Confidentiality for Database Applications with Encrypted Query Processing

Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan
{ralucap, cat_red, nickolai, hari}@mit.edu

Abstract

Online applications are vulnerable to the theft of sensitive information because adversaries can exploit software bugs to gain access to private data, and because curious or malicious administrators may capture and leak data. EncDB is a system that provides practical and provable confidentiality in the face of these attacks for applications backed by SQL databases. It works by fully *executing SQL queries over encrypted data* using a collection of efficient SQL-aware encryption schemes. EncDB also *chains encryption keys to user passwords*, so that a data item can be decrypted only using the password of one of the users with access to that data. As a result, even if all servers are compromised, the adversary cannot decrypt the data of any user that is not logged in. Our evaluation shows that EncDB has low overhead: on a real application, phpBB, EncDB reduces throughput by 13%, and on the TPC-C benchmark by 27% compared to regular Postgres. Chaining encryption keys to user passwords requires 11–13 unique schema annotations for database schemas of three multi-user web applications to capture their policies.

1 Introduction

Theft of private information is a significant problem, particularly for online applications [31]. An adversary can exploit various software vulnerabilities [27] and bugs to gain unauthorized access to servers; curious or malicious administrators at a hosting or application provider can snoop on private data [6]; and attackers with physical access to servers can cause significant damage [19].

One approach to reduce the damage caused by server compromises is to encrypt sensitive data, as in SUNDR [23], SPORC [15], and Depot [25], and run all computations (application logic) on clients. Unfortunately, several important applications don't lend themselves to this approach, such as a database-backed web site that processes queries to generate data for the user, or an application that computes over large amounts of data. Even when this approach is tenable, converting an existing server-side application to this form can be difficult. On the other hand, one might consider theoretical solutions such as fully-homomorphic encryption [17] to allow servers to compute over encrypted data. However, these schemes are currently wildly impractical, with slowdown estimates on the order of a trillion times [10].

This paper presents EncDB, a system that explores an intermediate design point to provide practical privacy guarantees for applications that use database management systems (DBMSs). EncDB leverages the typical structure of database-backed applications, consisting of a DBMS server and a separate application server, as shown in Figure 1; the latter runs the application code and issues DBMS queries on behalf of different users. EncDB's main idea is to *execute queries over encrypted data* and the key insight that makes it practical is that SQL uses a well-defined set of operators, each of which we are able to run over encrypted data.

EncDB addresses two threats. The first threat is a *curious database administrator* (DBA) who tries to learn private data (e.g., health records, financial statements, personal information, etc.) by snooping on the DBMS server; here, EncDB ensures that the DBA cannot extract any private data. The second threat is an *adversary that gains complete control of application and DBMS servers*. In this case, EncDB cannot provide any guarantees for users that are logged-in the application during an attack, but still guarantees the confidentiality of other users' data.

There are two challenges in combating these threats. The first lies in minimizing the amount of data revealed to the DBMS server, without sacrificing the ability to execute queries efficiently. Using a single key to encrypt each row in the database would not allow the DBMS server to execute many SQL queries without access to the decryption key; consider, for instance, a query that asks for the average salary of employees or for the names of employees whose salary is greater than $60,000. These queries require computing over encrypted data, but existing approaches that provide this capability are either too slow for real-world use or don't provide adequate confidentiality. An ideal solution should impose a low performance overhead on the DBMS server, avoid executing queries outside the DBMS, and run on unmodified DBMS server software to benefit from decades of engineering effort in optimizing DBMS performance.

The second challenge is to minimize the amount of data leaked when an adversary compromises the application server. An ideal solution would ensure that the application can issue queries only for data it legitimately

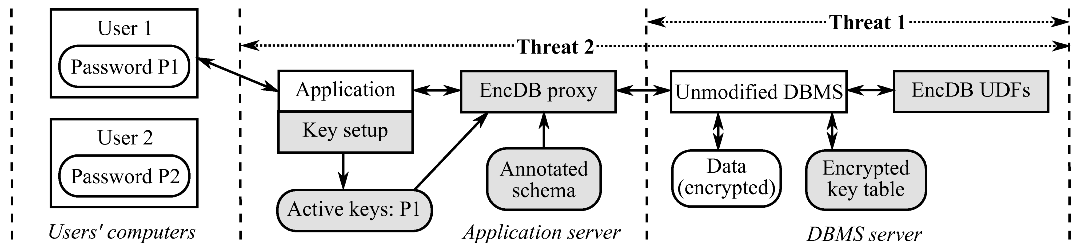

Figure 1: EncDB's architecture, made up of two parts: a *database proxy* and an unmodified *DBMS*, which uses user-defined functions (UDFs) to implement components of EncDB. Rectangular and rounded boxes represent processes and data, respectively. Shading indicates components added by EncDB. Dashed lines indicate separation between users' computers, the application server, and the DBMS server. EncDB addresses two kinds of threats, shown as dotted lines. In threat 1, a curious database administrator with complete access to the DBMS server snoops on private data, in which case EncDB leaks no data. In threat 2, an adversary gains *complete* control over both the software and hardware of the application and DBMS servers, in which case EncDB leaks no data of currently inactive (not logged in) users (data of user 1 in this example may leak).

requires, and not for arbitrary data requested by an adversary. However, this task is difficult because the DBMS server can also be subverted by the attacker, so EncDB cannot trust it to prevent the application from reading arbitrary data. Moreover, the application itself has legitimate access to EncDB's decryption keys, and assigning each user a different database encryption key is inadequate for applications with shared data, such as a bulletin board or a conference review site.

EncDB addresses these challenges using three novel ideas. The first is to *execute SQL queries over encrypted data*. EncDB implements this idea using an *SQL-aware encryption strategy*, which leverages the fact that all SQL queries are made up of a well-defined set of primitive operators, such as equality checks, order comparisons, aggregates (sums), and joins. By adapting known encryption schemes (for numeric and string equality, additions, and order checks) and using a new privacy-preserving cryptographic method for joins, EncDB encrypts each data item in a way that allows the DBMS to execute on the transformed data. EncDB is efficient because it mostly uses symmetric-key encryption, avoids fully-homomorphic encryption, and runs on unmodified DBMS software (with user-defined functions).

Some encryption schemes leak more information about the data to the DBMS server than others, but are required to process certain queries. To avoid revealing all possible encryptions of data to the DBMS *a priori*, EncDB carefully *adjusts* the SQL-aware encryption scheme for any given data item, depending on the queries observed at run-time. To implement these adjustments efficiently, EncDB uses *onions of encryption*. Onions are a novel way to compactly store multiple ciphertexts within each other in the database and avoid expensive re-encryptions.

The third idea is to *chain encryption keys to user passwords*, so that each data item can only be decrypted through a chain of keys rooted in the password of one of the users with access to that data. As a result, if the user is not logged into the application, and if the adversary does not know the user's password, the adversary cannot decrypt the user's data, even if the DBMS and the application server are compromised. To construct a chain of keys that captures the application's data privacy and sharing policy, EncDB allows the developer to provide policy annotations over the application's SQL schema, specifying which users (or other principals, such as groups) have access to various data items.

We have implemented EncDB in a way that should work with most standard SQL DBMSes, and ran it on both unmodified Postgres and MySQL servers. We ran EncDB on three multi-user web applications: phpBB (a popular online bulletin board), HotCRP (a conference review system), and grad-apply (our university's graduate admission system) and on the industry-standard TPC-C benchmark. EncDB secured all sensitive fields in the three applications with only 11–13 unique annotations, which expressed a variety of policies (including an interesting new one in HotCRP for handling papers in conflict with a PC chair). EncDB supported all queries on encrypted data from these applications. Compared to an unencrypted DBMS, EncDB reduced throughput by a relatively modest amount of 13% for phpBB and 27% for TPC-C.

2 Security Overview

Figure 1 shows EncDB's architecture and threat models. EncDB works by intercepting all SQL queries in a *database proxy*, which rewrites queries to execute on encrypted data. The proxy encrypts and decrypts all data, and changes some query operators, while preserving the semantics of the query. The DBMS server never receives decryption keys to the plaintext so it never sees sensitive data, ensuring that a curious DBA can never learn private information (threat 1).

To guard against application server compromises together with DBMS compromises (threat 2), developers may annotate their SQL schema to define different principals, whose keys will allow decrypting different parts of the database and make a small change to their applications to provide encryption keys to the proxy as described in §4. The EncDB proxy determines what parts of the database should be encrypted under what key. The result is that EncDB guarantees the confidentiality of data belonging to users that are not logged in during a compromise (e.g., user 2 in Figure 1), and who do not log in until the compromise is detected and fixed by the administrator.

We now describe the two threat models addressed by EncDB, and the security guarantees provided under those threat models.

2.1 Threat 1: DBMS Server Compromise

In this threat, EncDB guards against a curious DBA who has full access to read the data stored in the DBMS server, or against a *passive* adversary who manages to gain access to the DBMS machine. This includes protection against DBMS software compromises, root access to DBMS machines or even access to the RAM of physical machines. With the rise in database consolidation inside enterprise data centers, outsourcing of databases to public cloud computing infrastructures, and the use of third-party DBAs, this threat is increasingly important.

To overcome this threat, EncDB executes SQL queries over encrypted data on the DBMS server. In so doing, the proxy uses some secret keys to encrypts all data inserted in the DB and queries reaching the DBMS. To compute query results, the proxy needs to reveal certain relationships among data to the DBMS. By using SQL-aware encryption that adjusts dynamically to the queries presented, EncDB reveals only relations between tuples that are necessary for the server to execute the query. EncDB provides the following properties:

- Sensitive data is never decrypted at the DBMS server, which never gets the decryption key to the plain data.
- If the application requests no relational predicate filtering on a column, nothing about the column content leaks. If the application requests equality checks on a column, EncDB's proxy reveals which items repeat in that column. If the application requests order checks on a column, the proxy reveals the order of the elements in the column. EncDB never reveals the plaintext data to the DBMS server.
- The DBMS server cannot process queries different from the ones requested by the application.

We believe that these confidentiality properties are about as good as one can provide in practice with current encryption technologies.

In §7, we show that all sensitive fields in the tested applications remain encrypted with highly-secure encryption schemes, leaking almost nothing about their content, while some semi-sensitive fields benefit from flexible encryption schemes. In addition, EncDB allows the developer to optionally specify the lowest security level allowed for a column. For example, the developer may set the social security number or credit card field to allow at most equality checks, but not reveal order within the column. The proxy will disallow queries that are inconsistent with the specified security setting.

We protect confidentiality against any arbitrary attack on the database, under the assumptions that the attacker does not change queries coming from the application, query results or data in the DB and the application and proxy are free of compromise. For the first condition, we believe most DBA's are more likely to attempt to read the data than to dare to change the data or the query results. In §8, we cite related work concerning data integrity that could be used in complement with our work. For the second condition, an adversary that modifies query results may be able to trick the application into, for example, sending a user's data to the wrong email address, when the user asks the application to email him a copy of his own data. Such active attacks on the DBMS fall under the second threat model, which we will now discuss.

2.2 Threat 2: Arbitrary Threats

We now describe the second threat where the application server, proxy, and DBMS server infrastructures may be compromised arbitrarily. The approach in threat 1 is insufficient because an adversary compromising the application server can get access to the keys used to encrypt the entire database.

The solution is to encrypt different data items (e.g., data belonging to different users) with different keys. To determine the key that should be used for each data item, developers annotate the application's database schema to express finer-grained confidentiality policies. A curious DBA still cannot obtain private data by snooping on the DBMS server (threat 1), but in addition, an adversary who compromises the application server or proxy can now decrypt only data encrypted under the keys of currently logged-in users (stored in the EncDB proxy). Data of currently inactive users would be encrypted with keys not available to the adversary, and would remain confidential.

In this configuration, EncDB provides strong guarantees in the face of *arbitrary* server-side compromises, including gaining root access to the application server. EncDB at most leaks *the data of currently active users for the duration of the compromise*. By "duration of a compromise", we mean the interval from the start of the compromise until any trace of the compromise has been erased from the system. For a read SQL injection, the duration of the compromise is the moment when the attacker's queries get executed. In the above example of an adversary changing the email address of a user in the database, we consider the system compromised as long as the attacker's email address persists in the database.

Guarantees other than confidentiality, such as data integrity, are outside of EncDB's scope, as are attacks on user machines, such as cross-site scripting.

3 Queries over Encrypted Data

This section describes how EncDB executes SQL queries over encrypted data. The threat model in this section is threat 1 from §2.1; the DBMS machines and administrators are not trusted, but the application and the proxy are trusted.

EncDB enables the DBMS server to execute SQL queries on encrypted data almost as if it were executing the same queries on plaintext data. Existing applications do not need to be changed because the proxy exports the same SQL interface as the DBMS. The DBMS's query plan for an encrypted query is the same as for the original query. However, the operators comprising the query, such as selections, projections, joins, aggregates, and orderings, are performed on ciphertexts, and use modified operators in some cases.

The EncDB proxy stores a secret master key, MK, and the database schema. The DBMS server sees an anonymized schema, encrypted user data, and some auxiliary tables used by EncDB. EncDB also equips the DB server with EncDB-specific UDFs that enable the server to compute on ciphertexts for certain operations.

Processing a query in EncDB involves four steps:

1. The application issues a query, which the proxy intercepts and rewrites: it anonymizes each table and column name, and, using the master key MK, encrypts each constant in the query with an encryption scheme best suited for the desired operation (§3.1).
2. The proxy passes the query to its *onion key manager (OKM)* module, which assesses if the server should be given onion keys to execute the query. If so, the OKM does so by issuing an `UPDATE` query at the server that invokes a UDF to adjust the privacy level of the appropriate columns (§3.2).
3. The proxy forwards the encrypted query to the DBMS server, which executes it using standard SQL (occasionally invoking UDFs for aggregation).
4. The DBMS server returns the query result, which the proxy decrypts and returns to the application.

3.1 SQL-aware Encryption

The key idea to our design is that different encryption schemes enable various SQL operations. We use existing encryption schemes, optimize a recent scheme, and design a new cryptographic primitive for joins.

EncDB uses the same key (derived from MK) to encrypt each data item in a column so that the same computation can be performed on every element in that column. (We will use finer-grained encryption in §4 to reduce the potential damage of application compromise.)

For each encryption type we use, we explain the security property that EncDB requires from it, its functionality, and how to implement it with existing schemes.

Random (RND). RND provides maximum security: indistinguishability under an adaptive chosen-ciphertext attack (IND-CCA2). Two equal values will be mapped to different encryptions with high probability. RND does not allow any computation to be performed efficiently on the ciphertext. To implement RND, EncDB uses AES in UFE mode [12].

Deterministic (DET). DET has a slightly weaker guarantee, yet is still has strong security: it only leaks which encrypted values correspond to the same data value. This encryption level allows the server to perform equality checks, which means it can perform selects with equality filters, equality joins, `GROUP BY`, `COUNT`, `DISTINCT`, etc. There are many ways to implement DET, such as $\mathsf{DET}_K(v) = \mathsf{RND}_{K_1}(v) \parallel \mathsf{HMAC{-}SHA1}_{K_2}(v)$, where $\parallel$ is the concatenation operator, K_1 and K_2 are two keys derived from K, and K itself is derived by encrypting the table and column names with MK. For this DET construction, the server compares two encryptions by comparing their HMAC−SHA1 values.

Order-preserving encryption (OPE). OPE allows order relations between data items to be established based on their encrypted values, but does not leak any other information about the data. If $x < y$, then $\mathsf{OPE}_K(x) < \mathsf{OPE}_K(y)$, for any secret key K. Therefore, if a column is encrypted with OPE, the server can perform range queries when given encrypted constants $\mathsf{OPE}_K(c_1)$ and $\mathsf{OPE}_K(c_2)$ corresponding to the range $[c_1, c_2]$. The server can also perform `ORDER BY`, `MIN`, `MAX`, `SORT`, etc.

OPE is a weaker encryption scheme than DET because it reveals order. Thus, the EncDB proxy will only reveal OPE-encrypted columns to the server if users request order queries on those columns. OPE has provable security guarantees [4]: the encryption is equivalent to a random permutation that preserves order. Therefore, the difference between two encryptions $\mathsf{OPE}_K(y) - \mathsf{OPE}_K(x)$ is basically random, and reveals no information about $y - x$ except for the sign.

The scheme we use [4] is the first provably secure scheme. Until EncDB, there has not been an implementation or any measure of how practical the scheme would be. The direct implementation of the scheme took ~ 25 ms per encryption. We improved the algorithm by using AVL binary search trees when batch encryption is done (e.g., database loads), reducing the cost of OPE encryption to 7 ms per encryption without affecting its security. We also implemented a hypergeometric sampler that lies at the core of OPE, porting a Fortran implementation from 1988 [20].

Homomorphic encryption (HOM). HOM is a highly secure encryption scheme (IND-CCA secure), allowing

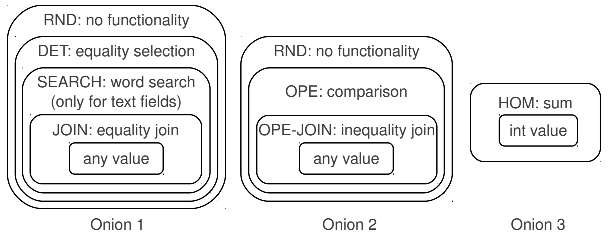

Figure 2: Onion layers of encryption and the classes of computation they allow.

the server to perform computations on encrypted data with the final result decrypted at the proxy. While fully-homomorphic encryption is prohibitively slow [10], homomorphic encryption for specific operations is efficient. To support summation, we implemented the Paillier cryptosystem [29]. With Paillier, multiplying the encryptions of two values results in an encryption of the sum of the values, i.e., $\mathsf{HOM}_K(x) \cdot \mathsf{HOM}_K(y) = \mathsf{HOM}_K(x+y)$, where multiplication is performed modulo some public-key value. To compute `SUM` aggregates, EncDB's proxy replaces `SUM` with calls to a UDF that performs Paillier multiplication on a column encrypted with HOM. HOM can also be used for computing averages by having the DBMS server return the sum and the count separately, and for incrementing values (e.g., `SET id=id+1`), on which we will elaborate shortly.

In HOM, the ciphertext is 2048 bits; however, we can pack multiple values into one ciphertext using the scheme of [16], with an amortized space overhead of only 2 times (e.g., a 32-bit value occupies 64 bits).

Word search (SEARCH). To allow word searches in a given column (e.g., using the MySQL "ILIKE" operator), we implement a cryptographic protocol for keyword searches on encrypted text [2, 37].

Join (JOIN and OPE−JOIN). A separate encryption is necessary to allow equality joins between two columns, because we use different keys for DET to prevent cross-columns correlations. JOIN also supports all operations allowed by DET and SEARCH, and also allows the server to detect repeating values between two columns. OPE−JOIN enables joins by order relations. We discuss our new join algorithm in §3.3.

EncDB uses existing transaction and indexing mechanisms in the DBMS server without any modifications. For transactions, the proxy passes along any `BEGIN`, `COMMIT`, and `ABORT` queries to the DBMS. The DBMS builds indexes of encrypted columns in the same way as it builds indexes of plaintext data. The proxy does not request indexes on RND encryptions, since no lookups are performed at that level, but the DB server can construct indexes on DET, JOIN, OPE, and OPE−JOIN encryptions, as with unencrypted data.

3.2 Adjustable Query-based Encryption

A key part of EncDB's design is *adjustable query-based encryption*, which dynamically adjusts the level of encryption on the DBMS server. The goal is provide the most secure encryption scheme that can also run each query. For example, if there is no reason to compare data items in a column, or sort a column, the column should be encrypted with RND, and for columns that perform equality checks but not inequality checks, DET suffices. Unfortunately, there are many situations where the query set is not known in advance. Thus, we need an adaptive scheme that dynamically adjusts encryption strategies.

Our idea is to encrypt each data item into an *onion*: each value in the table is dressed in layers of increasingly stronger encryption, as illustrated in Figs. 2 and 3. Each layer of each onion enables certain kinds of functionality as explained in the previous subsection. For example, the outermost layers, RND and HOM, provide maximum security, whereas OPE provides more functionality. For numeric values, EncDB maintains three onions, and for string values, EncDB maintains two onions (the DET onion, of the same length as the string, the OPE onion, of constant length, and no HOM onion).

For each level of each onion, the proxy uses the same key for encrypting values in the same column, and different keys across columns, onion levels, and tables. All these keys are derived from the master key MK. For example, for table t, column c, encryption level l, the proxy uses the key

$$K_{t,c,l} = \mathsf{PRP}_{\mathsf{MK}}(\text{table } t, \text{column } c, \text{level } l), \qquad (1)$$

where PRP is a pseudorandom permutation (e.g., AES).

Each onion starts out encrypted with the most secure encryption scheme (RND for onions 1 and 2, and HOM for onion 3). As the proxy receives SQL queries from the application, the onion key manager (OKM) determines whether layers of encryption need to be removed. Given a predicate P on column c needed for the query, the OKM first establishes what onion layer is needed to perform P on c. If the encryption of c is not already at an onion layer that allows P, the OKM strips off the onion layers to allow P on c, by sending the corresponding onion key to the server. The lowest encryption level is never stripped.

EncDB implements onion layer decryption using UDFs that run on the DBMS server. For example, in Fig. 3, to decrypt onion 2 of column 2 in table 1 to level OPE, the OKM issues the following query to the server using the `DECRYPT_RND` UDF:

```
UPDATE Table1 SET C2-Onion2 =
                  DECRYPT_RND(K, C2-Onion2)
```

where K is the appropriate key computed as in Eq. (1).

Employees

ID	Name
23	Alice

Table1

C1-Onion1	C1-Onion2	C1-Onion3	C2-Onion1	C2-Onion2
x2b82ae	xcb9e4	xc234e4	x8ab113	xd101e3

Figure 3: Data layout at the server. When the application creates a table with the schema on the left, the table created at the server is the one from the right.

Note that onion decryption is performed entirely by the DBMS server. In the steady state, *no server-side decryptions are needed*, because onion decryption happens only when a new type of computation is performed on a column. For example, after an equality check is performed on a column and the server brings the column to level DET, the column remains in that state, and future queries with equality checks require no decryption. This property ensures that the overhead of EncDB is modest in the steady state (see §7) because the server mostly performs typical SQL processing. The security level to which the database converges is the maximum privacy level for the set of queries issued by the application.

Read Query Execution. Consider an example consisting of a table `Employees`, which has four columns of interest: `id`, `name`, `address`, and `salary`. Initially, each column in the table is dressed in all onions of encryption, with RND and HOM as outermost layers, as shown in Figure 2. At this point, the server can learn nothing about the data content other than the number of columns, rows, and data size.

To execute predicates on a column, the proxy replaces the column with the name of the onion that allows the necessary operation on that field, and, for certain operations (such as `SUM`), the proxy replaces the operation with its equivalent UDF that operates on ciphertexts.

To illustrate when onion layers are removed, consider the query `SELECT * FROM Employees WHERE name = 'Alice'`, which requires lowering encryption of `name` to level DET. In this case, the proxy first issues the query `UPDATE Table1 SET C2-Onion1 = DECRYPT_RND(`$K_{1,2,\mathsf{RND}}$`, C2-Onion1)`, and then `SELECT C1-Onion1, C2-Onion1, C3-Onion1, C4-Onion1 FROM Table1 WHERE C2-Onion1 = x7d35a3`, where `x7d35a3` is an encryption of "Alice" with key $K_{1,2,\mathsf{DET}}$. The proxy decrypts the results from the server and returns them to the user.

If the next query is `SELECT COUNT(*) FROM Employees WHERE name = 'Bob'`, no server-side decryptions are necessary, and the proxy directly issues the query `SELECT COUNT(*) FROM Table1 WHERE C2-Onion1 = xbb234a`, where `xbb234a` is the encryption of "Bob".

Write Query Execution. To support `INSERT`, `DELETE`, and `UPDATE` queries, the EncDB proxy applies the same processing to the predicates (i.e., the `WHERE` clause) as for read queries. `DELETE` queries require no additional processing. For all `INSERT` queries and for `UPDATE` queries that set the value of a column to a constant, the proxy encrypts each inserted column's value with each onion layer that has not been stripped off yet in that column

The remaining case is an `UPDATE` that sets a column value based on another column value, such as `salary=salary+1`. Such an update would have to be performed using HOM, to handle additions. However, in doing so, the values in the OPE and DET onions would become stale. In fact, an encryption scheme that allows both addition and comparison at the same time is fundamentally insecure: if a malicious server knows the order of the items (OPE) and can increment the value by one, the server can keep adding one to each field homomorphically until the field becomes equal to some other value. This would allow the server to compute the difference between any two values in the database, which is almost equivalent to knowing their values.

There are two solutions to this problem. If a column is incremented and then only projected (no comparisons performed on it), the solution is simple: when requesting the value of this field, use the value of Onion 3 rather than Onion 1 or 2, because Onion 3 is up-to-date. This is the case for increment TPC-C queries. If a column is used in comparisons after it is incremented, the solution is to split the query in a select of the old values to be updated, and then an update with the new values. This strategy usually works in practice (such as in TPC-C and our applications) because updates are executed on individual or on few rows.

3.3 Computing Joins

There are two kinds of joins: *equi-joins* in which the join predicate is based on equality, and *range joins*, which involve order checks. Supporting these joins is a challenging problem. If two columns are to be joined, they need to be encrypted with the same key.

To provide maximum privacy for equi-joins, the DBMS server should not be able to join columns for which the user did not request a join, so columns that are never joined should not be encrypted with the same keys. Moreover, if users request a join of columns A and B, and a join of columns C and D, the DBMS server should not be able to join B and C. Thus, the question is, which keys should each column be encrypted with, given that we do not know in advance what columns will be joined?

To address this problem, we introduce a new cryptographic primitive that allows the DBMS server to dynamically adjust the JOIN encryption keys of each column. Each column is initially encrypted with a different JOIN key, preventing all joins. When a query requests a join, the proxy will give the DBMS server an onion key to re-encrypt the two columns to the same JOIN key, allowing joins between the two columns.

Our algorithm is based on elliptic-curve cryptography (ECC). When a row is inserted, the JOIN encryption of value v is computed as $\mathsf{JOIN}_K(v) := H(v)^K$, where K is the initial key for that table, column, and level, and H is a mapping from values (integers or strings) to a point on an elliptic curve. When the query joins columns c and c', the proxy computes $\Delta K = K/K'$, which can be used to bring the JOIN encryptions of c and c' to the same key. Given $\mathsf{JOIN}_{K'}(v)$ (stored in column c') and ΔK, the DBMS server uses a UDF to compute $\mathsf{JOIN}_{K'}(v)^{\Delta K} = H(v)^{K' \times K/K'} = H(v)^K = \mathsf{JOIN}_K(v)$. Now columns c and c' share the same JOIN key, and the DBMS server can perform an equi-join on c and c' as usual.

We proved the security of this scheme cryptographically (elided here for space, but will be in an extended version) using the hardness assumption of computing the discrete logarithm in elliptic curve-based groups. The server can only join pairs of columns that were joined by a legitimate query and the encryption scheme does not reveal data.

For range joins, a similar dynamic re-encryption scheme is difficult to construct due to lack of structure of OPE schemes. Instead, EncDB requires that pairs of columns that will be involved in such joins be declared by the application ahead of time, so that matching keys are used for level OPE−JOIN of those columns. Fortunately, range joins are rare (and not used in any of our example applications).

3.4 Optimizations

EncDB implements four performance optimizations.

Insensitive fields. By default, EncDB encrypts all fields. The programmer can annotate just the sensitive fields in the schema (§4) to avoid encrypting public fields.

Known query set. If an application has a known query set or a set of queries likely to happen, as is the case for many web applications, EncDB has a training module that can adjust onions at this level before system startup, thus reducing onion adjustment work from runtime. For a fully known query set, EncDB can also discard any onions that will not be needed (e.g., discard OPE onion for a column if range queries are not performed on that column).

Security convergence. Even if the query set is not known in advance, after an application has run for a considerable time, the proxy may drop any onions that have not been used, because these onions are unlikely to be used in the future. If a query does happen to use these onions later, the proxy can go through the cost of adding back the deleted column, but such event should be rare.

Ciphertext caching. A significant ongoing cost for the proxy lies in generating OPE and HOM encryptions of constants used in queries. To avoid this cost, the proxy maintains a cache of commonly-used constants, along with their encryptions under different keys. Since some constants are repeatedly used in many queries (e.g., the constant 1), this optimization reduces the amount of CPU time spent by the proxy encrypting data.

3.5 Discussion

EncDB's design supports most relational queries and aggregates on standard data types, such as integers and text/varchar types. Additional operations can be added to EncDB by extending its existing onions, or adding new onions for specific data types (e.g., spatial and multi-dimensional range queries [34]).

There are certain computations EncDB cannot support on encrypted data. For example, it does not support both computation and comparison on the same column, such as `WHERE salary > age*2+10`. EncDB can process a part of this query, but it would also require some processing on the proxy. In EncDB, such a query should be (1) rewritten into a sub-query that selects a whole column, `SELECT age*2+10 FROM ...`, which EncDB computes using HOM, and (2) re-encrypted in the proxy, creating a new column (call it `aux`) on the DBMS server consisting of the newly-encrypted values. Finally, the original query with the predicate `WHERE salary > aux` should be run. We have not been affected by this limitation in our test applications (TPC-C, phpBB, HotCRP, and grad-apply).

4 MULTIPLE USERS

We now extend the threat model to the case when the application infrastructure and proxy are untrusted. This model is especially relevant for a multi-user web site running a web and application server. To explain the problems encountered by a multi-user web application and EncDB's solution to these problems, consider phpBB, a popular online web forum. Each user has an account and a password, belongs to certain groups, and can send private messages to other users. Depending on their groups' permissions, users can read entire forums, only forum names, or not be able to read a forum at all.

There are several confidentiality guarantees that would be useful in phpBB. For example, ensuring that a private message sent from one user to another is not visible to anyone else; ensuring that posts in a forum are only accessible to users in a group with access to that forum; and ensuring that the name of a forum is only shown to users belonging to a group that's allowed to view it. EncDB provides these capabilities in the face of arbitrary compromises, limiting the damage caused by a compromise.

There are two main problems. First, EncDB must make it easy to express policies like the ones above. Rather than require intrusive additions to the code, we allow the developer to annotate the database schema (§4.1). Second, EncDB must encrypt the data with several different keys; otherwise, an attacker that compromises the proxy or application server can decrypt all data. However, simply encrypting each data item with a single user's key does not work: in our example, forum posts must be shared between users, and the set of users with access to a forum may change at runtime.

Our solution limits the leakage resulting from a compromised application or proxy to only the data accessible by users who were logged in during the compromise. The amount of data leaked in our solution seems unavoidable given our assumptions about the impracticality of fully-homomorphic encryption: the relevant data must be decrypted whenever the web application performs arbitrary computations to service the requests of active users.

EncDB encrypts different data items with different keys, according to the specified policies. EncDB enforces the policies, which involve multiple users being able to read/write any given data item, at run-time by encrypting and decrypting the required keys, starting from the password of the user and ending at the desired keys, following a chain of keys (§4.2).

4.1 Policy Annotations

To express the privacy policy of a database-backed application at the level of SQL queries, the application developer can annotate the schema of a database in EncDB by specifying, for each subset of data items, which *principal* has access to them. A principal is an entity, such as a user or a group, over which it is natural to specify an access policy. Each SQL query involving an annotated data item requires the privilege of the corresponding principal.

An application developer annotates the schema using the steps described below and shown in Figure 4.

Step 1. The developer must define the *principal types* (`PRINC TYPES`) used in her application, such as users, groups, or messages. A *principal* is an instance of a principal type, e.g., principal 5 of type user. Principals are of two types: external and internal. External principals correspond to end users who explicitly authenticate themselves to the web site using a password. When a user logs into the application, the application must provide the user password to the EncDB proxy so that the user can gets the privileges of his external principal. Privileges of other (internal) principals can only be acquired through delegation, as described in the Step 3. The application also informs the proxy when the user logs out.

Step 2. The developer must specify which columns in her SQL schema contain sensitive data, along with the principals that should have access to that data, using the `ENCRYPT_FOR` annotation. EncDB requires that for each private data item in a row, the name of the principal that should have access to that data be stored in another column in the same row. For example, in Figure 4, the decryption of *msgtext* `x37a21f` is only available to principal 5 of type *msg*.

```
PRINC TYPES physical_user EXTERNAL;
PRINC TYPES user, msg;

CREATE TABLE privmsgs (
  msgid   int,
  subject  varchar(255) ENCRYPT_FOR PRINC msgid TYPE msg,
  msgtext text             ENCRYPT_FOR PRINC msgid TYPE msg );

CREATE TABLE privmsgs_to (
  msgid int,   rcpt_id int,   sender_id int,
  PRINC sender_id TYPE user
      HAS_ACCESS_TO PRINC msgid TYPE msg,
  PRINC rcpt_id TYPE user
      HAS_ACCESS_TO PRINC msgid TYPE msg);

CREATE TABLE users (
  userid int,   username varchar(255),
  PRINC username TYPE physical_user HAS_ACCESS_TO
      PRINC userid TYPE user);
```

Example table contents, without anonymized column names:

msgid	subject	msgtext
5	xcc82fa	x37a21f

msgid	rcpt_id	sender_id
5	1	2

userid	username
1	'Alice'
2	'Bob'

Figure 4: Part of phpBB's schema with annotations to secure private messages. Only the sender and receiver may see the private message. An attacker that gains complete access to phpBB and the DBMS can access private messages of only currently active users. Bold text indicates annotations added for EncDB.

Step 3. Programmers can specify rules for how to delegate the privileges of one principal to other principals. For example, in phpBB, a user should also have the privileges of the groups he belongs to. Since many applications store such information in tables, programmers can tell EncDB to infer delegations from rows in an existing table. In EncDB, programmers can annotate a table T with `PRINC` a `TYPE` x `HAS_ACCESS_TO PRINC` b `TYPE` y. This annotation indicates that each row present in that table grants principal a of type x access to everything that principal b of type y can access. Here, x and y must always be fixed principal types. Principal b is always specified by the name of a column in table T. On the other hand, a can be either the name of another column in the same table, a constant, or $T2.col$, meaning *all* principals from column *col* of table $T2$. For example, in Figure 4, principal "Bob" of type *physical_user* has access to principal 1 of type *user*, and in Figure 6, all principals in the *contactId* column from table *PCMember* (of type *contact*) have access to the *paperId* principal of type *review*. Optionally, the programmer can specify a predicate, whose inputs are values in the same row, to specify a condition under which delegation should occur, such as excluding conflicts in Figure 6. §5 provides more examples of using annotations to secure real applications.

To aid debugging, we allow the developer to provide a SQL trace of a previous run of the application that also includes when users logged in or out, and the proxy can verify if previous queries are permitted to run given the current annotations, and flag whether the developer should revise the annotations.

4.2 Key Chaining

Each principal (i.e., each instance of each principal type) is associated with a secret, randomly-chosen key. If principal B has access to principal A, then principal A's key is encrypted using principal B's key, and stored in a special access_keys table in the database. In this way, principal B has access to principal A's key. For example, in Figure 4, to give users 1 and 2 access to message 5, the key of *msg* 5 is encrypted with the key of *user* 1, and also separately encrypted with the key of *user* 2; both encryptions are stored in access_keys. Each sensitive field is encrypted with the key of the principal in the ENCRYPT_FOR annotation. EncDB encrypts the sensitive field with onions in the same way as for single-principal EncDB, except that onion keys are derived from a principal's key as opposed to a global master key.

The key of each principal is a combination of a symmetric key and a public–private key pair. In the common case, EncDB uses the symmetric key of a principal to encrypt any data and other principals' keys accessible to this principal, with little CPU cost. However, this is not always possible, if some principal is not currently online. For example, in Figure 4, suppose Bob sends message 5 to Alice, but Alice (user 1) is not online. This means that EncDB does not have access to user 1's key, so it will not be able to encrypt message 5's key with user 1's symmetric key. In this case, EncDB looks up the public key of the principal (i.e., user 1) in a second table, public_keys, and encrypts message 5's key using user 1's public key. When user 1 logs in, they will be able to use the secret key part of their key to decrypt the key for message 5.

For external principals (e.g., physical users), EncDB assigns a random key just as for any other principal. To give an external user access to the corresponding key on login, EncDB stores the key of each external principal in a third table, external_keys, encrypted with the principal's password. This allows EncDB to obtain a user's key given the user's password, and also allows a user to change his or her password without changing the key of the principal.

In EncDB, one user can grant a second user access to principal P only if the first user already has access to P himself. At the cryptographic level, this means that the first user must have P's key in order to re-encrypt it for the second user. This closely follows real-world semantics, since if someone can grant others access to P, they can also grant themselves access to P.

When encrypting data in a query or decrypting data from a result, EncDB follows key chains starting from passwords of users logged in until it obtains the desired keys. As an optimization, when a user logs in, EncDB's proxy loads the keys of some principals to which the user has access (in particular, those principal types that do not have too many principal instances—e.g., for groups the user is in, but not for messages the user received—which EncDB determines by keeping aggregate statistics).

Applications inform EncDB of a user logging in or out by simply issuing SQL `INSERT` and `DELETE` queries to a special table encdb_active, which the proxy recognizes.

EncDB guards the data of inactive users at the time of an attack. However, some special users such as administrators with access to a large pool of data enable a larger compromise upon an attack. To avoid attacks happening when the administrator is logged in, the administrator should create a separate user account with restricted permissions when accessing the application as a regular user. Also, as best practice, an application should automatically log off users who have been inactive for a while.

5 Application Case Studies

In this section, we explain how EncDB can be used to secure three existing multi-user web applications. In our examples, annotations are shown in bold. For brevity, we show simplified schemas, ignoring irrelevant fields and type specifiers. Overall, we find that once the programmer specifies the principals in their schema, and the delegation rules for them using HAS_ACCESS_TO, protecting additional sensitive fields is quite easy using ENCRYPT_FOR.

phpBB is a widely-used open source forum with a rich set of access control settings. Users are organized in groups; both users and groups have a variety of access permissions that the application administrator can choose. We already showed how to secure private messages between two users in phpBB in Fig. 4. A more interesting case is securing access to posts, as shown in Fig. 5. This example shows how to use predicates (e.g. PRED_POST) to give conditional access to principals, and also how one column (`forumid`) can be used to represent multiple principals (of different type) with different privileges. There are more ways to access to a post, but we do not show them all here for brevity.

HotCRP is a popular conference review application. A key policy for HotCRP is that PC members cannot see who reviewed their own (or conflicted) papers. Fig. 6 shows EncDB annotations for HotCRP's schema to enforce this policy. Today, HotCRP cannot prevent a curious or careless PC chair from logging into the database server and seeing who wrote each review for a paper that he is in conflict with. As a result, many conferences set up a second server to review the chair's papers or use inconvenient out-of-band emails. With EncDB, a PC chair cannot learn who wrote each review for his paper, even if he breaks into the application or database, since he does not

```
PRINC TYPES physical_user EXTERNAL;
PRINC TYPES user, group, forum_post, forum_name;

CREATE TABLE users ( userid int, username varchar(255),
  PRINC username TYPE physical_user HAS_ACCESS_TO
       PRINC userid TYPE user);

CREATE TABLE usergroup ( userid int, groupid int,
  PRINC userid TYPE user HAS_ACCESS_TO
       PRINC groupid TYPE group);

CREATE TABLE aclgroups ( groupid int, forumid int, optionid int,
  PRINC groupid TYPE group HAS_ACCESS_TO
       PRINC forumid TYPE forum_post IF optionid=20,
  PRINC groupid TYPE group HAS_ACCESS_TO
       PRINC forumid TYPE forum_name IF optionid=14);

CREATE TABLE posts ( postid int, forumid int,
  post text ENCRYPT_FOR PRINC forumid TYPE forum_post);

CREATE TABLE forum ( forumid int,
  name varchar(255) ENCRYPT_FOR PRINC forumid
       TYPE forum_name);
```

Figure 5: Annotated schema for securing access to posts in phpBB. A user has access to see the content of posts in a forum if any of the groups the user is part of, has such permissions, which is indicated by optionid 20 in an aclgroups table for the corresponding forumid and groupid. Similarly, optionid 14 enables users to see the forum's name.

have the decryption key.[1] (We assume the PC chair is not malicious, but might be careless, so he won't modify the application to log the passwords of PC members to subvert the system.)

grad-apply is a graduate admissions system used at a major university. We annotated its schema to allow an applicant's folder to be accessed only by the respective applicant and any faculty using: `PRINC reviewers.reviewer_id TYPE reviewer` (meaning all reviewers) `HAS_ACCESS_TO candidate_id` in table candidates, and `HAS_ACCESS_TO letter_id` in table letters. The applicant can see all his folder except for letters of recommendation. Overall, grad-apply has simple access control and therefore simple annotations.

6 IMPLEMENTATION

The EncDB proxy consists of a C++ and a Lua module. The C++ module consists of a query parser; a query encryptor and rewriter, which encrypts fields or calls UDFs; and a result decryption module. To allow applications to transparently use EncDB, we used MySQL Proxy and implemented a Lua module that passes queries and results through our C++ module. We implemented our new cryptographic protocols using NTL [35]. EncDB is ~ 8500 non-empty lines of code.

[1]Fully implementing this policy would require setting up two PC chairs: a main chair, and a backup chair responsible for reviews of the main chair's papers. HotCRP allows the PC chair to impersonate other PC members, so EncDB annotations would be used to prevent the main chair from gaining access to keys of reviewers assigned to his paper.

```
PRINC TYPES physical_contact EXTERNAL;
PRINC TYPES contact, review;

CREATE TABLE ContactInfo ( contactId int, email varchar(120),
  PRINC email TYPE physical_contact HAS_ACCESS_TO
       PRINC contactId TYPE contact);

CREATE TABLE PCMember ( contactId int );
CREATE TABLE PaperConflict ( paperId int, contactId int );

NoConflict (paperId, contactId):           /* Define a SQL function */
  (SELECT count(*) FROM PaperConflict c WHERE
       c.paperId = paperId AND c.contactId = contactId) = 0;

CREATE TABLE PaperReview ( paperId int,
  reviewerId       int  ENCRYPT_FOR PRINC paperId TYPE review,
  commentsToPC text ENCRYPT_FOR PRINC paperId TYPE review,
  PRINC PCMember.contactId TYPE contact
       HAS_ACCESS_TO PRINC paperId TYPE review IF
                      NoConflict(paperId, contactId));
```

Figure 6: Annotated schema for securing reviews in HotCRP. Reviews and the identity of reviewers providing the review will only be available to PC members (table PCMember includes PC Chairs) who are not conflicted, and PC Chairs cannot override this restriction.

EncDB is portable and supports both Postgres 9.0 and MySQL 5.1; initially implemented in Postgres, porting EncDB to MySQL required changing only 86 lines of code, mostly in the code for connecting to the MySQL server and declaring UDFs. As mentioned earlier, EncDB does not change the DBMS; we implement all server-side functionality with UDFs and server-side tables. This modular change was possible because the DBMS lies in between two aspects of query processing that EncDB must modify: query planning and execution is between query parsing and low-level operations on data items. EncDB should run on top of any SQL DBMS that supports UDFs.

7 EXPERIMENTAL EVALUATION

This section evaluates the security EncDB provides to applications, developer effort to annotate the schema, as well as performance overheads. Our multi-user application evaluations are done with phpBB, HotCRP, and grad-apply (§5). To study the performance of our encrypted DBMS, we also measure the query and transaction throughput for the industry-standard TPC-C benchmark and its SQL operators.

7.1 Application Security and Annotations

The first observation from porting TPC-C, phpBB, HotCRP, and grad-apply, is that EncDB supports, on encrypted data, all queries from TPC-C and all queries on sensitive fields from our three applications. Equally important, our annotations were able to secure sensitive fields in the three applications without limiting these applications' functionality.

Figure 7 shows the effectiveness of annotations and fields secured for our applications. Overall, EncDB requires few unique annotations, and minimal changes to

Application	Annotations	Login/logout code	Sensitive fields secured, and examples of such fields
phpBB	32 (11 unique)	7 lines	24: private messages (content, subject), posts, forums
HotCRP	29 (12 unique)	2 lines	22: paper content and paper information, reviews
grad-apply	106 (13 unique)	2 lines	98: student grades (61), scores (17), recommendations, reviews
TPC-C (single princ.)	0	0	92: all the fields in all the tables encrypted

Figure 7: Effectiveness of annotations and fields secured. This table shows, for a few different applications, the number of annotations the programmer needs to add to secure sensitive fields, lines of code to be added to provide EncDB with the password of users, and the number of sensitive fields that EncDB secures with these annotations. We count as one annotation any invocation of our three types of annotations and any line of any SQL predicate used in a HAS_ACCESS_TO annotation. Since multiple fields in the same table are usually encrypted for the same principal (e.g., message subject and content), we also report unique annotations.

Application	RND	DET	OPE	**Total**	HOM
PhpBB	19	2	3*	24	2
HotCRP	19	1	2*	22	2
grad-apply	89	7	2*	98	0
TPC-C	65	19	8*	92	8

Table 1: Steady-state security of onion layers. Each number indicates how many encrypted fields remain at that onion level after the applications issued all query types. DET includes JOIN values because they have similar security level (AES). Each field is counted for the onion layer with lowest security. We also report how many fields in addition used the HOM onion. (*) All the fields at OPE were only slightly sensitive (e.g., post_time, number of forum posts).

the application code. It felt surprisingly simple how one simple annotation would secure a column in a database. For TPC-C, no annotations or application changes were required, because TPC-C ran in single-principal mode (corresponding to threat 1 from §2.1): the entire database was encrypted with the key of one principal single key.

To quantify the security level provided by EncDB to these applications, Table 1 shows the encryption schemes that are exposed to the server for each application in steady-state under a typical workload. The results show that most fields (71–91%) remain encrypted with RND, the most secure scheme. Looking at the most sensitive fields in each application, they were all at RND. DET, which leaks only information about repeating values, is used in 5%–21% of the fields. OPE, which leaks order, is used a little less frequently (9–13%), and only for fields that are marginally sensitive (e.g., timestamps). Thus, EncDB's adjustable security provides a significant improvement in confidentiality over revealing all encryption schemes to the server.

Finally, we empirically validated EncDB's confidentiality guarantees by trying real attacks on phpBB that have been listed in the CVE database [27], including two SQL injection attacks (CVE-2009-3052 & CVE-2008-6314), bugs in permission checks (CVE-2010-1627 & CVE-2008-7143), and a bug in remote PHP file inclusion (CVE-2008-6377). We found that, for users not currently logged in, the answers returned from the DBMS were encrypted; even with root access to the application server, proxy, and DBMS, the answers were not decryptable.

7.2 Performance Results

To evaluate the performance of EncDB, we use a server with an Intel Xeon 4-core 3.20 GHz CPU and 3 GB of RAM, and a machine with an Intel Xeon 4-core 1.6 GHz E7310 CPU and 8 GB RAM to simulate multiple users.

7.2.1 TPC-C

We compare the performance of TPC-C running on Postgres with its performance when using a EncDB proxy in front of a Postgres server. Figures 8 and 9 shows the query throughput on TPC-C when transactions are disabled and enabled, respectively. EncDB incurs a 21% overhead without transactions, and the overhead with transactions increases to 27% because of increased lock contention caused by longer processing times and expanded queries.

To understand the sources of EncDB's overhead, we examine the throughput of individual SQL operators; this analysis is useful because the mix of operators changes with application. For each operator, we collect the corresponding queries from TPC-C, and measure the latency for those queries running under EncDB and under Postgres. Table 3 shows the results. The proxy encryption time adds an average of 0.34 ms to the query. The ciphertext caching optimization masks the high latency of queries requiring OPE and HOM, as indicated by bold numbers and their corresponding Proxy$^-$ values. The DBMS server latencies with and without EncDB are similar, suggesting that the expansion of data at the server caused by encryption has a small impact. End-to-end, EncDB with the caching optimization adds 0.64 ms of latency to each query.

Figure 10 shows the throughput for the same mix of queries for Postgres, EncDB, and a strawman design that performs each query by first decrypting the data using a UDF, performing the query, and re-encrypting the result (if updating the row). In all cases except for insert, the strawman performs significantly worse than EncDB, since the DBMS cannot use an index to satisfy WHERE clauses. It is indeed an unexpected fact that the higher security of EncDB over the strawman in fact also brings better performance. For six SQL operators (Select equality, Select join, Select range, Delete, Insert, and Update set), the throughput overhead of EncDB is negligible compared to Postgres. These six constitute most of the queries

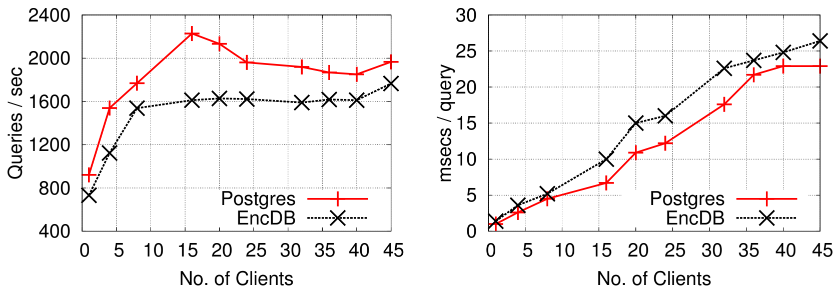

Figure 8: Throughput and latency for TPC-C queries without transactions.

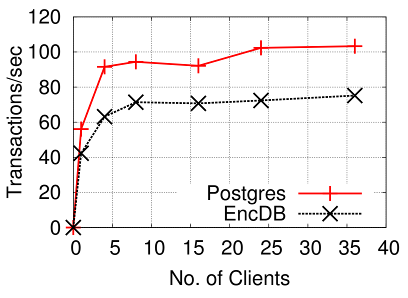

Figure 9: Throughput of TPC-C transactions.

DB	Login	R post	W post	R msg	W msg
MySQL	105 ms	85 ms	238 ms	107 ms	389 ms
EncDB	183 ms	133 ms	319 ms	166 ms	481 ms

Table 2: Latency for HTTP requests that heavily use encrypted fields in phpBB for MySQL and EncDB. R and W stand for read and write.

Query (& scheme)	EncDB			Postgres
	Server	Proxy	Proxy^{-}	Server
Select by = (DET)	0.43 ms	0.10 ms	0.10 ms	0.41 ms
Select join (JOIN)	0.72 ms	0.27 ms	0.27 ms	0.63 ms
Select range (OPE)	1.2 ms	**0.40 ms**	58.2 ms	0.99 ms
Select sum (HOM)	8.8 ms	0.18 ms	0.18 ms	0.46 ms
Delete	1.1 ms	0.15 ms	0.14 ms	1.1 ms
Insert	1.0 ms	**0.34 ms**	18.6 ms	0.99 ms
Update set	1.2 ms	0.17 ms	0.17 ms	1.1 ms
Update incr (HOM)	2.0 ms	**0.71 ms**	17.7 ms	1.8 ms
Overall	1.4 ms	**0.34 ms**	7.3 ms	1.1 ms

Table 3: Latency for SQL operators. Proxy is the encryption latency for EncDB; "Proxy^{-}" is the encryption latency without using encryption tables (§3.4). Bold numbers show where the caching optimization helps. For each operator, we show the predominant encryption scheme used. The "Overall" row is the average latency over the mix of TPC-C queries. "Update set" is an update where the fields are set to a constant, and "Update incr" is an update where the fields are incremented.

for TPC-C, and likely for many other applications. Homomorphic operations, such as Select sum and Update increment, incur a significant overhead with EncDB, due to the server-side cost of homomorphically multiplying large cryptographic numbers instead of adding 32-bit integers.

Adjustable query-based encryption involves decrypting columns to lower onion levels. Fortunately, such decryption is fast, and only needs to be performed once per column for the lifetime of the system.[2] Removing a layer of RND requires AES decryption, which a commodity machine can perform at $\sim$ 500 MBytes/s. Thus, removing an onion layer is bottlenecked by the speed at which the DBMS server can copy a column from disk.

7.2.2 *Multi-User Web Applications*

To evaluate the impact of EncDB on application performance, we measure the throughput of phpBB for a workload of 30 parallel clients continuously issuing HTTP requests to browse the forum, write and read posts, write and read private messages, etc. We pre-populate forums and user mailboxes with initial messages. Figure 11 shows the throughput of phpBB in four configurations, running on a single server with MySQL as the DBMS: (1) MySQL, (2) MySQL with the proxy performing only query parsing, (3) EncDB with half of phpBB's sensitive fields encrypted, and (4) EncDB with all sensitive fields encrypted. The results show that phpBB incurs a user-visible throughput loss of 13%. 80% of this loss is from parsing SQL queries in EncDB's proxy; an optimized parser would reduce this overhead. Finally, the difference between encrypting all the fields versus half of them is small, indicating that cryptographic operations are not a problematic bottleneck.

Figure 2 shows the end-to-end latency for various phpBB requests. We see that login is slowed down by

[2]One exception is if the administrator wants to periodically re-encrypt to increase the security level.

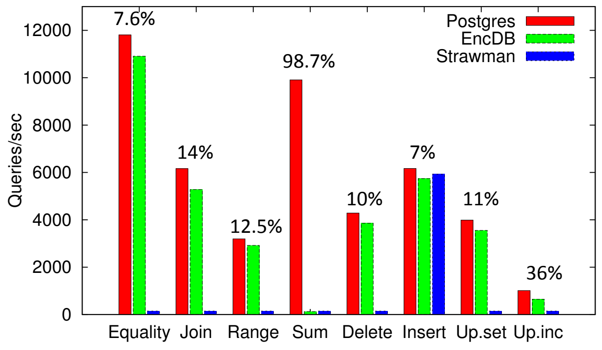

Figure 10: Throughput of the SQL operators from Table 3 running under EncDB and Postgres. The numbers on top of the bars show the throughput reduction with EncDB for that operator. "Up. inc" stands for update with increment and "up. set" for update set.

78 ms because EncDB loads keys from the DBMS on login. Most of the other latency increases are due to EncDB's proxy. Fortunately, the total increase is small.

7.2.3 *Storage*

The storage overhead of EncDB can come from two parts: the proxy and the DBMS. The memory footprint of the EncDB proxy process is 18.5 MBytes in our experiments. Caching the ciphertext of the 100000 most common values consumes $<$ 1 MByte for OPE and $\sim$ 12 MBytes for HOM. On the DBMS server, for TPC-C, EncDB increased the database size by 4.5$\times$ due to cryptographic expansion of certain integer fields, all fields being encrypted

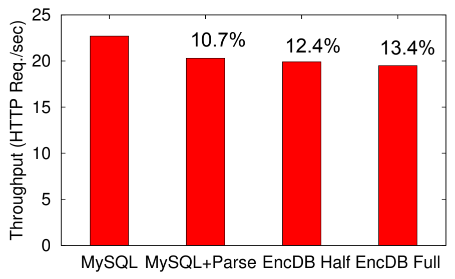

Figure 11: Throughput comparison for phpBB. "MySQL" denotes phpBB running directly on MySQL, "MySQL+Parse" includes the parsing cost of EncDB with cryptography and key management removed, "EncDB Half" shows the whole EncDB with half the annotations, and "EncDB Full" is the end-to-end system with all sensitive fields annotated. Most HTTP requests involved tens of SQL queries each. Percentages indicate throughput reduction relative to MySQL.

and most of them being integers. In fact, strings and binary data remain roughly the same size. Consequently, for phpBB, EncDB increases server storage by 42%, caused largely by the public keys of principals (users, groups and messages) and the HOM onion for encrypted integer fields. As the number of posts in a forum grows, the amortized storage overhead decreases because no new principals are added.

8 Related Work

Search and queries over encrypted data. Song et al. [37] and Amanatidis et al. [2] describe cryptographic tools for searching keywords over encrypted data; we use a hybrid of these schemes in EncDB. Bao et al. [3] extend these encrypted search methods to the multi-user case. Yang et al. run selections with equality predicates over encrypted data [40]. Evdokimov and Guenther present methods for the same selections, as well as Cartesian products and projections [14]. Agrawal et al. develop a statistical encoding that preserves the order of numerical data in a column [1], but it does not have sound cryptographic properties, unlike the scheme we use [4]. Boneh and Waters show public-key schemes for comparisons, subset checks, and conjunctions of such queries over encrypted data [5], but these are highly impractical.

These techniques are undoubtedly important contributions, but even with *all* of these previous techniques, efficient SQL query processing is unachievable. Joins are not supported, order checks are inefficient, and client-side query processing is required; moreover, implementing many of these techniques would entail unattractive changes to the innards of the DBMS, and would require users either to build and maintain indexes on the data at the server or to perform sequential scans for every selection/search. Moreover, none of these schemes was developed in the context of a prototype system.

Some researchers have developed prototype systems for subsets of SQL, but they provide no confidentiality guarantees, require a significant DBMS rewrite, and rely on client-side processing [9, 11, 18]; one proposal requires various trusted entities and two non-colluding untrusted DBMSs [8]. Hacigumus et al. split the domain of possible values for each column into partitions, storing encrypted data [18]. Each partition has as label a number and each value in a column is replaced with the number of the partition. However, confidentiality is often compromised because an adversary can now learn which elements are close together in value and can see the order of the partitions. In addition, clients have to filter query results. In contrast, our dynamically-adjustable encryption mechanism only reveals such relations for those columns that are used in a filter.

Untrusted servers. SUNDR [23] uses cryptography to provide privacy and integrity in a file system on top of an untrusted file server. Using a SUNDR-like system, SPORC [15] and Depot [25] show how to build low-latency applications, running mostly on the clients, without having to trust a server. However, existing server-side applications that involve separate database/storage and application servers cannot be used with these systems unless they are rewritten into distributed client-side applications to work with SPORC or Depot; moreover, many applications are not amenable to such a structure. In contrast, EncDB provides cryptographic privacy guarantees to existing database-backed applications.

Software security. Many tools help programmers either find or mitigate mistakes in their code that may lead to vulnerabilities, including static analysis tools like PQL [24, 26] and UrFlow [7], and runtime tools like Resin [41]. In contrast, EncDB provides privacy guarantees for user data *even if the adversary gains complete control over the application and database servers*. These tools provide no guarantees in the face of this threat, but in contrast, EncDB cannot provide privacy in the face of vulnerabilities that trick the user's client machine into issuing unwanted requests (such as cross-site scripting or cross-site request forgery vulnerabilities in web applications). As a result, we anticipate using EncDB together with these tools to improve overall application security.

Rizvi et al. [32] and Chlipala [7] specify (and enforce) an application's security policy over SQL views. EncDB's SQL annotations can capture most of these policies, except for result processing being done in the policy's view, such as allowing a user to view only aggregates of certain data. Unlike prior systems, EncDB enforces SQL-level policies cryptographically, without relying on compile-time or run-time permission checks.

Privacy-preserving aggregates. Privacy-preserving data integration, mining, and aggregation schemes (e.g. [22, 39]) are useful, but not usable by many applications because they only support specialized query types and require a rewrite of the DBMS. Differential privacy [13] is complementary to EncDB; it allows a trusted server to decide what answers to release and how to obfuscate answers to aggregation queries to avoid leaking information about any specific record in the database.

Query integrity. Techniques to ensure integrity for SQL queries can be integrated into EncDB because EncDB allows relational queries on encrypted data to be processed just like on plaintext. These methods can provide integrity by adding a MAC to each tuple [21, 23, 33], freshness using Merkle or chained hashes [21, 30, 33], and both freshness and completeness of query results using the approach in [28]. In addition, the client can verify the results of aggregation queries [38], and provide query assurance for most read queries [36].

9 Conclusion

We presented EncDB, a system that provides practical and provable confidentiality in the face of two significant threats—curious DBAs and arbitrary compromises of the application server and DBMS—confronting database-backed applications. EncDB meets its goals using three ideas: running queries efficiently over encrypted data using a novel SQL-aware encryption strategy, dynamically adjusting the encryption level using onions of encryption to minimize the information revealed to the untrusted DBMS server, and chaining encryption keys to user passwords in a way that allows only authorized users to gain access to the same encrypted data. The developer effort required to express confidentiality policies for multi-user applications is small, ranging between 11 and 13 unique annotations of the application's database schema across three existing applications (phpBB, HotCRP, and grad-apply). The throughput penalty of EncDB is modest, about 27% for the TPC-C benchmark and 13% for our synthetic phpBB workload. Our results suggest that EncDB is useful for applications where the ability to run confidentially over an untrusted or outsourced infrastructure trumps achieving the highest possible performance.

References

[1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order Preserving Encryption for Numeric Data. In *SIGMOD*, 2004.

[2] G. Amanatidis, A. Boldyreva, and A. O'Neill. Provably-secure schemes for basic query support in outsourced databases. In *IFIP WG 11.3 Working Conf. on Database and Applications Sec.*, 2007.

[3] F. Bao, R. H. Deng, X. Ding, and Y. Yang. Private query on encrypted data in multi-user settings. In *4th International Conference on Information Security Practice and Experience*, 2008.

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

[5] D. Boneh and B. Waters. Conjunctive, Subset, and Range Queries on Encrypted Data. In *Proc. of the 4th Conf. on Theory of Cryptography*, 2007.

[6] A. Chen. GCreep: Google engineer stalked teens, spied on chats. *Gawker*, Sep 2010. `http://gawker.com/5637234/`.

[7] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *OSDI*, 2010.

[8] S. S. M. Chow, J.-H. Lee, and L. Subramanian. Two-party computation model for privacy-preserving queries over distributed databases. In *Proc. of the 16th NDSS*, 2009.

[9] V. Ciriani, S. D. C. di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Keep a few: Outsourcing data while maintaining confidentiality. In *14th European Symposium on Research in Comp. Sec.*, 2009.

[10] M. Cooney. IBM touts encryption innovation; new technology performs calculations on encrypted data without decrypting it. *Computer World*, 2009.

[11] E. Damiani, S. D. C. di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational DBMSs. In *ACM CCS*, 2003.

[12] A. Desai. New paradigms for constructing symmetric encryption schemes secure against chosen-ciphertext attack. In *CRYPTO*, 2000.

[13] C. Dwork. Differential privacy: a survey of results. In *International Conf. on Theory and Applications of Models of Computation*, 2008.

[14] S. Evdokimov and O. Guenther. Encryption techniques for secure database outsourcing. Cryptology ePrint Archive, Report 2007/335.

[15] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *OSDI*, 2010.

[16] T. Ge and S. Zdonik. Answering aggregation queries in a secure system model. In *Proc. of the 33rd VLDB*, Vienna, Austria, Sep 2007.

[17] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proc. of the 41st STOC*, pages 169–178, May–Jun 2009.

[18] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*, 2002.

[19] J. A. Halderman et al. Lest we remember: Cold boot attacks on encryption keys. In *Usenix Security*, 2008.

[20] V. Kachitvichyanukul and B. W. Schmeiser. Algorithm 668: H2pec: sampling from the hypergeometric distribution. *ACM Trans. Math. Softw.*, 14(4):397–398, 1988.

[21] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proc. of the 2nd USENIX FAST*, pages 29–42, Berkeley, CA, USA, 2003.

[22] M. Kantarcioglu and C. Clifton. Security issues in querying encrypted data. In *IFIP WG 11.3 Working Conf. on Database and Applications Sec.*, 2005.

[23] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. of the 6th OSDI*, San Francisco, CA, Dec 2004.

[24] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Usenix Security*, pages 271–286, 2005.

[25] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *OSDI*, 2010.

[26] M. Martin, B. Livshits, and M. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA*, 2005.

[27] National Vulnerability Database. CVE statistics. `http://web.nvd.nist.gov/view/vuln/statistics`, Feb 2011.

[28] V. H. Nguyen, T. K. Dang, N. T. Son, and J. Kung. Query assurance verification for dynamic outsourced XML databases. In *Proc. of the 2nd Conference on Availability, Reliability and Security*, Vienna, Austria, Apr 2007.

[29] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.

[30] R. A. Popa et al. Enabling security in cloud storage SLAs with CloudProof. Technical Report MSR-TR-2010-46, Microsoft Research, 2010.

[31] Privacy Rights Clearinghouse. Chronology of data breaches. `http://www.privacyrights.org/data-breach`.

[32] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD*, 2004.

[33] H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *Proc. of the 10th NDSS*, 2003.

[34] E. Shi, J. Bethencourt, H. Chan, D. Song, and A. Perrig. Multi-dimensional range query over encrypted data. In *IEEE S&P*, Oakland, CA, 2007.

[35] V. Shoup. NTL: A library for doing number theory. `http://www.shoup.net/ntl/`, Aug 2009.

[36] R. Sion. Query execution assurance for outsourced databases. In *Proc. of the 31st VLDB*, pages 601–612, Trondheim, Norway, Aug–Sep 2005.

[37] D. X. Song, D. Wagner, S. David, and A. Perrig. Practical Techniques for Searches on Encrypted Data. In *IEEE S&P*, Oakland, CA, 2000.

[38] B. Thompson, S. Haber, W. G. Horne, T. S, and D. Yao. Privacy-preserving computation and verification of aggregate queries on outsourced databases. Technical Report HPL-2009-119, HP Labs, 2009.

[39] L. Xiong, S. Chitti, and L. Liu. Preserving data privacy for outsourcing data aggregation services. Technical report, Emory Univ., Dept. CS, 2007.

[40] Z. Yang, S. Zhong, and R. N. Wright. Privacy-preserving queries on en-

crypted data. In *European Symposium on Research in Comp. Sec.*, 2006.

[41] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *SOSP*, pages 291–304, 2009.