CryptDB: A Practical Encrypted Relational DBMS

Raluca Ada Popa, Nickolai Zeldovich, and Hari Balakrishnan
{raluca, nickolai, hari}@csail.mit.edu

ABSTRACT

CryptDB is a DBMS that provides provable and practical privacy in the face of a compromised database server or curious database administrators. CryptDB works by executing SQL queries over encrypted data. At its core are three novel ideas: an *SQL-aware encryption strategy* that maps SQL operations to encryption schemes, *adjustable query-based encryption* which allows CryptDB to adjust the encryption level of each data item based on user queries, and *onion encryption* to efficiently change data encryption levels. CryptDB only empowers the server to execute queries that the users requested, and achieves maximum privacy given the mix of queries issued by the users. The database server fully evaluates queries on encrypted data and sends the result back to the client for final decryption; client machines do not perform any query processing and client-side applications run unchanged. Our evaluation shows that CryptDB has modest overhead: on the TPC-C benchmark on Postgres, CryptDB reduces throughput by 27% compared to regular Postgres. Importantly, CryptDB does not change the innards of existing DBMSs: we realized the implementation of CryptDB using client-side query rewriting/encrypting, user-defined functions, and server-side tables for public key information. As such, CryptDB is portable; porting CryptDB to MySQL required changing 86 lines of code, mostly at the connectivity layer.

1. INTRODUCTION

Theft of sensitive private data is a significant problem [37]. Database management systems (DBMSs) are an especially appealing target for attackers, because they often contain large amounts of private information. When individual users or enterprises store their sensitive data in a DBMS today, they must trust that the server hardware and software are uncompromisable, that the data center itself is physically protected, and assume that the system and database administrators (DBAs) are trustworthy. Otherwise, an adversary who gains access to any of these avenues of attack can compromise the entire database, as has been documented in a number of published reports of data thefts [37] (and presumably there are more compromises that have not been publicized).

These stringent security requirements are also at odds with cost-saving measures such as the consolidation of DBMSs belonging to different business units into a common enterprise-wide IT infrastructure, moving databases into a public cloud, or outsourcing DBA tasks. In fact, "lack of trust" is an oft-quoted primary concern about moving data in database systems to more cost-effective cloud infrastructures. Moreover, thanks to high-profile thefts of social security identifiers, credit card numbers, and other personal information from various online databases, these concerns are increasingly being reflected in the law as well: for instance, recent legislation requires that all databases containing personal data about Massachusetts residents be encrypted [28].

This paper presents *CryptDB*, a practical relational DBMS that provides provable privacy guarantees without having to trust the DBMS server or the DBAs who maintain and tune the DBMS. In CryptDB, unmodified DBMS servers store *all* data in an encrypted format, and execute SQL queries over encrypted data without having access to the decryption keys. CryptDB works by intercepting and rewriting all SQL queries in a *frontend* to make them execute on encrypted data, by encrypting and decrypting all data, as well as changing some query operators, while preserving the overall semantics of the query. The frontend has access to the encryption key for the entire database. By not giving the DBMS server access to the decryption key, CryptDB greatly reduces trust requirements for a DBMS server. For example, CryptDB can alleviate privacy concerns when outsourcing databases to a cloud computing environment [47], such as Amazon's AWS, Microsoft's SQL Azure, or Google's AppEngine, or when outsourcing the work of DBAs. CryptDB can also prevent privacy breaches due to curious administrators [6] or compromised DBMS servers.

There are three significant challenges in designing and implementing a DBMS that operates on encrypted data. The first lies in supporting a wide range of SQL queries on encrypted data. Unlike a simple encrypted data store, a DBMS must perform *computations* on encrypted data to execute SQL queries. For example, an SQL query may ask for the average salary of employees, for the names of employees whose salary is greater than $60,000, or for the list of employees that share an office with more than two colleagues. Simply encrypting each row in the database with a single key would not allow a DBMS server to execute such SQL queries without access to the decryption key.

The second challenge is to carefully define "privacy" for an untrusted DBMS, as well as come up with a system design that provably achieves that definition. On the one hand, even if all of the data stored on a DBMS server were encrypted, the server must be able to perform certain operations on the rows, such as aggregations, selections, and joins. On the other hand, an adversary that compromises a DBMS server may now learn information about the data, such as relations between different rows in a table. Thus, we need to define

privacy in a way that balances the need for server-side computation with the need to minimize information revealed to the server.

The third challenge lies in making an encrypted DBMS practical to use. To provide good performance, an encrypted DBMS should impose minimal performance overheads on the server, but at the same time avoid offloading SQL query execution onto the client. To make an encrypted DBMS easy to deploy, an ideal system would also require *no changes to existing DBMS server software*, so that it can take advantage of more than four decades of engineering and optimization work, run over a range of commodity DBMS servers, as well as *as make no changes to client applications*.

We address these challenges in CryptDB's design using three key ideas. The first idea is an *SQL-aware encryption strategy*. We observe that SQL queries are composed of primitive operators, notably order comparisons, equality checks, and addition. For most of these operators, we found existing encryption schemes in which the operation can be performed on the ciphertext without knowing the decryption key. One exception is joins, for which no cryptographic primitive existed; in this case, we developed a novel cryptographic construction for privacy-preserving joins. Given these primitives, we encrypt each data item with encryption schemes that enable the server to execute the necessary SQL operators on that data. This approach is both efficient and practical, since the bulk of the DBMS, including query planning, data layout, transaction coordination, and the structure of the queries themselves, can remain the same, and only individual SQL operators used by a query may need to change.

The second idea is *adjustable query-based encryption*, where CryptDB dynamically adjusts the encryption level for each data item at runtime, so as to achieve the maximum privacy level given the user's queries. In particular, CryptDB initially encrypts all data with the strongest level of encryption, and, as the application issues SQL queries, CryptDB adjusts the level of encryption on the server, so that the server can perform the classes of computations necessary for that SQL query. This model forms the basis of our privacy definition, ensuring maximum privacy given the classes of computations required by the queries presented to the DBMS, and avoids the need to modify application code to declare the necessary level of encryption ahead of time.

The third idea is to implement adjustable query-based encryption by encrypting each data item in an *onion of encryptions*, from weaker forms of encryption that allow certain computations, to stronger forms of encryption that reveal no information, as shown in Figure 3. This approach allows CryptDB to efficiently adjust encryption levels on the server without having to re-encrypt all data at the client. For example, the outermost layer uses randomized encryption, which guarantees that the server can learn nothing about the data, aside from its length. If the user issues an SQL query containing `WHERE id=5`, CryptDB sends the server an *onion key* to decrypt the `id` column to a deterministic encryption level, where identical plaintexts have identical ciphertexts.[1] CryptDB then sends the server a deterministic encryption of the constant 5, allowing it to compute matching rows by only revealing the necessary *relations between data items*, and *not revealing the actual data*, or other relations between data items not used in this query.

To our knowledge, CryptDB is the first private system to support all of standard SQL over encrypted data without requiring any client-side query processing, modifications to existing DBMS codebases, changes to legacy applications and offloads virtually all query processing to the server. CryptDB works by rewriting SQL queries, storing encrypted data in regular tables, and using an SQL user-defined function (UDF) to perform server-side cryptographic operations. We have implemented a prototype of CryptDB that works with unmodified Postgres and MySQL databases.[2] Porting CryptDB to a new database is straightforward: our port to MySQL required changing just 86 lines of code. Our experimental results demonstrate that CryptDB is portable and imposes a 27% penalty in throughput for a TPC-C workload compared to an un-encrypted DBMS. We view this overhead as relatively modest and a tolerable penalty in many contexts where the desire for data privacy is more important than achieving the highest level of performance.

[1] CryptDB never gives the server onion keys to decrypt the data to plaintext.

2. THREAT MODEL

The goal of CryptDB is to ensure the *secrecy* of data in an SQL database in the face of *an adversary that has complete access to the database server*. The adversary could compromise the server software, or even physically attack the server by stealing and reading disks. Consequently, CryptDB makes *no assumptions about the database server keeping any data private*. This model covers all of the motivating scenarios we have mentioned so far, such as outsourcing a SQL database to the cloud, protecting against a curious database administrator, and guarding against attackers breaking into the database server machine. On the other hand, CryptDB assumes that the application and the CryptDB frontend (Figure 1) are not compromised, and do not reveal their keys to the adversary. Dealing with attacks on these components, such as SQL injection vulnerabilities or authentication bypass attacks, is outside of the scope of this paper.

For the purposes of this paper, we assume that a malicious server does not change the data or query results. Ensuring integrity for SQL queries has been heavily researched, and we refer to prior literature for techniques to achieve integrity, as follows. Since CryptDB allows the DBMS to process relational queries on encrypted data as it would on plaintext data, most previously-proposed approaches can be readily used with CryptDB. A malicious server can affect three aspects of data security: integrity, freshness, and completeness. Integrity is solved by adding a MAC to each tuple as in [22, 26, 39]. Freshness has been addressed using Merkle hashes or chained hashes [22, 36, 39] and both freshness and completeness of query results are addressed in [32]. Also, [25] and [44] allow the client to verify results of aggregation queries, [49] authenticates joins, and [42] provides query assurance for most read queries.

2.1 Security Definition

Intuition. At a high level, CryptDB's definition of privacy says that CryptDB only reveals the relations between tuples needed by the server to perform certain computations; in other words, CryptDB provides *maximum privacy given the classes of computations needed at the server* to process SQL queries. More specifically, the definition says two things:

- If users request no relational predicate filtering on a column (i.e., they only request projections and computation), nothing about the column content is leaked; if the user requests equality checks on a column, we reveal which items repeat in that column; if the user requests inequality checks on a column, we reveal the order of the elements in the column. We never reveal the actual data content.

- The server cannot process queries (that is, discover new data relations) different from the ones requested by the user.

This intuition suffices to grasp CryptDB's security model, so the reader uninterested in formalism can proceed to the "Implications"

[2] In fact, CryptDB will work with any other unmodified commodity relational DBMS servers too.

part of this section below (and perhaps return to the formal definition of security later).

Definition. To formalize cryptographic definitions, often times one defines an *ideal world* using oracles and then proves that the proposed protocol has as much privacy as the ideal world with respect to polynomial-time adversaries. We will instead use the term *ideal system*.

What is the ideal system considering our goals? It is a system in which the server only learns the data relations that it needs in order to perform typical SQL processing for user queries, and nothing else. This is equivalent to the server having access to *no data content*, but only to an oracle that is willing to answer questions about the relations between data items, restricted only to questions needed to process requested queries.

Given a query Q, let $\mathsf{FUNC}(Q)$ be the classes of computations needed by Q_i; that is, the relations between tuples that the server is allowed to know to process the query. For example, if Q is `SELECT * FROM table1, table2 WHERE table1.c1 = table2.c2`, then $\mathsf{FUNC}(Q)$ consists of questions of the form: "is the i^{th} item in c1 equal to the j^{th} item in c2". Obviously, the server needs to know this information to process a join, but it does not need to know the actual value of the two items. In the Appendix, we describe FUNC precisely.

DEFINITION 1 (IDEAL SYSTEM). *Let $Q_1 \ldots Q_t$ be the queries users requested up to time t. The database at the server consists of each data item encrypted with the strongest encryption scheme (random). Moreover, the server has access to an oracle that only answers questions from* $\mathsf{FUNC}(Q_1) \cup \ldots \cup \mathsf{FUNC}(Q_t)$.

Obviously, in the Ideal System, the server can process SQL queries on encrypted data because all the information it needs are the relations in $\mathsf{FUNC}(Q_1) \cup \ldots \cup \mathsf{FUNC}(Q_t)$, by the definition of FUNC. At the same time, such server always has the database encrypted with the strongest encryption scheme that leaks nothing.

We want the server in CryptDB to learn as much information as the server in the Ideal System and we show this property in §4.

Implications. While CryptDB does not reveal any data item, it does not hide data access patterns. For example, the database server can monitor the frequency with which some data item is returned in a result set. Masking such access patterns from the server would incur significant overhead [33], require virtually an overwrite of the underlying DBMS, add considerable client-side processing, and preclude server-side optimizations.

CryptDB's database server can also check any predicates used to execute past user-requested SQL queries, both at the time the query is executed, and at any later time. For example, if the application were to issue the query `SELECT * FROM table1 WHERE c1 = x2ad412`, the server would be able to check rows for this predicate; however, this will be mostly useless to the server because `x2ad412` is an encrypted value and all that the server learns is the number of rows that match this unknown constant.

Note that, when the server needs to evaluate a predicate P (say, `id = x245ab1?`), the server is allowed to perform the class of equality check computations on column `id` (albeit checking for equality with an encrypted value) rather than just the computation of checking for a specific value. As such, the server can check predicates of the form `c1 = ⋆`, where $*$ is any ciphertext, but whose corresponding value the server does not know. Therefore, what the server learns is the frequency of repeats of values only in column `c1`.

We considered allowing the server to check equality only for the specific constant given by the user, but cryptographic primitives supporting equality checks in this way are very expensive. Moreover, doing so will not bring much additional privacy. The reason is that, typically, user queries end up performing selection on the same column for a variety of different constants; after a few such queries, the privacy of this seemingly-stronger scheme will converge to the privacy of our scheme.

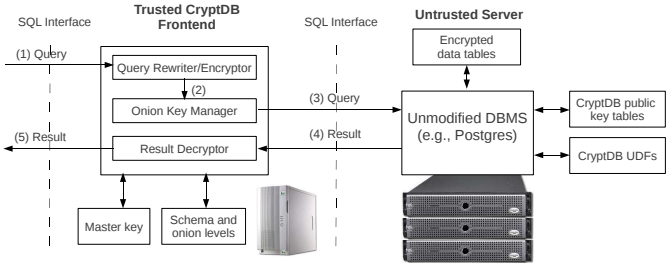

Figure 1: System overview. Steps (1)-(5) illustrate typical query flow, with the user or application issuing query (1) and receiving result (5). UDF stands for user-defined function.

2.2 User-enforced Security

In certain applications, it may be difficult for a programmer to find all of the queries issued by the application, making it difficult to reason about the precise privacy guarantees that CryptDB will provide. In this case, CryptDB allows the programmer to optionally annotate the schema (in the frontend, as defined in the next section) by specifying the lowest security level allowed for each column. For example, the programmer may allow equality checks, but may not want to reveal order within the column. In this case, CryptDB will not allow the server to perform inequality checks, and reject any query that requires such computations.

3. DESIGN OF CRYPTDB

CryptDB works by allowing the DBMS server to execute SQL queries on encrypted data almost as if it were executing the same queries on plain-text data. In CryptDB, the query plan for an encrypted query remains the same as for the original query, but individual operations comprising the query, such as equality comparison or summation, are performed on ciphertexts, and use modified operators in some cases.

Figure 1 shows the architecture of CryptDB. CryptDB is composed of two parts: a trusted client-side *frontend*, and an untrusted DBMS *server*. The frontend keeps track of a secret master key MK, the database schema as seen by the application (i.e., without encryption), and the level of onion encryption currently exposed at the server for each data item. The server, on the other hand, keeps track of the encrypted schema, the encrypted versions of user data (i.e., the lowest level of encryption revealed to the server by the frontend), and auxiliary tables used by CryptDB. The server implements CryptDB-specific UDFs that enable the frontend to compute on ciphertexts on the server. Finally, the server stores encrypted versions of the frontend's state, including the schema and the current onion levels, in a separate table, encrypted with MK, which allows frontends on multiple client machines to synchronize with one another.

Figure 1 also illustrates the typical flow of a query in CryptDB. In step (1), the user issues a query, which is first passed through the query rewriter/encryptor (QRE). This module anonymizes each table and column name. Using the master key MK, this module encrypts each constant in the query with an encryption scheme allowing the desired operation, as we will describe shortly. In step (2), the query is passed to the onion key manager (OKM). This

module assesses if the server should be given onion keys to execute the query. If so, the OKM provides the necessary onion keys by issuing an `UPDATE` query at the server that invokes a UDF to adjust the security of the appropriate columns to increase their functionality. In step (3), the OKM forwards the anonymized query to the server, which executes it using standard SQL (and occasionally invokes more UDFs, as we will explain). In step (4), the DBMS returns the query result, and in step (5) the result decryptor (RD) decrypts the results and returns them to the user. We elaborate on these steps in the rest of this section.

3.1 SQL-aware Encryption Strategy

To implement encryption that allows SQL query processing, we use existing encryption schemes, optimize a recent scheme, and design a new cryptographic primitive for joins, as we will now describe. CryptDB uses the same level of encryption to encrypt all data items in a given column, so that the same computation can be performed on every element in that column.

For each encryption type, we explain the security property that CryptDB requires from it. We explain how to implement it with tools that are believed to achieve such security; if such tools are ever broken, they can be replaced with other tools that provide such property, without breaking the security design of CryptDB.

Random (RND). RND provides maximum privacy, such as indistinguishability under an adaptive chosen-ciphertext attack (IND-CCA2), also known as semantic security. In particular, 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, we use AES in UFE mode [11].

Deterministic (DET). DET has a slightly weaker guarantee: it only leaks which encrypted values correspond to the same data value, and nothing else. 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 content. A recent proposal for OPE [3] is an encryption scheme that preserves order: 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 $\mathsf{OPE}_K(c_2)$ denoting the range c_1 through c_2. Moreover, the server can perform ORDER BY, MIN, MAX, SORT, etc.

OPE is a weaker encryption scheme because it reveals order. Thus, the frontend will only reveal OPE-encrypted columns to the server if users request range queries on those columns. OPE has provable security guarantees: 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 is not equal to $y - x$ except for the sign.

A provably secure OPE scheme was only proposed last year [3]. There was no implementation of the scheme, or any measure of how practical it would be, so we implemented it. The initial performance was about 20 ms per encryption, which was not great, given that each row in a table may have a few items that should be encrypted with OPE. We implemented several optimizations to reduce the cost of OPE. At a high-level, given a value v, OPE performs binary search in the field of encryptions to find an encryption for v. We realized that intermediate search results can be reused across encryptions of different values, so we cache them when performing many such encryptions (e.g., upon database load). To search the cache efficiently we used fast AVL binary search trees. This reduced the cost of OPE encryption to 4 ms, without affecting its security. We also implemented a hypergeometric sampler that lies at the core of OPE. The most efficient such scheme was proposed in 1988 (now used in many tools such as MathWorks) [21], and we translated its original implementation from Fortran-1988.

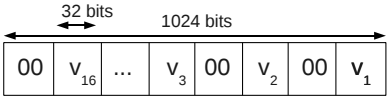

Figure 2: Packing of 16 integer values (32 bits each) into a single 1024-bit value that is subsequently encrypted using HOM.

Homomorphic encryption (HOM). HOM is an encryption scheme that is IND-CCA secure, but allows the server to perform computations on encrypted data, the final result being decrypted by the frontend. While homomorphic encryption for general operations is prohibitively slow [9], homomorphic encryption for summation is efficient. To support summation, we implemented the Paillier cryptosystem [35]. 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. When the server performs summation on a column encrypted with HOM, it calls a UDF that performs Paillier multiplication, and uses CryptDB's public key table to look up K, to homomorphically compute a ciphertext corresponding to the sum of the plaintext values. HOM is secure under a chosen-plaintext attack. HOM can also be used for computing averages (by having the server return the sum and the count, and dividing the decrypted sum by the count in the frontend), and for incrementing values (e.g., `SET id=id+1`), on which we will elaborate shortly.

One drawback with HOM is that the ciphertext length is 2048 bits long for each data value. However, we made the following observation: a row can store just one HOM ciphertext for several integer columns, because each HOM 2048-bit ciphertext corresponds to a 1024-bit plaintext value, and 8 32-bit values from different columns can be packed into one 1024-bit value at different bit offsets (allowing for 32 zero bits between each 32-bit integer value). Individual columns can still be aggregated by having the frontend extract the sum of the desired columns from the appropriate offset in the homomorphically-computed plaintext. Figure 2 illustrates this optimization.

Word search (SEARCH). To allow word searches (e.g., using the "ILIKE" keyword), we implement a cryptographic protocol for keyword searches on encrypted text [1, 43]. SEARCH allows the server to detect repeating words in a given column.

Join (JOIN and OPE−JOIN). A separate encryption level is necessary to allow equality joins between two columns, because the DET encryption level uses different keys for each column. At a high level, JOIN allows the server to compare values between values in columns A and B, given a token from the frontend for columns A and B. JOIN also supports all operations allowed by DET and SEARCH, and also allows the server to detect repeating values between two columns. For inequality joins, OPE−JOIN allows the server to perform inequality joins on any columns at that encryption level. The mechanics of these encryption levels are discussed in more detail in §3.5, along with the entire join mechanism.

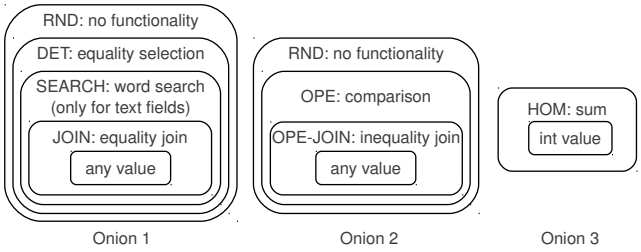

Figure 3: Onion layers of encryption and the classes of computation they allow.

3.2 Adjustable Query-based Encryption

A key part of CryptDB's design is *adjustable query-based encryption*, which dynamically determines the level of encryption to reveal to the server. The answer depends on the queries being asked over the data: if there is no reason to compare data items in a column or sort a column, the column will be encrypted with RND, and for columns that perform equality checks but not inequality checks, DET suffices. Unfortunately, there are many databases where the query set is not known in advance. Thus, we need an adaptive scheme that dynamically "does the right thing" in terms of choosing an encryption strategy for the query at hand.

Our idea is to encrypt each cell independently into an *onion*: each value in the table is dressed in layers of increasingly stronger encryption, as illustrated in Figure 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 privacy, whereas OPE provides more functionality. For numeric values, CryptDB maintains three onions, whereas for string values, CryptDB maintains two onions (i.e., no HOM). To prevent the server from learning information from column or table names, CryptDB's frontend also anonymizes the schema. Figure 4 shows an example server-side schema and data under CryptDB, along with the corresponding plaintexts. Each data item is stored encrypted in at most 3 onions (for integers) or 2 onions (for non-integer values), although CryptDB's optimizations can reduce that to 1 onion for some sets of queries (as discussed in §3.7).

For each level of each onion, the frontend 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 frontend uses the key

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

where PRP is a pseudorandom permutation (such as AES).

Each onion starts out encrypted with the most private encryption scheme (RND for onions 1 and 2, and HOM for onion 3). As the frontend 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 already at an onion layer that allows P, the OKM does nothing. Otherwise, the OKM must strip off the onion layers to allow P on c, by sending the corresponding onion key to the server.

To avoid changing the DBMS, CryptDB implements onion layer decryption using user-defined functions in the server. For example, 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:

Customers

ID	Name
23	Alice

Table1

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

Figure 4: Data layout at the server. When the frontend creates a table with the schema on the left, the table created at the server is the one from the right.

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

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

Note that decryption is performed entirely by the server and not by the client. More importantly, in the steady state, no server-side decryptions are needed, since column decryption happens only when a new type of predicate is performed on a column. For example, after an equality predicate is performed on a column and the server brings the column to level DET, the column remains in that state, and future such queries no longer require decryption. As §6 will illustrate, this ensures that, in the steady state, the overhead of CryptDB is acceptably low. The resulting privacy level to which the database converges is the maximum privacy level for the set of queries issued by the application.

3.3 Query Execution for Read Queries

To execute each SQL query, the frontend anonymizes, encrypts and rewrites the query before forwarding it to the untrusted DBMS server. To anonymize a query, the frontend replaces each table with the table name from the anonymized schema (see Figure 4). For projection, the frontend replaces each column with the name of the anonymized column for the first onion. For executing predicates or aggregates on a column, the frontend replaces the column with the anonymized name of the onion that allows the necessary operation on that field, and, for certain operations (such as `SUM`), the frontend replaces the operation with its equivalent UDF that operates on ciphertexts. Finally, each constant in the query is encrypted with the encryption scheme corresponding to the layer of the onion used by the predicate involving the constant.

To illustrate how this works, consider an example scenario 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 3. At this point, the server can learn nothing about the data content other than the number of columns, rows, and data size.

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 frontend 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 frontend 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 additional server-side decryptions are necessary, and the frontend directly issues the query `SELECT COUNT(*) FROM Table1 WHERE C2-Onion1 = xbb234a`, where x$bb234a$ is the encryption of "Bob".

Finally, the frontend replaces aggregation operators with equivalent UDFs that operate on encrypted values. For example, if the user

issues the query `SELECT SUM(salary) FROM Employees`, the frontend rewrites it to `SELECT HOM_AGG(C4-Onion3, PKTABLE.PK) FROM Table2`, where `HOM_AGG` is a UDF that performs Paillier multiplication (resulting in addition of plaintexts), `PKTABLE` is a table of public keys CryptDB stores at the server (see Fig. 1), and PK is the Paillier public key modulus.

3.4 Query Execution for Write Queries

CryptDB also supports queries that modify data on the server—namely, `INSERT`, `DELETE`, and `UPDATE`. For all modification queries, the frontend applies the same processing to the predicates (i.e., the `WHERE` clause) as for read queries. For `INSERT` queries, the frontend encrypts each inserted column's value with each onion layer that has not been stripped off yet in that column. For `DELETE` queries, no additional processing is performed. For `UPDATE` queries that set the value of a column to a constant, the frontend encrypts the value in the appropriate onions as for `INSERT`.

The remaining case is `UPDATE` queries that update a column value based on another column value, such as `salary=salary+1`. Such an update would have to be performed using HOM, because it allows 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 most TPC-C queries. If a column is used in comparisons after it is incremented, the solution is to split the query into two. That is, for query `UPDATE Employees SET salary=salary+1 WHERE id=3`, the frontend issues the encrypted version of `SELECT salary FROM Employees WHERE id=3`, and then the encrypted version of `UPDATE Employees SET salary=z WHERE id=3`, where z is one more than the result of the first query.[3] However, in most cases in practice (such as in TPC-C), such updates are executed on individual rows.

3.5 Computing Joins

Supporting joins is a challenging problem. If two columns are to be joined, they need to be encrypted with the same key for levels JOIN or OPE−JOIN. We first describe how CryptDB implements deterministic joins (the overwhelmingly common case, i.e., level JOIN), and then describe inequality joins (level OPE−JOIN).

To provide maximum privacy for equality joins, the 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 cryptographic JOIN keys. Moreover, if users request a join of columns A and B, and a join of columns C and D, the server should not be able to join B and C. Thus, the question is, which JOIN keys should each column be encrypted with, given that we do not know in advance what columns will be joined?

[3]If the first `SELECT` query returns more than one result, the frontend computes a mapping of old encrypted `salary` values and the corresponding new encrypted `salary` values, and uses a UDF in the second `UPDATE` query to update the `salary` column according to this mapping.

To address this problem, we propose a new cryptographic primitive that allows the server to dynamically adjust the JOIN encryption keys of each column. Each column is initially encrypted with a different JOIN key, thus disallowing all joins. When the user requests a join, the frontend will give the 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 initially 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 an elliptic curve. When the user requests to join columns c and c', the frontend 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 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 that columns c and c' share the same JOIN key, the server can perform an equality join on c and c' as usual.

THEOREM 1. *The server can only join pairs of columns that were joined by a legitimate user query.*

PROOF SKETCH. We delegate the full proof to the extended version of this paper, and provide only a sketch here. Our scheme is secure (i.e., the theorem is true) because K/K' does not reveal any information about K or K' alone. Moreover, the server cannot compute K from $H(v)^K$ because of the hardness assumption of computing the discrete logarithm in elliptic curve-based groups. Given two columns encrypted with K and K', if the server is not given K/K', it cannot bring the two columns to the same encryption level, and thus cannot perform unrequested joins. □

We chose ECC because of its efficiency: computing $\mathsf{JOIN}_{K'}(v)^{K/K'}$ is basically a multiplication, and does not involve any exponentiations. Moreover, elliptic curve-based ciphertexts are small ($\approx$ 160 bits) compared to ciphertexts in typical public key cryptosystems for the same security level [31].

For inequality joins, a similar dynamic re-encryption scheme is difficult to construct. Instead, CryptDB requires that pairs of columns that will be involved in inequality joins are declared by applications ahead of time, by annotating the schema, so that matching keys are used for level OPE−JOIN of those columns. Alternatively, level OPE−JOIN encryptions can be re-encrypted with matching keys at the expense of sending an entire column to the frontend for re-encryption, and then sending it back to the DBMS server.

3.6 Transactions and Indexes

CryptDB uses existing transaction and indexing mechanisms in the DBMS server without any modifications. For transactions, the frontend passes along any `BEGIN`, `COMMIT`, and `ABORT` queries to the DBMS, and CryptDB does not change any transaction semantics. For indexes, the DBMS builds indexes of encrypted columns much the same way in which it builds indexes of plaintext data. The frontend does not request indexes on RND encryptions, since no lookups are performed at that level, but the frontend does construct indexes on DET, JOIN, OPE, and OPE−JOIN encryptions, if the application requested an index on the corresponding column in the original schema (e.g., using an `ALTER` or `CREATE` query).

3.7 Optimizations

CryptDB implements three important optimizations to improve its performance, which we now describe.

Known query set. For many applications, the queries issued by the application are fixed and known ahead of time. In this case,

we do not need to adjust onion levels at runtime, and can start the database with the exact onion encryption levels that we need. Moreover, if a certain column is never used in certain operations, the corresponding onion can be omitted altogether. For example, if an application never performs range or order operations on a column, the OPE onion for that column can be omitted. Discarding onions significantly reduces performance costs both in the frontend and in the DBMS server. This optimization is implemented by a *train* module in the frontend, which, given a set of queries a user application will issue, determines the exact levels of encryption needed.

Security convergence. Even if the query set is not known in advance, after an application has run for a considerable while on a DB, the frontend 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 frontend can go through the cost of inserting the deleted column, by downloading the entire column and re-uploading it in a different, re-encrypted onion. Since these operations are amortized over long periods of time, the overall performance remains high.

Ciphertext caching. A significant ongoing cost for the frontend lies in generating OPE and HOM encryptions of constants used in queries. To avoid this cost, the frontend maintains a cache of recently-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 frontend encrypting data.

3.8 Discussion and Limitations

CryptDB's design supports most relational queries and aggregates on standard data types, such as integers and text/varchar types. Numeric data of decimal type with p digits after the decimal can be mapped to integer types by multiplying the input by 10^p. CryptDB can encrypt floating-point values, but cannot perform aggregations on floating-point values per se (however, if the values are converted to fixed-point, CryptDB can use HOM for aggregation). Additional operations can be added to CryptDB by extending its existing onions, or adding new onions for specific data types. For example, one could add spatial and multi-dimensional range queries using the protocols proposed by Shi et al [40].

CryptDB has certain limitations. For example, it does not support both computation and comparison *in the same predicate*, such as `WHERE salary > age*2+10`. CryptDB can facilitate processing such a query, but it would require a little bit of processing on the frontend. To use CryptDB, the query can be rewritten into a subquery that selects a whole column, `SELECT age*2+10 FROM ...`, which CryptDB computes using HOM, then re-encrypting the results in the frontend, creating a new column (call it `aux`) at the server consisting of the newly-encrypted values, and finally running the original query with the predicate `WHERE salary > aux`.

The current CryptDB prototype does not support stored procedures or other user-defined functions on the server. Supporting stored procedures written in SQL should be straightforward, by having the frontend rewrite the SQL statements inside of the stored procedure as it would for any other query. On the other hand, CryptDB's design cannot support the execution of arbitrary user-defined functions (not written in SQL) over encrypted data.

Finally, our current prototype handles server-side auto-increment columns by leaving the column values in plaintext on the server. We believe this is an acceptable trade-off, since the server is anyway involved in choosing the auto-incremented value. A more privacy-preserving scheme may involve using HOM to generate the auto-incremented value, but HOM would not allow joins on that column (whereas auto-increment is commonly used for primary key columns that require joins). More generally, CryptDB allows columns that are not privacy-sensitive to be left unencrypted on the server, to reduce overhead and allow more general computations (such as arbitrary UDFs).

4. PRIVACY ANALYSIS

In this section, we prove that CryptDB achieves the privacy goal we set out in §2. To do so, we assume that the cryptographic tools used by CryptDB actually provide the properties we require. The constructions we proposed as implementations in the previous section are secure based on some cryptographic assumptions; if such assumptions are ever broken, the cryptographic constructions can be replaced by others that have the same security guarantees, thus preserving the security of the overall CryptDB design.

THEOREM 2. *The server in CryptDB learns as much information about the data in the database as does the server in the Ideal System from Definition 1.*

PROOF SKETCH. The proof is by induction on the queries requested. The base case of the induction is when no queries have been requested so far. Since we start out the database with RND, we satisfy the base case.

In the inductive step, we need to prove that, by processing the i^{th} query, Q_i, the server only learns as much information as the relations contained in $\mathsf{FUNC}(Q_i)$. Such proof is by exhaustively considering all types of operations; for brevity, we only consider the most basic ones here, leaving the remainder for the extended version of this paper. Suppose Q_i contains an equality predicate: `c1 = x1c5a21`. Then, $\mathsf{FUNC}(Q_i)$ contains the relation {`c1 =` $*$}. CryptDB will lower the level of `c1` to DET. By assumption about the security level provided by DET, the server can only check whether some encrypted data from `c1` equals some other encrypted data from `c1`, all of which is already permitted by `c1 =` $*$ from $\mathsf{FUNC}(Q_i)$ because the server can replace $*$ with any value from `c1`.

Now consider that Q_i contains an inequality check `c1 > x1c5a21`, which means that $\mathsf{FUNC}(Q_i)$ contains {`c1 >` $*$}. In this case, CryptDB will reveal the OPE encryptions of column `c1` to the server. By assumption, OPE only leaks order between items, that is, `item i` $\overset{?}{>}$ `item j`. However, this information is already permitted by $\mathsf{FUNC}(Q_i)$ because the server can replace `c1` with item i and $*$ with a value from item j in when posing a question to the oracle from the set of questions allowed by {`c1 >` $*$}. □

5. IMPLEMENTATION

We implemented CryptDB in C++ on top of Postgres 9.0, and later ported it to MySQL, as reported in §6.4. We used the NTL library for doing number theory on large numbers [41] to implement some of our cryptographic protocols. CryptDB is 4700 lines of code, not counting empty lines, standard libraries, the NTL library, or evaluation code.

As mentioned earlier, CryptDB does not change the innards of a DBMS. We managed to implement all the server-side functionality with UDFs and server-side tables. The insight into why such modular change was possible is that the DBMS in fact lies in between two aspects of query processing that CryptDB must modify. Specifically, query planning and execution is between query parsing and low-level operations on data items. Therefore, CryptDB can run on top of any DBMS that is SQL-compatible and supports UDFs, by rewriting queries in the frontend and replacing individual operations through UDFs.

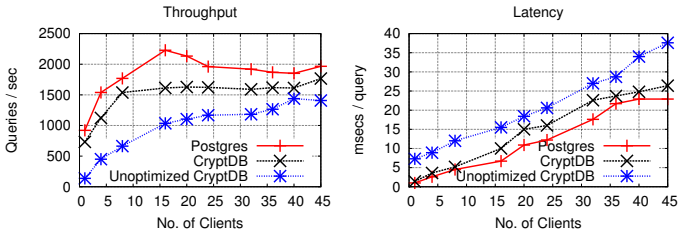

Figure 5: Throughput and latency for TPC-C queries without transactions, as a function of the number of concurrent clients.

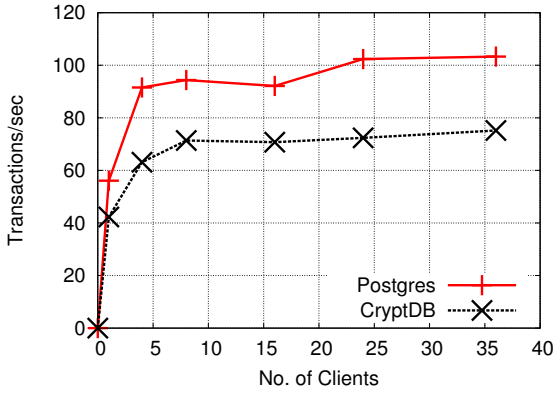

Figure 6: Throughput of TPC-C transactions as a function of the number of concurrent clients.

6. EXPERIMENTAL EVALUATION

This section evaluates three aspects of CryptDB's design and our prototype implementation: performance overheads, portability of CryptDB to different DBMS servers, and portability of CryptDB to different SQL applications. The results shown in the rest of this section show that CryptDB has low runtime overheads (on the order of 27% throughput cost and 0.64 ms latency cost for TPC-C queries), is easy to port to new DBMS servers, and requires no application code changes.

6.1 Overall Performance Results

To measure CryptDB's performance, we use an experimental setup consisting of a server that has an Intel Xeon 3.20 GHz CPU with 4 cores and 3 GB of RAM, and a client machine that has an Intel Xeon 1.6 GHz E7310 CPU with 16 cores and 8 GB RAM, on which we simulate multiple users. Since we are interested in measuring the overhead of CryptDB's cryptographic transformations, we use a workload where the entire data set fits in RAM on the DBMS server machine, and the DBMS server is not bottlenecked by disk accesses (which are equally slow with and without CryptDB).

For our performance experiments, we measure the throughput and latency of CryptDB and an unmodified Postgres DBMS using a TPC-C trace. Rather than just run the TPC-C benchmark, we generate a mix of random OLTP queries by collecting TPC-C execution traces and then having multiple clients present these queries to CryptDB. This approach allows us to evaluate the performance overhead of CryptDB on a variety of random OLTP-like access patterns.

A significant part of CryptDB's CPU overhead running TPC-C queries is masked by the ciphertext caching optimization, because TPC-C queries repeatedly use a small set of constants, which may or may not be representative of other workloads. Thus, we also report results for CryptDB with the ciphertext caching optimization disabled (called unoptimized CryptDB in our results), which achieves lower but still reasonable performance.

To measure the raw number of queries the server can process per second, we first report the throughput for TPC-C queries when transactions are not used, as shown in Figure 5. We can see that CryptDB incurs a throughput reduction of 21% compared to an unmodified Postgres server (based on the highest throughput for both configurations), and unoptimized CryptDB incurs a 35% overhead. Furthermore, because of the CPU cost of encrypting data in the frontend, unoptimized CryptDB requires more clients (running on different cores) to achieve maximum throughput. Finally, the difference in latency between CryptDB and the unmodified Postgres server is within 4 ms.

Figure 6 shows the throughput achieved for the TPC-C query mix when transactions are enabled. In this case, CryptDB incurs a 27% penalty in terms of throughput compared to unmodified Postgres, largely due to increased transaction contention because of longer client-side processing times and expanded queries. In case of unoptimized CryptDB, client-side processing times for each query are increased even further, leading to significantly more transaction conflicts and an overall throughput reduction of 70%.

6.2 Query Microbenchmarks

To understand the sources of overhead incurred by CryptDB, we examined the throughput of individual queries, since different applications may result in various query mixes. For each query type, we collected corresponding queries from TPC-C, and measured the latency for those queries running under CryptDB and under unmodified Postgres. The results of this experiment are shown in Table 1. We can see that the client encryption time is generally low, adding an average of 0.34 ms to the query. The unoptimized CryptDB also has low latency except for queries requiring OPE and HOM; the overall client latency of 7.3 ms may still be acceptable for some applications. The fact that server latencies are similar for CryptDB and unmodified Postgres suggests that the expansion factor due to encryption has a moderate impact on performance. In summary, overall CryptDB adds 0.64 ms of latency to each query, and unoptimized CryptDB adds 7.6 ms.

Figure 7 shows the throughput for the same mix of queries running under CryptDB and unmodified Postgres. We can see that for six query types (Select equality, Select join, Select range, Delete, Insert, and Update set), the throughput overhead is minimal. These six query types constitute most of the queries both for TPC-C and likely for many other applications, making CryptDB a good choice. Homomorphic operations, such as Select sum and Update increment, incur a significant overhead with CryptDB, due to the server-side cost to homomorphically multiply large cryptographic numbers instead of adding 32-bit integers. Applications that use sums and increments heavily would incur significant overhead with CryptDB, but applications with a low percentage of sums (such as in TPC-C), CryptDB overheads would be much lower.

In terms of storage, the frontends only need to store one master key, the schema and onion status, as well as ciphertext caches for OPE and HOM as an optimization. For TPC-C queries, the memory footprint of the frontend process is minimal: 92 kB of memory, not including the code and shared libraries like libc, or less than 4 MB of memory including code and shared libraries. If the frontend decides to cache ciphertext for OPE and HOM, to cache 100000 values (presumably the most common), it takes < 1 MB for OPE and ≈ 12 MB for HOM.

On the server side, CryptDB increases the size of the database due to multiple onions and ciphertext expansion. For TPC-C, the

Query	CryptDB			Postgres
	Server	Client	Client Unopt.	Server
Select equality	0.43	0.10	0.10	0.41
Select join	0.72	0.27	0.27	0.63
Select range	1.2	0.40	58.2	0.99
Select sum	8.8	0.18	0.18	0.46
Delete	1.1	0.15	0.14	1.1
Insert	1.0	0.34	18.6	0.99
Update set	1.2	0.17	0.17	1.1
Update increment	2.0	0.71	17.7	1.8
Overall	1.4	0.34	7.3	1.1

Table 1: Latency figures in milliseconds. The client latency is the encryption time for CryptDB and unoptimized CryptDB. The overall time reflects the average latencies for the mixture of TPC-C queries. "Update set" indicates an update where fields are set to a constant, and "Update increment" indicates an update where fields are incremented.

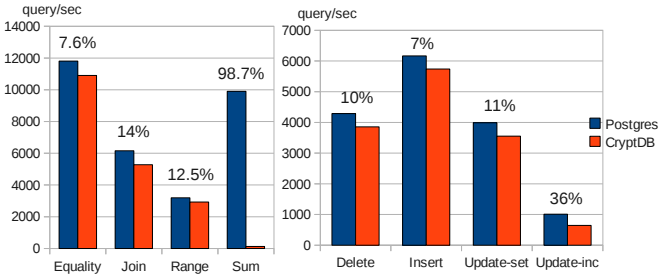

Figure 7: Throughput comparison for query types from Table 1 running under CryptDB and unmodified Postgres. For each query type we show the percentage throughput reduction under CryptDB as compared to unmodified Postgres.

database size using unmodified Postgres was 135 MB and with CryptDB, it was 619 MB, amounting to 4.5 times increase. This is mostly because of aggregates which are large cryptographic numbers, without which it is a three fold increase. Since disk space is relatively cheap, we do not consider increased storage cost to be a significant barrier to adoption of CryptDB. CryptDB's runtime overheads are significantly less than the storage expansion factor because reading a column (or evaluating a predicate on a column) requires accessing only one of the onions for that column, instead of every onion for that column. Moreover, there is virtually no expansion for large data items such as long strings or binaries. Therefore, a database storing large text or binaries (e.g., photos) will have almost no storage overheads.

6.3 Adjustable Query-based Encryption

Adjustable security involves decrypting columns to lower onion levels. Fortunately, such decryption itself is fast, and only needs to be performed once per column for the lifetime of the system. In particular, in our implementation, removing a layer of RND encryption requires decrypting AES, which our server can perform at approximately 500 MB/sec. Thus, the cost of removing an onion layer of encryption is bottlenecked by the speed at which the DBMS server can copy an entire column of data in a single table; the decryption speed is negligible in comparison with disk read and write bandwidth.

Table 2 shows the resulting state of the encryption levels of the various fields in TPC-C's schema after executing the TPC-C queries. We can see that 71% of the columns remain at RND which means

RND	DET but not OPE	OPE	Uses HOM	Total
65	19	8	8	92

Table 2: Resulting onion state of the database for TPC-C.

that no information leak about them whatsoever. This shows the importance of dynamically adjusting encryption based on queries rather than just starting out with encryption that enables all operations. For OPE only, 2 columns have range queries, the other 6 are decrypted to OPE due to `ORDER BY`. One could avoid such decryption by simply ordering the items on the frontend.

6.4 Server Portability

To demonstrate the portability of CryptDB, we ported CryptDB to MySQL. We only had to change and add a total of 86 lines of code. This code was mostly connection code, allowing the CryptDB frontend to connect to the MySQL server, a new format for UDF declarations (though the content was the same), and different handling of information sent to and received from the server. However, we did not change any of the DBMS code in the MySQL server, and did not change any code for CryptDB's logic in the frontend. This suggests that CryptDB can be easily ported to other DBMS servers that support UDFs.

6.5 Application Portability

To demonstrate the portability of existing applications to CryptDB, we ran several applications on CryptDB, including TPC-C and two different versions of an academic institution's graduate admissions web application, one written using hand-written SQL queries and one written using the Django ORM system [12] for Python. CryptDB was able to support all SQL queries from these applications without having to modify the queries or the application in any way.

7. RELATED WORK

At a high level, the main contribution of CryptDB over prior work is a practical novel approach for guaranteeing data privacy in a DBMS. To our knowledge, CryptDB is the first private system to support all the operators used commonly in SQL, perform virtually all the query processing on the server, work without modifying the internals of existing relational DBMS codebases or client applications, and run at a fairly modest performance degradation.

We divide related work into applied cryptography, providing tools for processing encrypted data, but not a comprehensive systems solution, systems approaches done in the context of implemented designs, and theoretical approaches.

Applied cryptography.

Over the past few years, researchers have developed cryptographic tools for searching keywords over encrypted text [4, 43] and some researchers have proposed using these tools to process SQL queries on encrypted data [2, 5, 14, 51]. These approaches are a good first cut at the problem, but are incomplete in substantive ways: they do not support many basic SQL queries, providing mostly only equality comparisons, most of them require significant client-side query processing, require changing the innards of a DBMS; most are too inefficient, e.g., requiring users either to build and maintain indexes on the data at the server or else to perform sequential scans for every selection/search; finally, many remained at the level of cryptographic protocols and did not build and demonstrate a system. Nevertheless, these approaches are useful for private text search, and we use a similar method for "ILIKE" to the ones in [1, 43]; however,

they do not constitute a complete or efficient privacy-preserving DBMS.

Amanatidis et al. [1], slightly different from the schemes above, allow the server to build indexes, but their privacy is significantly lower than CryptDB's: they reveal *a priori* all repeating values and all common ranges (same prefix values) in the whole database (across all columns and rows). Key to our security is the dynamically-adjustable encryption based on queries: fields that are never used in range queries or equalities should remain encrypted with the strongest encryption scheme. Finally, they do not support all other queries besides equality and prefix-based ranges, and did not build a system.

In addition, the privacy provided by the most secure of these previously developed schemes is approximately similar to CryptDB. For example, in Yang et al.'s work [51], after one query has been made, the server only knows the repetitions of the constant in the selection filter, and does not know all the repetitions of data in a column as in CryptDB. Therefore, after one equality query, their protocol is more secure. However, after a few different queries with the same structure, but different constants, CryptDB reveals as much information as their approach. These are specific tools and do not enable to server to perform other requirements in a DBMS: general range queries, updates of a whole column or range of tuples (e.g. increments), aggregates, some cannot support joins.

There is also work allowing the server to build secure indexes [18] on encrypted data without having access to the data. These approaches require significant changes to the DBMS and would not be as portable. Moreover, due to complex cryptographic tools they are slower. In our case, the server builds indexes naturally as it would on typical (longer) numbers. Ge and Zdonik [15] enable comparisons and designs indexes for a column store.

There has been work on processing queries on encrypted XML [20, 46]. This work provides useful security semantics for XML data, but requires a cumbersome DBMS rewrite. Moreover, the results sent to the client are a superset of the true result, requiring the client to perform additional post-processing. They also do not discuss updates and inserts.

Systems approaches.

While early proposals attempted to enable SQL processing over encrypted data, their privacy mechanisms were heuristic without formal guarantees, required a significant rewrite of the DBMS design, relied on considerable client-side processing, and did not support a wide range of SQL queries.

In [19], Hacigumus et al. split the domain of possible values for each column in partitions. Each partition has a number (random or order-preserving) and each value is replaced with the number of the partition. The actual tuple is stored encrypted. By grouping elements in partitions, privacy may be compromised because an untrusted entity may well know which elements are close to each other in value. The more partitions, the more useful work the server does, but the less privacy clients have. With our dynamically-adjustable encryption mechanism, we do not reveal relations for columns not used in a filter.

Ozsoyoglu et al. [34] use user-defined functions and do not require changes to the DBMS software. However, their approach only applies to integers, does not support joins (must be processed at the client), do not discuss updates and inserts. Importantly, they encrypt fields with a *distance*-preserving encryption function for which $a - b = E(a) - E(b)$. Such scheme is not secure because the mere knowledge of one value a for an encryption $E(a)$ leaks the decryption of all other encrypted values.

Damiani et al. [10] provide a solution for processing range queries over encrypted data. Each element in a tuple is encrypted and hashed to a small number of buckets to improve privacy. Range queries can be computed using a B+ tree, but these are done by the trusted client that needs to traverse the B+ tree by sequentially performing queries at the server. Such work does not support joins, aggregates, or string searches.

Ciriani et. al [8] propose a new approach to confidentiality: replacing data encryption with fragmentation; they store a part of the data at the trusted client (e.g. sensitive columns or relations between columns) and the rest of the data unencrypted at the untrusted server, thus avoiding encryption altogether. However, each client (trusted) has to store a potentially large amount of data locally and has to perform query processing whenever sensitive data is involved in a query.

Chow et al. [7] require the presence of two additional parties, a randomizer and a query engine, which are assumed not to communicate; moreover, the data is stored at multiple DBMSs that do not trust each other. The security of their protocols hinge on such security assumptions and on no collusions happening; such a setting is not always possible, and solutions without such strong trust assumptions may be desirable.

Theoretical approaches.

Theoretical solutions promise a high degree of security, however, they are prohibitively impractical. Recent work [16] based on fully homomorphic encryption [17] enables an untrusted party to perform any general computation on encrypted data (such computation could be query processing) without leaking even access patterns to the data. Unfortunately, as an example, performing a simple string search using homomorphic encryption is about a trillion times slower than without encryption [9].

Work in PIR [33] allows a user to request a tuple from the database without the server learning what tuple was requested. While they provide excellent security, such approaches are highly infeasible because, for each tuple requested, the server must scan the whole database.

Other work.

There has been a significant body of work on distributed privacy preserving data integration, aggregation, and mining: constructing decision trees [27], computing association rules, classification and clustering [23, 24, 45, 50], outsourced data aggregation [48] and [44]. These works provide a rich and useful set of privacy-preserving tools for different purposes than CryptDB, and they can be used together with CryptDB.

Work on differential privacy [13, 29, 38], introduced by Dwork and enhanced by Miklau [30], as well as privacy in statistical databases, allows a trusted server to decide what answers to release or how to obfuscate answers to aggregation queries to avoid leaking information about any specific record in the database. Such work has a different goal and model from CryptDB; in fact, one can add differential privacy to the front end in CryptDB to provide to users only privacy-preserving answers.

8. CONCLUSION

This paper presented CryptDB, a practical and novel system for ensuring data privacy on an untrusted SQL DBMS server. CryptDB uses three novel ideas to achieve its goal: an *SQL-aware encryption strategy*, *adjustable query-based encryption*, and *onion encryption*. As part of CryptDB's SQL-aware encryption strategy, we propose optimizations for existing cryptographic techniques, as well as a new cryptographic mechanism for private joins. Our prototype of CryptDB requires no changes to application or DBMS server

code, and uses user-defined functions to perform cryptographic operations inside an existing DBMS engine, including both Postgres and MySQL. Under a TPC-C workload, our prototype incurs a 27% reduction in throughput compared to an unencrypted DBMS, making CryptDB a practical option for privacy-sensitive data.

Acknowledgements

The authors would like to thank Carlo Curino and Sam Madden for helpful feedback. Carlo Curino and Evan Jones provided traces used in our TPC-C evaluation. Eugene Wu and Alvin Cheung also provided useful advice. This work was supported by the NSF (CNS-0716273) and Google.

References

[1] G. Amanatidis, A. Boldyreva, and A. O'Neill. Provably-secure schemes for basic query support in outsourced databases. In *Proceedings of the 21th Annual IFIP WG 11.3 Working Conference on Database and Applications Security*, pages 14–30, Redondo Beach, CA, July 2007.

[2] F. Bao, R. H. Deng, X. Ding, and Y. Yang. Private query on encrypted data in multi-user settings. In *Proceedings of the 4th International Conference on Information Security Practice and Experience*, pages 71–85, Sydney, Australia, 2008.

[3] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *Proceedings of the 28th Annual International Conference on Advances in Cryptology*, pages 224–241, Cologne, Germany, 2009.

[4] D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. In *Proceedings of the 4th Conference on Theory of Cryptography*, pages 535–554, Amsterdam, The Netherlands, 2007.

[5] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Proceedings of the 3rd Applied Cryptography and Network Security Conference*, New York, NY, June 2005.

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

[7] S. S. M. Chow, J.-H. Lee, and L. Subramanian. Two-party computation model for privacy-preserving queries over distributed databases. In *Proceedings of the 16th Network and Distributed System Security Symposium*, 2009.

[8] 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 *Proceedings of the 14th European Symposium on Research in Computer Security*, 2009.

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

[10] E. Damiani, S. D. C. di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational DBMSs. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, Washington, DC, October 2003.

[11] A. Desai. New paradigms for constructing symmetric encryption schemes secure against chosen-ciphertext attack. In *Proceedings of the 20th Annual International Conference on Advances in Cryptology*, pages 394–412, August 2000.

[12] Django Software Foundation. Django: The web framework for perfectionists with deadlines. `http://www.djangoproject.com/`, October 2010.

[13] C. Dwork. Differential privacy: a survey of results. In *Proceedings of the 5th International Conference on Theory and Applications of Models of Computation (TAMC'08)*, pages 1–19, Xi'an, China, 2008.

[14] S. Evdokimov and O. Guenther. Encryption techniques for secure database outsourcing. Cryptology ePrint Archive, Report 2007/335, 2007. `http://eprint.iacr.org/`.

[15] T. Ge and S. Zdonik. Fast, Secure Encryption for Indexing in a Column-Oriented DBMS. *ICDE*, 2007.

[16] R. Gennaro, C. Gentry, and B. Parno. Non-Interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers. In *Proceedings of the STOC*, 2010.

[17] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, pages 169–178, New York, NY, USA, 2009. ACM.

[18] E.-J. Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. `http://eprint.iacr.org/2003/216/`.

[19] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, Madison, WI, June 2002.

[20] R. C. Jammalamadaka and S. Mehrotra. Querying encrypted XML documents. In *Proceedings of the 10th International Database Engineering and Applications Symposium*, pages 129–136, Washington, DC, USA, 2006. IEEE Computer Society.

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

[22] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 29–42, Berkeley, CA, USA, 2003. USENIX Association.

[23] M. Kantarcioglu and C. Clifton. Security issues in querying encrypted data. Technical Report CSD TR 04-013, Purdue University, Department of Computer Sciences, 2004.

[24] M. Kantarcioglu and C. Clifton. Security issues in querying encrypted data. In *Proceedings of the 19th Annual IFIP WG 11.3 Working Conference on Database and Applications Security*, 2005.

[25] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Authenticated index structures for aggregation queries in outsourced databases. *Technical Report BUCS-TR-2006-011*, 2006.

[26] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 91–106, San Francisco, CA, December 2004.

[27] Y. Lindell and B. Pinkas. Privacy preserving data mining. In *Journal of Cryptology*, pages 36–54. Springer-Verlag, 2000.

[28] V. Maine and Associates. Massachusetts Data Security Law 201 CMR 17.00, 2010.

[29] F. McSherry. Privacy integrated queries. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, Providence, RI, June–July 2009.

[30] G. Miklau. Boosting the accuracy of differentially-private queries through consistency. In *Proceedings of the 36th International Conference on Very Large Data Bases*, Singapore, September 2010.

[31] National Security Agency. The Case For Elliptic Curve Cryptography. `http://www.nsa.gov/business/programs/elliptic_curve.shtml`, 2009.

[32] V. H. Nguyen, T. K. Dang, N. T. Son, and J. Kung. Query assurance verification for dynamic outsourced XML databases. *Conference on Availability, Reliability and Security*, 2007.

[33] R. Ostrovsky and W. Skeith. A survey of single-database private information retrieval: Techniques and applications. In *Proceedings of the Public Key Cryptography*, pages 393–411, 2007.

[34] G. Ozsoyoglu, D. A. Singer, and S. S. Chung. Anti-tamper databases: Querying encrypted databases. In *Proceedings of the 17th Annual IFIP WG 11.3 Working Conference on Database and Applications Security*, pages 4–6, Estes Park, CO, 2003.

[35] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology - EUROCRYPT '99*, pages 223–238. Springer-Verlag, 1999.

[36] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling Security in Cloud Storage SLAs with CloudProof. *Microsoft Technical Report*, 2009.

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

[38] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for mapreduce. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation*, San Jose, CA, April 2010.

[39] H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *Proceedings of the 10th Network and Distributed System Security Symposium*, 2003.

[40] E. Shi, J. Bethencourt, H. Chan, D. Song, and A. Perrig. Multi-dimensional range query over encrypted data. In *Proceedings of the 28th IEEE Symposium on Security and Privacy*, Oakland, CA, May 2007.

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

[42] R. Sion. Query execution assurance for outsourced databases. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 601–612, Trondheim, Norway, August–September 2005.

[43] D. X. Song, D. Wagner, S. David, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 21st IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000.

[44] 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.

[45] J. Vaidya and C. Clifton. Privacy preserving association rule mining in vertically partitioned data. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 639–644, 2002.

[46] H. W. Wang. L.V.S.: Efficient secure query evaluation over encrypted XML databases. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, Seoul, Korea, September 2006.

[47] Xeround, Inc. Private beta rollout progress update and some metrics. `http://blog.xeround.com/2010/09/27/private-beta-rollout-progress-update`.

[48] L. Xiong, S. Chitti, and L. Liu. Preserving data privacy for outsourcing data aggregation services. Technical report, Emory University, Department of Mathematics and Computer Science, 2007.

[49] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated join processing in outsourced databases. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 5–18, Providence, RI, June–July 2009.

[50] Z. Yang, S. Zhong, and R. N. Wright. Privacy-preserving classification of customer data without loss of accuracy. In *Proceedings of the 5th SIAM International Conference on Data Mining*, pages 21–23, Newport Beach, CA, 2005.

[51] Z. Yang, S. Zhong, and R. N. Wright. Privacy-preserving queries on encrypted data. In *Proceedings of the 11th European Symposium on Research in Computer Security*, 2006.

APPENDIX

We describe in more detail $\mathsf{FUNC}(Q)$ (defined in §2.1), for any Q an SQL query. A relation is an expression of the form `{x OP y}`, where `OP` can be any operation (such as $=, >, <, \geq, \leq$, `ILIKE`) that can be performed between columns and/or constants. x and y are either names of columns in a database or the wildcard $*$ (meaning any constant value). An instantiation of a relation is an expression where each column operand is replaced with the index of an item in the column, and $*$ is replaced with some value. For example, an instantiation of `{table1.c1 = const}` is `{item i of table1.c1 = x1c5a21}`. The evaluation of an instantiation is a boolean value: true if the instantiation of the relation is true and false otherwise.

Then, $\mathsf{FUNC}(Q)$ is the set of all relations for which there is an instantiation whose evaluation the server needs to know in order to perform SQL processing given Q.

For example, let `Q: SELECT MAX(table1.c1) FROM table1, table2 WHERE table1.c2 = table2.c1 AND table2.c2 = x1c5a21`. Then, FUNC(Q) = { `{table1.c1 > table1.c1}` (for `MAX`), `{table1.c2 = table2.c1}`, `{table2.c2 = *}` }. For joins, if we have a predicate of the form `c1 = c2`, this introduces {`c1 = c2`}, {`c1 = *`} and {`c2 = *`} in FUNC(). Inserts and aggregates do not increase FUNC(). Updates by incrementation do not increase FUNC() either; Updates by setting a value such as `SET c1 = x1c5a21` add $c1 = *$ to FUNC; Deletes add relation corresponding to their filters as discussed above.