# Transaction Processing on Confidential Data using Cipherbase

Arvind Arasu[1], Ken Eguro[1], Manas Joglekar[2], Raghav Kaushik[1], Donald Kossmann[1,3], Ravi Ramamurthy[1]

[1]*Microsoft Research,* [2]*Stanford University,* [3]*ETH Zurich*

***Abstract*—Cipherbase is a comprehensive database system that provides strong end-to-end data confidentiality through encryption. Cipherbase is based on a novel architecture that combines an industrial strength database engine (SQL Server) with lightweight processing over encrypted data that is performed in secure hardware. Cipherbase has the smallest *trusted computing base (TCB)* among comparable systems and provides significant benefits over the state-of-the-art in terms of security, performance, and functionality.**

**This paper presents a prototype of Cipherbase that uses FPGAs to provide secure processing and describes the system engineering details implemented to achieve competitive performance for transactional workloads. This includes hardware-software co-design issues (e.g. how to best offer parallelism), optimizations to hide the latency between the secure hardware and the main system, and techniques to cope with space inefficiencies. All of these optimizations were carefully designed not to affect end-to-end data confidentiality. Our experiments with TPC-C show that when customer data is strongly encrypted in Cipherbase, it provides 90% the throughput of SQL Server operating over unencrypted data. Even when all data is strongly encrypted, Cipherbase achieves 40% the throughput of plaintext SQL Server.**

## I. Introduction

Concerns over data security and data breaches [1] hinder the adoption of cloud technologies. Data is a valuable asset and unprotected data in the cloud is vulnerable to hackers and snooping administrators. This motivates database systems that can guarantee strong *data confidentiality*, where an adversary, ideally, cannot learn anything about the data. This would address security vulnerabilities and could spawn new opportunities in cloud *database-as-a-service*.

There is a rich body of academic [2], [3], [4], [5] and industry efforts [6], [7] that seek to ensure database data confidentiality using *encryption*. The general idea is that by keeping data encrypted and limiting access to encryption keys, we can keep the data confidential. In the end, we want database systems that, while preserving data confidentiality, are *general* and *performant*. General means that the system supports full SQL and rich features such as stored procedures, transactions, and recovery. Furthermore, a legacy application should run unchanged, except for a thin shim to handle encryption/decryption. Performant means that throughput and latency should be competitive with a state-of-the-art database system that does not provide confidentiality. In Section I-A we argue that all prior work suffers from fundamental limitations in the three desired dimensions: data confidentiality, generality, or performance.

This paper introduces Cipherbase, a new database system for achieving data confidentiality through encryption. Cipherbase is the first system that provides industrial strength transactional functionality with competitive performance while keeping all data that might be accessible to an adversary strongly encrypted. Cipherbase is based on a novel architecture that extends a traditional database system with *lightweight trusted computation* running on secure hardware. This lightweight trusted computation is a unique design point and provides a number of security, functionality, performance, and software engineering benefits. This basic architecture can be modified to build other interesting systems that trade off, e.g., performance for stronger data confidentiality; see [8] for initial conceptual ideas.

### A. Prior Work and Limitations

Query processing is a fundamental challenge for database systems that use encryption to provide confidentiality—encryption "hides" data, making it difficult to perform operations over it. There are two broad techniques for addressing this [9]: (1) use *homomorphic encryption* schemes that allow computation directly over ciphertext, or (2) use a *trusted module* that is inaccessible to the adversary to temporarily operate on data in plaintext. All prior systems, including Cipherbase, can be viewed as instantiations of (1) and (2).

Fully homomorphic encryption schemes [10] that allow arbitrary computation over ciphertext are currently very slow. However, there are practical *partial homomorphic encryption (PHE)* schemes that support limited operations. CryptDB is a recent system that relies purely on PHE for query processing [3]. Since PHE schemes do not support all operations, there exist simple queries (including many in the TPC-C benchmark) that such systems cannot support without giving up confidentiality. Another limitation of the PHE-only approach is that data confidentiality is closely tied to, and thus limited by, the operations performed over the data. For example, if we need to apply equality predicates on a column, the values must be stored using deterministic encryption, thereby revealing the frequency distribution.

Some prior work [2], [4] adds the capability to use the client for trusted computation. These systems use PHE to perform encrypted query processing in the server, to the extent possible, and then use a fallback plaintext database system running on the client to perform any remainder query processing. Again, since PHE schemes do not support all operations, there exist simple queries that require extensive data movement or a large number of roundtrips over a relatively low-bandwidth, high-latency network. In [11] we have shown that client-based approaches can incur up to two orders of magnitude slowdown on TPC-C with strong encryption.

TrustedDB [5] is a recent system that introduces the idea of adding a trusted module to the server, using *secure hardware* to provide a protected location in which to perform processing. The server-based trusted module has higher communication bandwidth and lower latency compared to the client machine, thereby addressing a fundamental limitation. However, TrustedDB runs a *complete* database system within its trusted module, leading to fundamental drawbacks: (1) the *trusted computing base (TCB)* of TrustedDB (i.e. the code with access to client encryption keys and plaintext that needs to be free of vulnerabilities) is an entire database system. Ensuring and convincing the cloud consumer that such a large code base is free of vulnerabilities seems unrealistic. (2) Current secure hardware such as the cryptographic co-processors used in TrustedDB are an order-of-magnitude slower than regular (e.g., Intel) processors. Because of this the performance of TrustedDB suffers considerably when most of the data is confidential and must be handled in the co-processor. (3) The resource constraints of the secure hardware forces TrustedDB to run SQLite [12], limiting functionality.

### B. Cipherbase: Salient Features

The basic architecture of Cipherbase confers several advantages over current state-of-the-art systems:

1. *Lightweight Trusted Computing Base:* Like TrustedDB, Cipherbase relies on in-server secure hardware. Unlike TrustedDB, the logic in the trusted hardware is a simple stack machine used to evaluate expressions over encrypted values and not perform full query processing. The overall size of TCB is a few thousand lines of Verilog. Code of this size is amenable to formal verification to prove it is free from bugs and backdoors. This verification is currently underway.

2. *Rich Functionality:* The architecture of Cipherbase minimally modifies an industrial-strength database system (SQL Server), farming out low-level expression evaluation logic to the trusted module. This design allows Cipherbase to inherit rich functionality from SQL Server for "free". For example, Cipherbase supports *all* of the concurrency and recovery features of SQL Server without any additional code. Currently, Cipherbase can run the full TPC-C benchmark with all data strongly encrypted, while supporting all necessary features such as stored procedures, indexes, and recovery.

3. *Strong, Interpretable Data Confidentiality:* Like most prior systems, Cipherbase supports column-level encryption and an application can pick different encryption schemes for different columns. Unlike most previous systems where the choice of encryption seriously influences performance and/or functionality, the performance of Cipherbase remains robust and the functionality unaffected. For example, when all columns in TPC-C are changed from plaintext to strongly encrypted, the performance drops by about a factor 2. This number can be around 100 for a client-based system [11].

At the same time though, reasoning about end-to-end data confidentiality is subtle for any system that uses secure hardware. This is because an adversary may learn about the underlying data just by looking at patterns of data movement to and from the trusted module. In spite of such dynamic information leakage, the end-to-end confidentiality of Cipherbase can be simply and favourably characterized using previous well-understood systems such as CryptDB [3] (Section II).

4. *Performant:* The lightweight trusted module helps Cipherbase achieve high performance. All but a tiny fraction of the actual processing happens in the insecure host system, which has significantly more computational and memory resources. This division of labor between untrusted and trusted hardware is carefully engineered to ensure that end-to-end data confidentiality is not undermined.

5. *FPGA as Secure Hardware:* While most of the design of Cipherbase is agnostic to the choice of secure hardware, our current prototype relies on a PCIe-based FPGA board. While FPGAs have a history of use in database systems [13], our use is novel and non-standard. Unlike prior systems which place the FPGA along the datapath to the main processing system, our FPGA are part of the inner loop of query processing. For example, in the TPC-C NewOrder transaction, there are about 1000 (unoptimized) roundtrips to the FPGA. At first glance, it appears that the latency associated with this many roundtrips should affect Cipherbase performance. However, we show that with careful hardware-software co-design these roundtrips can be managed without seriously degrading performance.

## II. End-to-End Functionality and Security

This section presents the end-to-end functionality of Cipherbase, discussing data confidentiality and the threat model.

A client application interacts with Cipherbase just as it would with a standard database system: using ad-hoc queries, stored procedures, and SQL scripts. A client library (*Cipherbase client*) mediates between the application and the Cipherbase server, providing the application with a transparent interface.

The data confidentiality requirement for each column can be independently specified using a column-level encryption scheme. Informally, Cipherbase ensures that whenever a value (or derived value) is in the cloud, outside of the secure hardware module, it is encrypted using the specified (or stronger) encryption scheme. The supported encryption schemes are vetted standards: (1) *strong encryption*, which is non-deterministic (meaning that multiple encryptions of the same plaintext will produce different ciphertext). This provides strong data-at-rest security—formally, *indistinguishability under chosen-plaintext attack (IND-CPA)*. It is implemented in Cipherbase with AES in CTR mode [14]; (2) *deterministic encryption*, which always generates the same ciphertext from a given plaintext value. This allows plaintext equality to be checked without first decrypting by directly comparing ciphertext values—useful for operations such as equality filters, joins, and grouping. It is implemented in Cipherbase with AES in ECB mode and format-preserving encryption [15].

Figure 1 illustrates columnar encryption for an example instance of the the *Account* table of the TPC-C benchmark.

| *AcctID* | *AcctBal* | *Branch* |
|---|---|---|
| 1 | 100 | Seattle |
| 2 | 200 | Seattle |
| 3 | 200 | Zurich |

| *AcctID* | *AcctBal (RND)* | *Branch (DET)* |
|---|---|---|
| 1 | !@#$xyz | 0x123 |
| 2 | @%^abc | 0x123 |
| 3 | ⋆&#pqr | 0x363 |

Fig. 1. (a) Example Plaintext and (b) Encrypted Database

Figure 1a shows the original, plaintext instance of the table. Figure 1b shows the encrypted version of that table using a policy that specifies that the *AcctID* column is stored in plaintext, the *AcctBal* column is probabilistically encrypted (RND), and the *Branch* column is deterministically encrypted (DET).

### A. Threat Model

Our goal is to ensure data confidentiality from entities with privileged OS and database access. To characterize this threat, we introduce the *strong adversary*, who has unbounded observational power of a cloud server and can view the contents of the server's memory/disk at every instant, along with all external and internal communication. A strong adversary cannot observe state or computations within the secure hardware—as discussed in Section IV, it is specifically designed to provide such protection.

We also consider a related threat, exemplified by a datacenter operator with physical access to a cloud server and who can walk away with the contents of the server's disk/memory (e.g., using a cold boot attack). To model this threat, we introduce the *weak adversary*, who can obtain a one-time snapshot of *data-at-rest*. Data-at-rest refers to any data that persists across query lifetimes such as data in disk and buffer pool. As the naming suggests, a strong adversary can learn anything a weak adversary is able to learn.

Both strong and weak adversaries are idealizations that help characterize data confidentiality, and the power of real-world adversaries would be in-between these two. For example, the data center operator above might get access to the state of some queries running at the time of the snapshot, so he is slightly stronger than a weak adversary. We assume that both strong and weak adversaries are *passive (honest but curious)* and do not tamper with contents of the database or influence query processing; extending Cipherbase to address an *active* adversary is future work.

### B. Data Confidentiality

We specify Cipherbase's data (and query) confidentiality guarantees by describing what an adversary can (and cannot) learn. For presentation simplicity, we only cover the case in which all columns are strongly encrypted.

As a starting point, Cipherbase does not provide metadata confidentiality beyond name anonymization. Thus, even a weak adversary learns the number of tables, the number of columns in each table, (primary) key properties, the cardinalities of tables and the lengths of data values. Beyond this, though, Cipherbase provides high data confidentiality against the weak adversary that directly corresponds to the encryption level specified for each column. In particular for strongly encrypted columns, data remains strongly encrypted in the server's disk and memory, so a weak adversary cannot learn anything additional from a system snapshot. One exception to this is range indexes, which will be discussed shortly.

Against a strong adversary, Cipherbase provides *operational* data confidentiality; i.e. the adversary learns some information, based on the operations performed over the data. Ideally, we would want *semantic security* where a strong adversary does not learn anything. However, this is an impractical goal since even communicating encrypted results over an untrusted network reveals some information. In Section VII-A, we consider proposals for stronger-than-operational data confidentiality and argue why they are impractical for a transactional system.

Operational data confidentiality has been discussed in prior work. We adapt the formalization presented by CryptDB [3]. The data confidentiality provided by Cipherbase is equivalent to that of an *abstract system* that uses an *oracle* for computations over strongly encrypted values. For non-boolean operations such as addition, the oracle takes in encrypted inputs and produces encrypted outputs—thus, the adversary does not learn any additional information. For boolean operations such as checking a filter condition or checking equality for joins and grouping, the oracle returns the boolean result in plaintext—thus, the strong adversary learns certain relations for encrypted values.

Figure 2 summarizes the information leaked to a strong adversary by various relational operators. For example, to evaluate $\sigma_{A=5}(R)$, the abstract system provides encrypted $A$ values to the oracle, which returns *true* if a value is 5 and *false* otherwise. Since the details of the predicate are internal to the oracle, the adversary only learns the value of some unknown predicate $p(A)$ over $R$ tuples. For the join $R \bowtie_A S$, the oracle is used to check equality of $A$ values for the $R$ and $S$ tuples. This reveals the join graph and the equivalence relations of column $A$ values for the $R$ and $S$ tuples participating in the join. Grouping on attribute $R(A)$ likewise invokes the oracle for checking equality of $A$ values, revealing the equivalence relation of $R(A)$; note that this reveals the number of distinct values and their frequencies. Finally, the query $\pi_{A+B}(R)$ does not involve any boolean operations. Thus, it does not leak any information to the strong adversary. (The above information leakage is reference based, not value-based, which can used to show that it subsumes access pattern leakage.)

Indexes on tables have different data confidentiality guarantees. Cipherbase supports two kinds of indexes: *equality* indexes that support equality lookups and *range* indexes that support range lookups. An equality index provides operational data confidentiality against a strong adversary; i.e. when Cipherbase uses an equality index for the lookup $\sigma_{A=5}(R)$, the adversary learns which of the $R$ records have the same $A$ value, but does not learn the common value (5). In the limit, if the entire key space is queried, the adversary learns the full equivalence relation of column $A$. At the same time, a weak adversary does not learn anything from an equality index. On the other hand, for a range index Cipherbase reveals

| Operation | Strong | Weak |
|---|---|---|
| $\sigma_{A=5}(R)$ | Unknown predicate $p(A)$ over $R$ tuples | Nothing |
| $R \bowtie_A S$ (hash-based) | The join graph and the equivalence relations over $R(A)$ and $S(A)$ for joining $A$ values. | Nothing |
| $\pi_{A+B}(R)$ | Nothing | Nothing |
| $\mathcal{G}by_A^{\mathrm{SUM}(B)}(R)$ (hash-based) | The equivalence relation over $R(A)$ | Nothing |
| $\sigma_{A=5}(R)$ using an equality index on $A$ | Set of $R$ tuples having an unknown common $A$ values | Nothing |
| $\sigma_{A=5}(R)$ using a range index on $A$ | Set of $R$ tuples having an unknown common $A$ values (& ordering) | Ordering of $R$ tuples based on $A$ values |

$(a)$ Cipherbase

| Strong $\simeq$ Weak$^{\#}$ |
|---|
| Equivalence relation over $R(A)$ |
| The join graph and the equivalence relation over $R(A)$ and $S(A)$ for *all* $A$ values. |
| Nothing |
| The equivalence relation over $R(A)$ |
| The equivalence relation over $R(A)$ |
| The ordering of $R$ tuples based on $A$ values |

$(b)$ CryptDB

Fig. 2. Operational Data Confidentiality of Cipherbase and CryptDB. $^{\#}$Strong adversary, in addition, learns queries and their access patterns.

the full ordering information of the index keys, even to a weak adversary. In other words, range indexes in Cipherbase provide similar confidentiality guarantees as *order preserving encryption (OPE)* [16]. However, existing constructions of OPE either leak higher-order bits of plaintext values [16] or require a client-side component and do not guarantee encryption *immutability* [17]. (Informally, in a mutable encryption scheme, the encryption of a value could change when other values are inserted.) The range indexes of Cipherbase do not have these limitations (there is an additional distinction for non-clustered indexes which we discuss in Appendix A).

*Comparison with prior work:* All prior systems using PHE [3], [4] provide weaker operational data confidentiality (summarized in Figure 2 using CryptDB as a representative). Their guarantees are weaker since (1) operational information is leaked even to the weak adversary and (2) the scope of information leakage is an entire column, not just the data touched during query processing. As an example, to be able to evaluate $\sigma_{A=5}(R)$, values of column $A$ need to be stored on disk using deterministic encryption. This reveals the full equivalence relation. For the same query, if Cipherbase uses a scan-based plan, only the truth values of a hidden predicate $p(A)$ over $R$ tuples are revealed (an index-based plan would reveal the equivalence relation, limited to the records in the output). Client-based systems [2], [4] can avoid operational information leakage, but only at the cost of extensive data shipping. TrustedDB has similar operational guarantees as Cipherbase since it does not hide data access patterns.

## III. Overview

This section presents the architecture of Cipherbase and gives an overview of query and transaction processing.

### A. Design Philosophy

A large part of database processing does not deal with data value semantics. This includes moving data, query setup/result communication, and almost all of concurrency control and recovery. Almost all of this processing is unaffected whether we are handling plaintext or encrypted data. Cipherbase exploits this observation and uses a regular database system (SQL Server) to realize this functionality.

At the same time, a core part of database processing does depend on data value semantics, including query expression evaluation over tuples and comparisons during index traversals. In general, this cannot be performed directly over encrypted data (e.g. for strongly encrypted data we cannot perform basic operations such as equivalent plaintext comparison or addition). In Cipherbase we factor out such processing and run it in a *trusted module (TM)*. The trusted module is designed to be inaccessible to the adversary, allowing us to securely decrypt data and perform operations on plaintext.

More concretely, the Cipherbase TM only supports simple *expression evaluation* over encrypted basic types—no other aspect of database functionality. Such a small TM not only provides well-recognized security advantages [18], [19], it also provides benefits related to concurrency/recovery, multi-tenancy, and database administration. A detailed discussion of these advantages is beyond the scope of this paper.

A small TM containing only expression evaluation does introduce performance challenges: we potentially need to make one TM roundtrip for every operation—for the TPC-C NewOrder transaction this translates to over 1000 roundtrips. When the TM runs on a separate device, this design could incur large latencies. The technical core of this paper presents optimizations that manage this latency for an FPGA-based trusted module that demonstrates the viability of this design.

### B. Architecture

Figure 3 presents the Cipherbase architecture. An application interacts with Cipherbase through the *Cipherbase client* module. The Cipherbase client presents a plaintext database interface and hides data confidentiality details from the application. This client performs query encryption before sending the request to the Cipherbase server and decrypts returning results before forwarding them to the application. To implement this functionality, the Cipherbase client takes the encryption schema (see Section II) and the client encryption key as input. The Cipherbase server consists of the trusted module (*TM*) and a modified (SQL Server) database system (the *untrusted module, UM* or *UMDBMS*).

*Design of the TM:* Figure 4 shows the design of the Cipherbase TM. The TM is a simple *stack machine* that can evaluate *stack programs* using an instruction set that supports comparison,

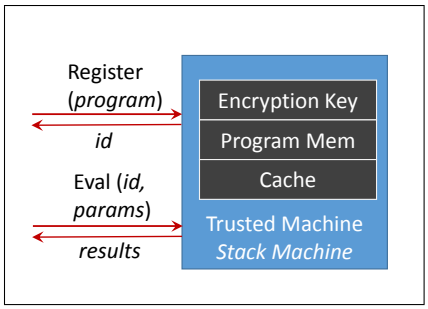


Fig. 4. Design of the Trusted Machine

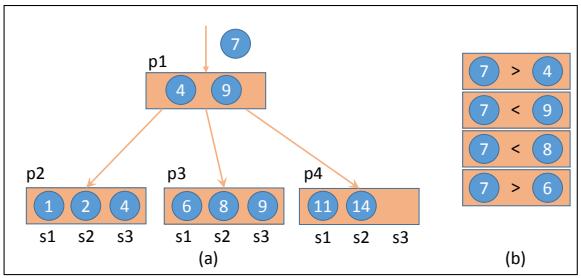


Fig. 5. A single-column range index. Strongly encrypted ciphertext is shown with a blue circle over the plaintext (not visible to the adversary). Page ids ($p1$-$p4$) and slot ids ($s1$-$s3$) within pages are also shown. Also shown are comparisons and their plaintext results while inserting (encrypted) key 7. The results of comparisons ensure that key 7 is inserted between keys 6 and 8.

arithmetic, and cryptographic operations. The TM stack programs are used to evaluate expressions over encrypted data during query processing. For example, a scan-based evaluation of $\sigma_{A=5}(R)$ might use a stack program that takes in an encrypted $A$ value, decrypts it and outputs $true$ if the value is 5 and false otherwise. This stack program would be invoked once for each tuple of $R$. As this example suggests, the same stack program is typically invoked multiple times. To accommodate this usage pattern, a stack program is *registered* once and invoked subsequently using the handle returned by the register method. The TM also supports a data cache that we leverage for performance. Apart from the data cache and the registered programs, the TM is stateless. In our current prototype, the TM is realized using FPGA as secure hardware (Section IV), but most of our design is agnostic to the choice of secure hardware; see Section IV for a discussion.

*Design of the UMDBMS:* The UMDBMS is a regular database system (SQL Server) modified to farm out expression evaluation to the TM. These changes are discussed below.

### C. Implementation of Core Functionality

*Encryption Key Management:* For clarity, in this paper we assume that a single master key is used for encrypting the database, relevant portions of the query, and the results. In practice, the system could use multiple keys, either derived from one customer's master key or keys for different cloud *co-tenants*. Handling the former case is straightforward [3], but performance engineering for the latter is beyond the scope of this paper. These keys are securely communicated to the trusted module using standard key-exchange techniques; the manufacturer of the secure hardware device assigns a public-key identity for running the TM.

*Data Encryption:* Cipherbase uses *cell-level encryption* for data in the UM, meaning that each column of each record is encrypted independently. For base table columns the encryption scheme is specified (Section II) and for derived columns the encryption scheme is inferred from its lineage. Internally, encrypted data types are *aliased* to binary data types, so most of the UMDBMS code can handle these types transparently. Cell-level encryption allows each cell to be accessed independently, minimizing TM traffic during query processing. This design also means that database loading and insertions (when not concerning an index) do not involve the TM. One drawback of cell-level encryption is the storage overhead for smaller data types: e.g. a 4-byte integer encrypted with AES-CTR produces a 12-byte ciphertext (with IV inlined). We address this by exploiting the prefix-compressibility of IVs in the CTR mode using SQL Server compression feature (see Section VII-B for more details).

*Indexing:* As discussed in Section II, Cipherbase supports equality and range indexes, which differ in terms of supported functionality and confidentiality guarantees. Cipherbase supports indexes over multiple columns and they can be primary or secondary, clustered or non-clustered. All variants are implemented using B-trees in the UMDBMS.

Although what is stored for a range index over strongly encrypted columns is strongly encrypted ciphertext, the indexed keys in the B-tree are ordered by their plaintext values. When building the index or performing operations such as lookups or updates, comparisons are routed to the TM. The TM has pre-registered programs that decrypt the provided ciphertext pair and return the comparison result ($<, =, >$) as plaintext. The TM comparisons during an index build or update ensure the indexed keys are ordered by their plaintext values. Figure 5 shows a sample range index and the comparisons performed on ciphertext values while inserting a new key. Note that the vast majority of index processing, including latching/locking for concurrency and managing page splits/merges during updates, remains unchanged, regardless of encryption. Also note that since the B-tree is stored in the UM, a weak adversary can learn the plaintext ordering of indexed keys. However, a weak adversary does not learn equality relationships: E.g., in Figure 5, the adversary knows that the key stored in slot $s1$ is $\leq$ the key in slot $s2$, but cannot determine if these two encrypted keys are equal. A strong adversary learns the equality relationships of keys, but only for the queried ranges.

An equality index is implemented by first deterministically encrypting index keys and then storing them in the range index. Note that the index keys are still stored strongly encrypted, but their relative ordering in the B-tree is determined by their deterministic encryption, not plaintext values. Using the conjectured pseudo-random permutation properties of standard deterministic encryption schemes, we can show

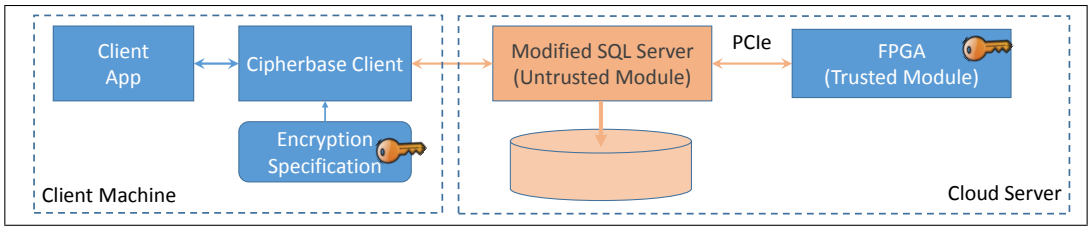


Fig. 3. Cipherbase Architecture

that a weak adversary does not learn any ordering or equality information. A strong adversary learns equality information for the queried portion of the index key space.

*Transaction Processing:* In Cipherbase, query processing happens in two steps: a one-time *prepare* step, followed by any number of parametrized *execution* steps. During the prepare step for a query $Q$, the Cipherbase client first identifies all the TM programs required to evaluate expressions in $Q$ and registers these programs with the TM. The Cipherbase client encrypts the programs to be registered to ensure that an adversary does not learn their details. In particular, a TM program can include internal constants and we leverage this to hide query constants. After the TM programs are registered, the client rewrites $Q$ to reference the program handles returned during registration. Consider the following parametrized query $Q_{ex}$:

```
UPDATE Accounts SET Balance = Balance + @Amt
WHERE Id = @Id
```

Assume both `Balance` and `Id` columns are strongly encrypted, with an equality index on `Id`. This query requires a TM program that takes in two strongly encrypted integers and returns the strongly encrypted sum. Let 21 be the handle for this program returned by the TM during registration[1]. The original query is rewritten[2] as $Q'_{ex}$:

```
UPDATE Accounts
SET Balance = TMEval(21, Balance, @Amt)
WHERE Id = @Id
```

where *TMEval* is a new built-in function we have added to SQL Server; *TMEval* is an n-ary function that invokes a TM stack program. The rewritten query is `PREPARE`d in the Cipherbase UMDBMS (SQL Server), which compiles and caches the query plan. This `PREPARE` step has no knowledge of encryption and is unmodified SQL Server code.

In our example query, assuming an index-based plan, the equality index identifies the record with `Id` equal to the encrypted `@Id` parameter. The encrypted `Balance` of this record and the encrypted `@Amt` parameter are passed as inputs to TM stack program 21, which returns the encrypted sum of the two, which is used to update the `Balance` field. This update happens in the UM and the *Update* operator "thinks" it is replacing one binary value with another.

[1]We have added a new built-in function *TMRegister()* to SQL Server to register stack programs.

[2]For simplicity, we show rewritten queries - see Appendix D for more details.

During query execution, the Cipherbase client encrypts any query parameters and runs the rewritten query at the UMDBMS[3]. Query execution at the UMDBMS proceeds largely agnostic to encryption, using unmodified SQL Server code except for the two cases described earlier : (1) index lookups over strongly encrypted data involve comparisons in the TM and (2) any calls to the *TMEval* function are routed to the TM. To the rest of the query processing system, *TMEval* looks like any other $n$-ary built-in function.

Note that partial homomorphic encryption schemes such as OPE or DET encryption can help avoid round-trips to the TM. For instance, in the above example, if the `Id` column is deterministically encrypted, then the index lookup does not need to involve the TM (similar to the case when `Id` is in plaintext).

The Cipherbase client ensures that the output of a TM program is a strongly encrypted value, a cryptographic hash, or a plaintext boolean. Specifically, the output of a TM program is plaintext only for boolean operators (equality for joins/group-by and filter predicates), which are consistent with the data confidentiality guarantees in Section II-B.

*Concurrency and Recovery:* One of the main advantages of our minimal TM design is that we inherit almost all SQL Server concurrency and recovery features without making any encryption-related modifications. SQL Server, like most modern databases, uses a combination of latching and locking to ensure physical and logical consistency. These primitives are almost always acquired on physical structures, even for logical concepts (e.g., physical record address for record level locking), and work correctly since encrypted data is simply stored as binary values. One subtlety relates to *key-range locking* [20], where locks are obtained on leaf-level key values. However, the correctness of this scheme relies only on the keys being unique (the ordering information is implicit in the position of the keys in the B-tree). Since uniqueness remains unaffected when we replace plaintext keys with ciphertext values, we do not require encryption-specific modifications for key-range locking.

The ARIES-style recovery of SQL Server relies on physical page-oriented redos and logical undos [21]. The correctness of page-oriented redos relies only on the binary representation of data and is unaffected by encryption. Logical undos are affected by encryption, but use the same code path as regular transaction processing. Thus, the query processing modifications described earlier are sufficient.

[3]We omit details of "type inferencing" to infer `@Amt` and `@Id` in the example require strong encryption.

| Feature | UMDBMS (Sql Server) | TM (FPGA) |
|---|---|---|
| SELECT query | Storage engine(IO, Buffer pool, locking, latching)<br>Query Processing (Operator memory management, spooling, data flow, admission control, inter-query resource governance) | Query expressions (e.g., equality, range comparisons, GetHashCode, Addition, Multiplication, String Concatenation) |
| UPDATE query | All of the above + Data update +<br>Constraint checking + logging and recovery | Query expressions (same as above) |
| $B$-Tree Indexing | Page organization, Tree organization, Page splits and merges, Concurrency control (including key value range locking) | Comparison |
| SQL Extensions (e.g. loops, if-then-else) | Program control flow, memory management | Program expressions |

Fig. 6. Summary of division of work between UMDBMS and TM in Cipherbase

## IV. FPGA AS SECURE HARDWARE

The TM is an abstraction that provides a location for code execution and data that is not accessible to the adversary; Cipherbase relies on this abstraction to provide data confidentiality. The current Cipherbase prototype uses FPGA-based *secure hardware* to realize the TM. The use of special purpose hardware to provide isolation (e.g. memory protection) is established practice. The use of FPGAs for security applications also has precedence [22]. In particular, there are several benefits to using an FPGA as secure hardware in a cloud setting, as discussed in [23].

At the same time, the design of Cipherbase allows other physical realizations of the TM. This flexibility is a consequence of our minimal TM design—e.g., running a TrustedDB style full DBMS on an FPGA would be impractical. An interesting future instantiation possibility is the recently announced *Intel Software Guard Extensions (SGX)* [24]. This is a set of new instructions that allow the creation of a process within a protected address space, known as an enclave. Data and computation within an enclave is shielded from all other processes, including the OS. The TM is a natural candidate for execution within an enclave and exploring this when SGX-enabled processors are available is future work.

Our FPGA platform communicates with the host CPU over a standard *PCIe* bus. Other alternative communication mechanisms exist, such as HyperTransport/QPI or deploying an array of FPGAs as a network appliance. Exploring these architectures is future work, and [25] offers a general discussion of the pros and cons of these types of systems.

Regardless of the specific platform used, any instantiation of the TM will have two performance characteristics that are relevant to the optimizations we present in Section V: the bandwidth of communication and latency of round-trip communication between the UM and the TM. Since the TM must be isolated from the UM to provide the desired confidentiality guarantees, this implies data transfer and/or synchronization. Thus, both bandwidth and latency concerns are likely fundamental to the architecture and apt to determine the performance of the overall system.

There are also some performance characteristics that may be specific to the FPGA/PCIe platform we have used to prototype. For example, there are two alternative ways of communicating over PCIe: (a) *MMIO (memory-mapped IO)* and (b) *DMA (direct memory access)*. As shown in Figure 7, MMIO communication is only suitable for very small transfers. On the other hand, DMA offers much better performance for larger transfers, but has non-trivial startup cost; in the platform used for our testing, DMA transfers from 4B-16KB incurred the same latency. Another potential issue is that FPGAs are inherently highly parallel, but have limited sequential compute power [26]. Lastly, FPGAs have a small amount of fast on-chip memory and much larger, slower off-chip memory (on the order of a few MBs vs. GBs respectively).

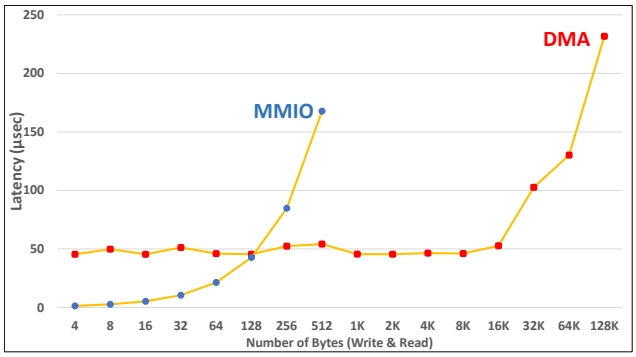


Fig. 7. PCIe Communication. Latency of loopback host/FPGA transfer.

## V. IMPLEMENTATION DETAILS AND OPTIMIZATIONS

This section presents system engineering details and optimizations that we implemented to achieve competitive transaction processing performance. We note that the the goal of this section (and next) is to establish the viability of the overall Cipherbase design and not necessarily be exhaustive in the set of optimizations.

### A. Transaction Processing Challenges

Cipherbase farms out expression evaluation over encrypted data to the TM. Although expression evaluation comprises a relatively small part of the processing in a typical transaction [27], there are generally a large number of individual expressions and they are interspersed with other processing, such as getting locks, latches, and moving pages to the buffer pool. As an example, the TPC-C NewOrder transaction has more than 1000 distinct expressions, including index comparisons. In Cipherbase, this can translate to a large number of fine-grained TM accesses, shown schematically in Figure 8. Note that each (unoptimized) TM call involves a relatively small amount of data; e.g. encrypted addition requires sending $12 \times 2$ bytes of input to the TM and receiving 12 bytes of output.

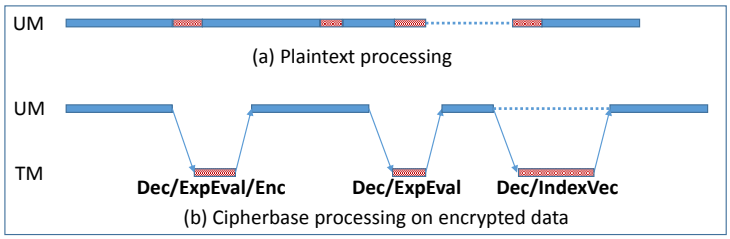


Fig. 8. Transaction processing in Cipherbase involves a large number of fine-grained TM accesses.

| Optimization | TM Concurrency | FPGA Compute | PCIe B/W | Xact Latency |
|---|---|---|---|---|
| Multiple FPGA Cores | √ | √ | | √ |
| Work unit batching | √ | | | ? |
| Index vectorization | | √ | ? | √ |
| Expression folding | √ | √ | √ | √ |
| Functional caching | | | | √ |
| Data caching | | | √ | |

Fig. 9. Benefits of different optimizations

Such tight-coupling of UM processing with TM roundtrips naturally increases transaction latency as compared to plaintext processing. This is both because every TM roundtrip incurs a PCIe transfer latency and because computation performed over encrypted data in the FPGA is likely to be slower than plaintext computations performed locally on the server. While adding even a few milliseconds might be acceptable in terms of an individual query's latency (especially in a cloud setting), as we will discuss, added latency in expression evaluation can indirectly reduce throughput.

TM is a shared resource in Cipherbase and this can affect concurrency and potentially reduce throughput. For concreteness, assume a naïve implementation where the FPGA/PCIe subsystem appears as an *undivided* shared resource to the rest of the system. Every transaction now requires exclusive access (e.g., using a lock) to make a TM call. If each of 1000 TM calls in the NewOrder transaction takes 10 $\mu$sec, using Amdahl's law, the maximum throughput we can achieve is 100 tps. To achieve higher throughput, we need to enable concurrent access to the TM, and we achieve this through a combination of parallelism at the FPGA and PCIe communication batching. Batching is a general technique that leverages the DMA characteristics shown in Figure 7, by sending and receiving multiple *work units* in a single communication over the PCIe. (We use the term *work unit* to refer to the payload—input parameters and output result—of a TM call.) Several of our optimizations build upon this general idea.

Increasing TM concurrency increases the utilization of different components and can expose newer bottlenecks. In our naïve example, the compute resources at the FPGA and the PCIe bandwidth do not form a bottleneck since they are masked by a bigger bottleneck, the PCIe latency. But the batching optimization could expose these two components as bottlenecks. As more transactions concurrently access the TM, data is transferred and stack programs are invoked at the FPGA at a higher rate. Some of our optimizations are designed to better utilize resources at these components: e.g., we cache data in the FPGA to reduce PCIe traffic and we add special instructions to the stack machine to enable computation reuse. These optimizations increase the achievable throughput before a component becomes a bottleneck.

Finally, the increased latency discussed earlier could itself reduce throughput since transactions hold on to locks (and sometimes latches) longer, which increases data contention and reduces throughput. Some of our optimizations are therefore designed to reduce latency.

### B. Optimizations

The optimizations we present are based on the general principles outlined above. An optimization can provide more than one benefit: e.g., data caching in the FPGA not only reduces PCIe bandwidth, it also reduces FPGA computation since cached data is stored as plaintext (eliminating the need for additional decryption). Figure 9 summarizes our optimizations with benefits they provide.

*Multiple FPGA stack machines:* A simple way to increase TM parallelism is to have multiple stack machine instances on the FPGA that can process up to $n$ TM calls independently. This optimization increases the overall compute resources on the FPGA. Parallel processing cores are feasible since the FPGA resources required for each stack machine is quite small; for details see [28].

*Work unit batching:* This optimization was briefly mentioned in Section V-A. Whenever all stack machines (*SMs*) are busy, the UM enqueues transaction work units. When an SM becomes available, all queued work units are sent as a batch. A SM processes all the work units in a batch before returning the results as a single PCIe communication. In addition to latency advantages based on the non-linear characteristics of DMA transfers mentioned earlier, this design also helps improve throughput—each SM is computing for a longer duration, minimizing PCIe traffic and freeing the bus for other SMs. Note that this optimization could sometimes increase transaction latency, since a work unit waits for all other work units in its batch to complete. However, empirically we found that the throughput advantages outweigh any potential latency penalty. Beyond this, there are other low-level system engineering details required to implement this optimization that we cannot cover in detail. These include (1) asynchronous communication to enable a single "controller" on the UM to feed multiple TM stack machines; (2) techniques to minimize context switches when enqueueing; (3) details of switching between DMA and MMIO transfers based on the batch size.

*Expression folding:* In Section III-C we mentioned that the Cipherbase client rewrites query expressions involving encrypted data to use the *TMEval* function and that each invocation of

*TMEval* translates to a TM call during execution. In expression folding, the Cipherbase client folds "adjacent" expressions into a single *TMEval* call. Expression folding can be viewed as intra-transaction batching and provides similar benefits to those discussed above, including: (1) reduced FPGA computation due to computational reuse; (2) reduced PCIe traffic; and (3) lower transaction latency due to fewer TM calls.

To illustrate expression folding and its benefits, consider the query shown in Figure 10(a) that has two expressions in its `UPDATE` clause, and assume all columns are strongly encrypted. The naïve execution, shown in Figure 10(b), evaluates the two expressions using two *TMEval* calls. Expression folding, shown in Figure 10(c), rewrites this to use a single *TMEval* call. The corresponding combined stack program (7) evaluates both expressions and returns two concatenated encrypted outputs, which are separated in the UM.

This expression folding rewriting, apart from eliminating one TM call and the associated latency, has two other benefits. First, the encrypted parameter `@l_qty` can be decrypted once and used for both expression evaluations; in the naïve rewriting we require two decryptions. This is an example of common sub-expression elimination (CSE). Implementing CSE requires us to go beyond a classic stack machine, since we need to store expression results outside the stack and reference them. The second benefit is that the parameter `@l_qty` only needs to be communicated to the TM once, thereby reducing the consumption of PCIe bandwidth.

*Index lookup vectorization:* The naïve (range or equality) index lookups presented in Section III invokes a TM call for every comparison. For a 3-level B-tree with 512 index keys per index node, this translates to $3 \log 512 = 27$ calls per lookup. With index lookup vectorization, the search key and the vector of all keys stored in a B-tree node are passed to the TM in a single call. The corresponding stack program performs a binary search over the (sorted) vector and returns the position of the search key. Only the keys touched during the binary search need to be decrypted. This change greatly reduces the number of TM calls, from 27 to 3 in our example, reducing transaction latency. It also reduces the work done at the TM since the search key is only decrypted once and used for all comparisons within a single node. At the same time, though, it increases the amount of data shipped to the TM—from $9 + 1$ keys per node to $512 + 1$ keys in our example. Empirically, though, the benefits far outweigh the disadvantages. Furthermore, data caching, discussed in the next section, partially addresses this problem.

*Data caching:* As discussed, FPGAs have a limited amount of on-chip memory. We exploit this memory to implement a simple form of data caching, specifically designed to address the data shipping costs of index lookup vectorization. To accomplish this, we identify commonly accessed index key vectors and cache them in plaintext within the FPGA (note that on-chip memory cannot be observed by the adversary, so sending ciphertext that is decrypted and cached inside the FPGA meets our confidentiality requirements). When we encounter a B-tree node that is cached, we only send the search key and a (single word) handle for the cached vector to the TM. This optimization reduces both the amount of data transferred over PCIe and the work done in the FPGA, since no decryptions are required for cached keys.

Our implementation of data caching is an initial step with several avenues for future explorations: (1) FPGAs have access to much larger off-chip memory where greater amounts of data can be cached and accessed with fairly low latency. This memory is insecure, though, so any cached data needs to be kept as ciphertext. (2) We currently divide the available on-chip memory among different stack machines and replicate the data cache at each of them. Other alternatives are possible, such as sharding the cache or sharing a common cache.

*Function result caching:* A TM stack program is purely functional, in the sense that different invocations of a program with the same encrypted parameters will produce outputs corresponding to the same underlying plaintext (the actual ciphertext output might vary due to non-deterministic encryption). We use this property to implement a function result cache in the UM. The function result cache is more general than the data cache and works for all TM programs, not just index-related ones. A function result cache clearly helps avoid TM roundtrips for operations over deterministically encrypted data (other than equality, which does not require the TM). However, there are subtle benefits even for operations over strongly encrypted data. First, when a lock is not immediately granted in SQL Server(particularly relevant during index lookups), an operation can be repeated with the same input ciphertext. Such repetitions hit in the function result cache. Second, strongly encrypted loop counters in stored procedures start with the same initialization constants and can benefit from a function result cache (see Appendix C for an example).

*Impact of optimizations on data confidentiality:* All of the optimizations we discuss here have been carefully designed to preserve data confidentiality. Batching and parallelism within FPGA do not change the operations being performed. Expression folding actually slightly increases data confidentiality: e.g. when two expressions are folded together, any intermediate data is kept within the TM where it is invisible to an adversary. For index vectorization, we can show that an adversary learns the same information at every node, whether or not vectorization is used. For data and function result caching, we can show that as long as the caching strategy only relies on the access patterns of data (information already available to an adversary) and not the index plaintext values, data confidentiality remains unaffected. (On the other hand, for example if we were to compress cached plaintext index key vectors, the adversary could learn something based on the aggregate amount of data we are able to cache.)

## VI. Performance Experiments and Results

In this section we present an empirical evaluation of Cipherbase. The goal of this evaluation is to quantify the overhead of transaction processing on encrypted data and study

```
UPDATE  STOCK
SET s_ytd =  s_ytd + @l_qty
SET s_qty  = s_qty - @l_qty
WHERE s_id = @li_id
```

(a): Original

```
UPDATE  STOCK
SET s_ytd = TMEval(5, s_ytd, @l_qty)
SET s_qty  = TMEval(6, s_qty, @l_qty)
WHERE s_id = @li_id
```

(b): Naïve rewriting

```
UPDATE  STOCK
SET @var0 = TMEval(7, s_ytd, s_qty, @l_qty)
SET s_ytd = UMExtract(@var0, 0)
SET s_qty = UMExtract(@var0, 1)
WHERE s_id = @li_id
```

(c): Rewriting under expression folding

Fig. 10. Illustration of expression folding

the effects of different optimization techniques. As mentioned earlier, a key emphasis is to establish the viability of our architecture.

### A. Software and Hardware Components

We include a brief description of the implemented functionality and optimizations (summarized in Figure 11):

*Cipherbase Client:* This is a small C# library that relies on a SQL parsing library for query rewriting and identifying expressions. It also contains a simple compiler to produce TM stack programs for single and folded expressions, and code for identifying common sub-expressions and performing simple type inferencing used to determine the encryption scheme for various query identifiers.

*UMDBMS (modified SQL Server):* We added a new module called the *TM Manager* to handle communication with the TM. The TM Manager lives within the "SQL OS" layer that provides low-level primitives to the rest of the SQL Server code. Most of the optimizations discussed in Section V are also implemented within the TM Manager, including batching, routing work to different stack machines, and managing the function result/data cache. Outside the TM Manager, we modified some parts of the SQL Server storage engine to handle index vectorization and route SARGable predicate evaluations to the TM. The primary change to the query engine is the addition of the *TMEval* function, which hides the details of TM calls from the rest of the query engine[4]. Outside of the TM Manager, our changes to the SQL Server code were fairly modest ($\leq$ 1000 line of code); however, this was still a fairly challenging software engineering effort since the changes spanned multiple layers of the stack spread over millions of lines of C++ code.

*TM (FPGA):* We used a Xilinx Virtex 6 FPGA board (XC6VLX240T). We programmed four stack machines on the board, each clocked at 150 MHz and with a data cache of 64 KB. The four stack machines and 64 KB cache is a skyline point; increasing the number of stack machine would have decreased the available cache, and vice-versa. The specific numbers were determined based on experimentation (the utilization of the FPGA is summarized in Appendix B). As summarized in Figure 11, some of our optimizations required changes in the hardware. These include new instructions for optimizations such as index vectorization, an out-of-stack scratch memory to enable shared computations, and a data cache. The hardware implementation was around 6000 lines of Verilog. We have presented more details on the FPGA implementation and its resource requirements in a separate paper [28].

*UM Hardware:* All testing was performed on a dual-processor, 16-core Intel Xeon E5-2640v2 server with dual banks of 32GB quad-channel DDR3 memory. The FPGA board was connected to the server via x4 PCIe v2.0.

### B. Encryption Policies

We compare the performance of production SQL Server operating on plaintext TPC-C data with Cipherbase operating on TPC-C data encrypted using the following policies:

- *Customer*: All personally identifiable information (PII) columns in the `Customer` table (all columns except `C_ID`) are strongly encrypted. All other columns are in plaintext.
- *Strong/Weak*: Index key columns and foreign key columns that reference them are encrypted using deterministic encryption. All other columns are strongly encrypted.
- *Strong/Strong*: All columns are strongly encrypted.

These three policies are interesting points in the data confidentiality spectrum both for practical applicability and for illustrating specific aspects of Cipherbase performance. Recall from Section II that the encryption policy determines the protection we get against the weak adversary. The *Customer* policy is interesting since keeping PII data confidential is often required for compliance reasons (e.g., HIPAA). But identifying PII columns might not be straightforward [29], so schemes that keep all columns encrypted such as *Strong/Weak* and *Strong/Strong* are interesting. The *Strong/Weak* policy keeps index columns encrypted using deterministic encryption, so equality index lookups do not require roundtrips to the TM; *Strong/Strong* requires roundtrips for all operations. The relative performance of Cipherbase on these policies therefore helps separate the overhead of indexing and non-indexing operations for strongly encrypted data. *Strong/Weak* reveals the join graph of tables related by referential integrity to the weak adversary, and this level of data confidentiality might be sufficient for most transactional applications. Finally, *Strong/Strong* represents the upper limit of (operational) data confidentiality and competitive performance with strong/strong would simplify physical design, obviating the need to identify a suitable encryption policy for each column.

### C. TPC-C Benchmark Environment

We use transactions from the TPC-C benchmark for our evaluation. Our evaluation is not a complete run of the TPC-C benchmark and in particular we used the following

[4]Other changes handle non index-nested loop joins and grouping, but are not central to this paper.

| Feature | Client | UMDBMS (Sql Server) | TM (FPGA) |
|---|---|---|---|
| Basic Functionality | Compile, register stack programs<br>Query rewrite, PREPARE<br>Encrypt parameters | QE: New functions *TMEval* & *TMRegister*<br>SE: Send index comparisons & SARGable predicates to TM | Stack machine |
| FPGA Parallelism | | TMM: Route work to different SMs | Replicate stack machines |
| Work unit batching | | TMM: Queueing, batching work units | Batched SM input and output |
| Index vectorization | | SE: Batched search at B-tree node level | Binary search instruction |
| Expression folding | Compile Multi-expressions<br>Identify common sub-exprs | | Scratch for temp results<br>Instr. to reference temp results |
| Data caching | | TMM: Cache update policy<br>TMM: Replace cached params with refs | API to update cache<br>Instr. to reference cache data |
| Function result caching | | TMM: Function result cache lookup/update | |

Fig. 11. Summary of hardware-software codesign. For UMDBMS column: SE = storage engine, QE = query engine, TMM = TM Manager.

simplifications. We ran a mix of the `NewOrder`, `OrderStatus` and `Payment` transactions with the same relative distribution as the original benchmark. These transactions account for nearly 90% of the transactions in the original benchmark and have stringent response time and consistency requirements. We assume that the data is completely cached in memory and as a result we varied the parameter W (the number of warehouses) from 10 to 100. We used a driver that runs a number of workers continuously issuing transactions. The throughput was measured by averaging multiple one-minute runs with checkpointing turned off.

We ran experiments varying the workers (N) from 10 to 500 and we pick a particular data point (W=10, N = 100) to discuss our results—these numbers were fairly representational of the results for other parameters and give a good sense of how Cipherbase performs for transactional workloads. We defer a more thorough evaluation including examining sensitivity to different parameters such as I/O, and data skew to future work.

### D. Experiment 1: TPC-C Throughput

Figure 12 shows Cipherbase throughput for the three encryption schemes, normalized to SQL Server plaintext throughput. (We show normalized throughput since Microsoft policy does not allow publication of absolute numbers.) When all optimizations turned on, Cipherbase achieves 90% of the plaintext throughput for the *Customer* policy and around 80% for *Strong/Weak*. When all the columns are strongly encrypted and every data operation needs to go through the TM, Cipherbase achieves 40% of plaintext throughput. Given that we are comparing with an industrial strength high-performance transactional system, this performance is competitive and should be sufficient for most applications. This experiment illustrates that Cipherbase can provide robust performance even for the strongest encryption policy. This points to the utility of having in-server secure hardware.

Figure 12 also suggests that the optimizations in Section V are important; they provide about 33% throughput improvement for *Customer*, about 50% improvement for *Strong/Weak*, and about a factor 2 improvement for *Strong/Strong*. Figure 13 shows the number of distinct roundtrips and the number of bytes sent to the TM for each policy. We observe that (1) as expected, the number of round trips and amount of data sent increases with the strength of the encryption policy;

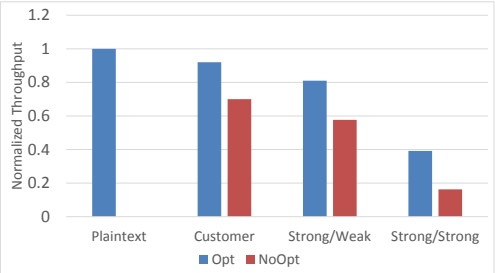


Fig. 12. Normalized TPCC Throughput (W=10)

| | Round Trips Per Xact | Bytes Transferred Per Xact |
|---|---|---|
| Customer (No Opt) | 13.96 | 898.54 |
| Customer (Opt) | 11.76 | 842.2 |
| Strong/Weak (No Opt) | 57.80 | 2248.12 |
| Strong/Weak (Opt) | 6.68 | 1066.95 |
| Strong/Strong (No Opt) | 522.69 | 19501.21 |
| Strong/Strong (Opt) | 89.05 | 122176.3 |

Fig. 13. Number of Round-Trips and Bytes To TM

(2) our optimizations considerably reduce the number of round trips for *Strong/Weak* and *Strong/Strong*; and (3) while our optimizations reduce the number of bytes sent for *Customer* and *Strong/Weak*, they increase the data sent for *Strong/Strong*, since we used vectorization to navigate B-Trees. However, even though a larger number of bytes is sent, it still yields significant benefits in terms of throughput since reducing the number of round-trips is more important. We omit the detailed analysis of the contribution of each optimization technique. Briefly, the interaction of the techniques is nuanced, they are all important and only the combination of optimizations results in these improvements.

### E. Experiment 2: Bottleneck Analysis

Cipherbase has three potential bottlenecks: (a) data contention from concurrent accesses to shared data, (b) the bandwidth of the PCIe bus, and (c) the computational power of the trusted hardware. Figure 15(a) shows the utilization of the PCIe bus. While the theoretical PCIe bandwidth of our machine is 2GB per second, the maximum PCIe utilization for any policy was less than 9%. Likewise, the utilization of the FPGA stack machines was less than 10%.

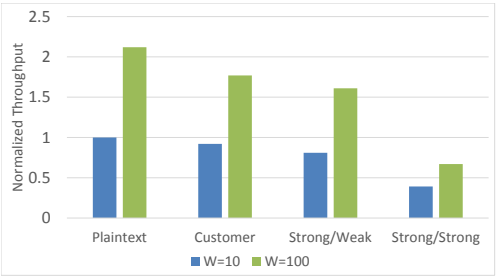


Fig. 14. TPC-C: Varying Number of Warehouses. Throughput normalized with respect to plaintext at 10 warehouses

| | Bytes/sec |
|---|---|
| Customer | 3.5 MB |
| Strong/Weak | 3.2 MB |
| Strong/Strong | 169.57 MB |

($a$) PCIe Utilization

| | Median | 90th Percentile |
|---|---|---|
| Plaintext | 7 | 29 |
| Customer | 8 | 30 |
| Strong/Weak | 11 | 31 |
| Strong/Strong | 51 | 151 |

($b$) TPCC Latency (msec)

Fig. 15.

This leaves data contention to be the key bottleneck, which was confirmed by an additional experiment shown in Figure 14. Scaling the benchmark from 10 to 100 warehouses has the primary characteristic of reducing contention on the *item* hotspot and we observe that the throughput of Cipherbase increases across all policies. Thus, we intend to further explore the interaction between the latency of accessing the TM and data contention.

### F. Experiment 3: TPC-C Latency

Figure 15(b) shows the latency of the TPC-C transactions, measured over all transactions. It shows the median and 90th percentile latency. While the latency of Cipherbase increases with the strength of encryption, it becomes clear that even with the *Strong/Strong* database, the end to end latency is reasonable (less than 200 msecs). The end-to-end latency requirement of the TPC-C benchmark is 5 seconds for the 90th percentile [30], an SLA easily met by even the *Strong/Strong* database.

### G. Prior work on OLTP workloads

All three encryption policies we use involve operations over strongly encrypted data. As such, they cannot be supported by pure PHE-based systems such as CryptDB. Specifically, the TPC-C benchmark involves operations such as both additions and multiplications on the same column data[5], string concatenations, and substring operations; current practical PHE schemes do not support these operations. This implies that CryptDB cannot support, without reducing data confidentiality, a stored-procedure based implementation of TPCC that involves a single client-server roundtrip per transaction. These limitations could be overcome by an implementation that makes one client roundtrip for each SQL statement in a transaction[6]. However, it is accepted wisdom that a stored-procedure implementation is required for high-performance; the TPC-C results of major vendors use stored procedures [30]. Even with this multi-roundtrip implementation, the performance for *Strong/Strong* falls off the cliff, since this policy precludes the use of indexes in the server. TrustedDB has no PHE or trusted client-related limitations. However, TrustedDB seems to be optimized for OLAP performance, not OLTP. Achieving high transaction throughput in TrustedDB, where two different database systems must be able to concurrently update a database page, seems non-trivial and no relevant details for this capability are provided in [5].

## VII. Discussion

### A. Operational Data Confidentiality and Beyond

Cipherbase provides operational data confidentiality for a strong adversary. We note that such a guarantee is usable and practical for OLTP systems for a variety of reasons. First, it is unrealistic to assume that an adversary can constantly monitor/persist contents of the server's disk/memory for highly concurrent OLTP systems. Thus, in practice an adversary is likely to lie in between the weak and strong adversaries as discussed in Section II. Second, weaker encryption schemes can in reality leak less information than assumed based on data characteristics (e.g., using DET encryption for a ID column with a UNIQUE constraint). Third, offering higher security for OLTP systems (where support for indexing is required) may incur significant overheads. For instance, ORAM [31] and other proposals that do data shuffling [32] incur writes even for a read only transaction which should seriously affect concurrency and overall performance. Such data shuffling also reduces spatial and temporal locality of reference which further undermines performance. In fact, a state-of-the-art system that uses ORAM reports a 32x slowdown [33] in memory access performance. In contrast, Cipherbase can offer a 2x slowdown for a policy in which all columns are strongly encrypted for TPC-C. Thus, we believe that Cipherbase offers a compelling design point for OLTP on encrypted data.

That said, security beyond operational guarantees may be feasible for other workload patterns, notably OLAP queries. There is prior work [8], [32] that explores security beyond operational data confidentiality for OLAP queries while not relying on building blocks such as ORAM. Interestingly, such schemes also require secure hardware — understanding and quantifying their security-performance tradeoffs is an interesting avenue for future work.

### B. Impact of Encryption on Space

Encryption is known to increase the space consumed by the database considerably. For instance, using the classic AES-CBC (Cipher-Block Chaining) mode of non-deterministic encryption expands a 32-bit plaintext to 256 bits of ciphertext.

[5]Any encryption scheme that supports both additions and multiplications is fully-homomorphic [10].

[6]A `NewOrder` transaction has 30+ such statement instances.

| Encryption Policy | Space Consumed |
|---|---|
| Plaintext | 836 MB |
| Strong/Strong (CBC) | 3634 MB |
| Strong/Strong (CTR) | 1427 MB |
| Strong/Strong (CTR) Compressed | 1082 MB |

Fig. 16. Space Consumption

In Cipherbase, we use the AES-CTR (Counter) mode of non-deterministic encryption (while also offering the CBC mode to clients, should they so choose.) The CTR mode is also a standard mode of AES and offers the IND-CPA guarantee alluded to earlier. Further, it offers significant benefits in space consumption. The intuition behind the space savings is as follows. First, the CBC mode is a *block* cipher requiring plaintext to be padded to multiples of a fixed block size (128 bits). In contrast, the CTR mode is a *stream* cipher and therefore does not pad the plaintext. Second, non-deterministic encryption is realized by incorporating an Initialization Vector (IV) that changes with every input into the encryption function. The CBC mode requires the IV to be randomly generated. In contrast, CTR mode only requires the IV to be unique (nonce). In Cipherbase, we use a 64-bit counter maintained by the TM as the IV. The counter is incremented with every encryption. Since the IV is sequential, it lends itself to effective compression even using SQL Server's off-the-shelf compression techniques. Note that this has no impact on the confidentiality - the compression is independent of the actual data values. Table 16 reports the space consumed by the Plaintext and Strong/Strong databases with CBC and CTR modes with and without compression for the TPC-C database (with W=10). As we can see, the CTR mode with compression consumes only 30% more space than plaintext. In contrast, CBC consumes more than 4x the space consumed by plaintext. We believe these savings are significant since: (1) clients pay for space usage in a cloud setting, and (2) the assumption that large databases fit in main memory may not hold in a multi-tenant environment, hence space savings could potentially translate to reduced IO and thereby performance improvement.

## VIII. Related Work

A comprehensive overview of alternative ways to implement confidentiality in a database system is provided in a recent tutorial [9]. In general, the approaches differ depending on the kind of attack they protect against, the encryption techniques supported, and whether they rely on hardware.

Commercially the most established technique is to keep all data encrypted on disk and to decrypt as soon as it is loaded into main memory for processing. Both Oracle [7] and Microsoft [6] have adopted this approach as part of their database products. While this approach protects the confidentiality of data in the event that disks are stolen, this approach is not sufficient to protect the confidentiality of the data against attacks from database administrators who have root access to machines. One fundamental problem of this approach is that the encryption key is stored as part of the database and is, thus, easily accessible for a database administrator. Amazon CloudHSM [34] addresses this problem by leveraging trusted hardware (Hardware Security Modules) to securely store keys. However, even if the key is not accessible and the administrator has no permissions to execute queries, the administrator can retrieve confidential data by taking a memory dump of the machine. To protect data from "honest and curious" database administrators, data must be kept encrypted on disk and in main memory. The most relevant related systems are CryptDB [3], Monomi [4] and TrustedDB [5]. These systems were covered when we dicussed prior work in Section I-A.

There has been prior work on leveraging FPGAs for query processing [35]. The main focus has been on using FPGAs as accelerators for specific database operators and certain database operators (e.g., streaming aggregates) are run in the FPGA. The key differences in our approach are: 1) encryption as a novel use case for leveraging FPGAs for query processing and 2) the use of the FPGA as a co-processor which runs only expression evaluation (query operators still run in the UM). Similar remarks apply to related work on using GPUs to accelerate query processing [36], [37].

The Cipherbase architecture and the basic co-processor design (without optimizations) have been previously outlined in our prior work [38], [28]. The main contribution of this paper is novel optimization techniques for OLTP workloads and a comprehensive experimental evaluation and analysis for the TPC-C benchmark. The function result caching optimization is related to caching techniques used for expensive predicates [39]. In some sense, TM calls are like expensive predicates so that all query processing techniques with expensive predicates are potentially applicable and some of the query optimization issues discussed in [39] are likely to be relevant for analytical queries.

## IX. Conclusions

In summary, we presented the design and evaluation of Cipherbase for OLTP workloads. We believe Cipherbase offers a unique architecture for building encrypted databases with: 1) a secure hardware based solution with a minimal TCB, 2) industrial strength transactional functionality with competitive performance, even when all data stays strongly encrypted throughout the entire stack (disk, memory and network), and 3) an interesting design point in the performance-security continuum that provides operational guarantees with the flexibility to choose an appropriate encryption policy. Understanding the performance characteristics of Cipherbase on new secure hardware platforms (such as SGX), examining the impact of OLAP workloads and studying if we can provide practical higher security beyond operational guarantees remain interesting avenues of future work.

## References

[1] (2013) The notorious nine: Cloud computing top threats in 2013. Cloud Security Alliance. https://cloudsecurityalliance.org/download/the-notorious-nine-cloud-computing-top-threats-in-2013/.

[2] H. Hacigümüs, B. R. Iyer, C. Li *et al.*, "Executing sql over encrypted data in the database-service-provider model," in *SIGMOD*, 2002.
[3] R. A. Popa, C. M. S. Redfield, N. Zeldovich *et al.*, "CryptDB: protecting confidentiality with encrypted query processing," in *SOSP*, 2011.
[4] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," *PVLDB*, vol. 6, no. 5, 2013.
[5] S. Bajaj and R. Sion, "TrustedDB: a trusted hardware based database with privacy and data confidentiality," in *SIGMOD*, 2011.
[6] Microsoft Corporation, "SQL Server Encryption," http://technet.microsoft.com/en-us/library/bb510663.aspx.
[7] Oracle Corporation, "Transparent Data Encryption," http://www.oracle.com/technetwork/database/options/advanced-security/index-099011.html.
[8] A. Arasu and R. Kaushik, "Oblivious query processing," in *ICDT*, 2014.
[9] A. Arasu, K. Eguro, R. Kaushik, and R. Ramamurthy, "Querying encrypted data," in *SIGMOD*, 2014, (tutorial).
[10] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009, crypto.stanford.edu/craig.
[11] K. Eguro, K. Rajan *et al.*, "Automatic secure partitioning for database applications," Microsoft Research, Tech. Rep. MSR-TR-2013-74, 2013.
[12] SQLite. http://www.sqlite.org/.
[13] (2011) The netezza data appliance architecture: A platform for high performance data warehousing and analytics. http://www.redbooks.ibm.com/redpapers/pdfs/redp4725.pdf.
[14] H. Lipmaa, P. Rogaway, and D. Wagner, "CTR-mode encryption," http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/ctr/ctr-spec.pdf.
[15] M. Bellare, P. Rogaway, and T. Spies, "The FFX mode of operation for format-preserving encryption."
[16] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill, "Order-preserving symmetric encryption," in *EUROCRYPT*, 2009, pp. 224–241.
[17] R. A. Popa, F. H. Li, and N. Zeldovich, "An ideal-security protocol for order-preserving encoding," in *IEEE Symposium on Security and Privacy*, 2013, pp. 463–477.
[18] B. W. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: Theory and practice," *ACM Trans. Comput. Syst.*, vol. 10, no. 4, pp. 265–310, 1992.
[19] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth, "Reducing tcb complexity for security-sensitive applications: three case studies," in *EuroSys*, 2006, pp. 161–174.
[20] D. B. Lomet, "Key range locking strategies for improved concurrency," in *VLDB*, 1993, pp. 655–664.
[21] D. B. Lomet, K. Tzoumas, and M. J. Zwilling, "Implementing performance competitive logical recovery," *PVLDB*, vol. 4, no. 7, 2011.
[22] S. Drimer, "Volatile FPGA design security—a survey," 2008, http://www.saardrimer.com/sd410/papers/fpga_security.pdf.
[23] K. Eguro and R. Venkatesan, "FPGAs for trusted cloud computing," in *FPL*, 2012, pp. 63–70.
[24] F. McKeen, I. Alexandrovich, A. Berenzon *et al.*, "Innovative instructions and software model for isolated execution," in *HASP*, 2013.
[25] A. Putnam, A. Caulfield, E. Chung *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *ISCA*, 2014.
[26] R. Müller, J. Teubner, and G. Alonso, "Data processing on fpgas," *PVLDB*, vol. 2, no. 1, pp. 910–921, 2009.
[27] S. Harizopoulos, D. J. Abadi, S. Madden *et al.*, "OLTP through the looking glass, and what we found there," in *SIGMOD Conference*, 2008.
[28] A. Arasu, K. Eguro, R. Kaushik *et al.*, "A secure coprocessor for database applications," in *FPL*, 2013.
[29] A. Narayanan and V. Shmatikov, "Myths and fallacies of "personally identifiable information"," *Commun. ACM*, vol. 53, no. 6, 2010.
[30] "The TPC-C Benchmark," `http://www.tpc.org`.
[31] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, no. 3, pp. 431–473, 1996.
[32] M. Kantarcioglu and C. Clifton, "Security issues in querying encrypted data," in *DBSec*, 2005, pp. 325–337.
[33] M. Maas, E. Love, E. Stefanov *et al.*, "PHANTOM: practical oblivious computation in a secure processor," in *CCS*, 2013.
[34] Amazon Corporation, "Amazon Cloud HSM Service," http://aws.amazon.com/cloudhsm/.
[35] R. Müller, J. Teubner, and G. Alonso, "Data processing on FPGAs," *PVLDB*, vol. 2, no. 1, pp. 910–921, 2009.
[36] S. Breß, "Why it is time for a HyPE: A hybrid query processing engine for efficient GPU coprocessing in DBMS," *PVLDB*, vol. 6, no. 12, 2013.
[37] C. Kim, J. Chhugani *et al.*, "FAST: Fast architecture sensitive tree search on modern CPUs and GPUs," in *SIGMOD*, 2010.
[38] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan, "Orthogonal security with Cipherbase," in *CIDR*, 2013.
[39] J. M. Hellerstein and J. F. Naughton, "Query execution techniques for caching expensive methods," in *SIGMOD*, 1996.

## APPENDIX

### A. Range Indexes vs OPE

As we explained in Section II-B, range indexes provide the same confidentiality guarantee as OPE for a weak adversary. However, there is an important difference for non-clustered indexes which the following example clarifies.

*Example A.1:* Consider a table T (A, B) and assume that there is a clustered index on column A and a non-clustered index on column B. In SQLServer, the non-clustered index stores the corresponding key value of the clustered index as a RID column. As previously mentioned, the clustered index on A does leak the full ordering of the A column. However, the A values stored in the non-clustered index can be re-encrypted to avoid revealing the corresponding correlation with the clustered index. This cannot be done in a system without secure hardware and any software based OPE scheme will in addition reveal the correlations between the non-clustered and clustered indexes.

Of course, OPE based implementations offer the important advantage that they can avoid roundtrips to hardware. As the results in the experimental section show, using deterministic encryption for the ID columns can lead to improved performance when compared to using equality indexes. A more thorough evaluation of OPE as an optimization (range predicates are more commonly used in OLAP workloads) is deferred to future work.

### B. FPGA utilization

| | LUTs | FF | BRAM | DSP48 |
|---|---|---|---|---|
| Full System | 59788 (39.7%) | 35909(11.9%) | 314(75.5%) | 40 (5.2%) |
| PCIe Infrast. | 46388 (30.8%) | 21068(7.0%) | 230(55.3%) | 0 (0.0%) |
| 4x DB Proc | 13400(8.9%) | 14840(4.9%) | 84(20.2%) | 40 (5.2%) |

Fig. 17. FPGA Resource Requirements

The FPGA resource requirements for the four-core TM used in our experiments are shown in Figure 17. The number of look-up tables, flip-flops, and multipliers (LUTs, FFs, and DSP48s are fundamental logic elements) required for all four cores is less than 8.9% of those available on the device. The most heavily utilized element is block memory (BRAM), with the four TMs requiring 20.2% of the available memory resources. The PCIe controller represented approximately two-thirds of the overall logic and memory requirements. The low logic requirements of this system and the fact that the selected FPGA is now several years old and has been superceeded by more capable devices suggest that there is sufficient headroom to expand the functionality of the system in the future.

The TMs relatively modest requirements indicate that although our prototype does not currently support all SQL

operators and data types, that there is sufficient headroom to expand the functionality of the system in the future. The XC6VLX240T that was used is a low-to-mid tier device that is now several years old. We expect that a more modern FPGA that would nominally affect the initial price of the complete platform (servers plus FPGAs, purchased in volume) would contain more than double the logic and memory capabilities.

### C. Function Result Caching

The following example illustrates how function caching can help optimize loop counters.

*Example A.2:* Consider the following stored procedure fragment. The code inserts a set of values into a table and one of the columns in the table is updated to the value of the loop variable (@li_no).

```
DECLARE @li_no int = 0

WHILE(...)
{

SELECT @li_no = @li_no + 1
INSERT INTO OrderLines (...,  @li_no, ...)

}
```

If the corresponding column is encrypted then the loop variable also needs to be encrypted to ensure that the stored procedure typechecks. The rewritten stored procedure is shown below; the addition on the encrypted data is performed by calling TMEval with the appropriate program (the program handle is 5). For this version of the stored procedure, each addition in the while loop requires a roundtrip to the TM.

```
DECLARE @li_no AESCBCInt = 0x22e3cf29

WHILE(...)
{

SELECT @li_no = TMEval(5, @li_no , 0x33cb7cc)
INSERT INTO OrderLines(...,  @li_no, ...)

}
```

Note that the stored procedure logic is identical for every invocation (including the initial constants). If we cache the result of addition for each encrypted input for the first transaction, then every subsequent invocation of the stored procedure that runs through the same logic can first check the cache and completely skip the rountrip to TM if there is a hit.

### D. Query Plans vs Statements

While discussing query evaluation (e.g., in section III-C), we have shown the corresponding rewritten query (augmented with TMEval calls) that is sent to the server. This is mainly for clarity of presentation - we note that the cipherbase client actually sends a query plan to the server. The main advantages in using a query plan based API are: 1) The security offered by the system is operational and thus, a function of the query plan. This API enables the user to pick a plan that is appropriate. 2) This simplifies query optimization which occurs at the client using statistics *on plaintext*. Query optimization was not a focus in this paper since we were optimizing OLTP workloads where index-based plans are usually the "optimal" choice.