 735, ETH Zurich*, 2011.

[9] Microsoft Corporation. SQL Azure. http://www.windowsazure.com/en-us/home/features/sql-azure/.

[10] R. A. Popa, C. M. S. Redfield, N. Zeldovich, et al. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.