Encrypted Databases: From Theory to Systems

Zheguang Zhao[*] Brown University
Seny Kamara[†] Brown University
Tarik Moataz[‡] Aroki Systems
Stan Zdonik[§] Brown University

ABSTRACT

End-to-end encrypted relational database management systems (EDBs) are the "holy grail" of database security and have been studied by the research community for the last 20 years. During this time, several systems have been proposed with a variety of limitations that include weak security, poor performance and restricted query expressiveness. We contend that this state of affairs is due, in part, to a lack of cohesion between the techniques and methodologies of the database and cryptography communities. We believe that the only way to make significant progress on this important problem is to properly leverage techniques and ideas from the two communities.

Towards this end, we identify five key design principles for end-to-end encrypted relational databases. These principles include security, performance and expressiveness considerations. We describe a system called `KafeDB`; the first encrypted relational database system that meets these principles. `KafeDB` is based on a new database encryption scheme called OPX that supports a nontrivial subset of SQL queries. Our prototype, built on top of `PostgreSQL`, shows the feasibility of our approach. TPC-H shows our prototype is about one to three order of magnitude slower than optimized plaintext `PostgreSQL` and requires about one order of magnitude more storage while offering end-to-end security.

1. INTRODUCTION

As we produce and consume increasing amounts of data, we are witnessing several conflicting trends. On one hand, these datasets are becoming more intrusive and privacy-sensitive and, on the other, they are becoming harder to secure. In fact, the constant occurrence of data breaches points to the fact that the data management systems produced over the last 40 years are not capable of adequately protecting the data they store. While systems sometimes encrypt data in transit and at rest, there are many stages in the data lifecycle where it remains unencrypted, especially when in use.

An alternative way of deploying encryption is *end-to-end* encryption. In this approach, the data is encrypted by the user before it even leaves its device. End-to-end encrypted systems and services provide much stronger security and privacy guarantees than the current generation of systems. The main challenge in building such systems, however, is that end-to-end encryption breaks many of the applications and services we rely on, including cloud computing, analytics, spam filtering, databases and search. The area of *encrypted systems* aims to address the challenges posed by end-to-end encryption by producing practical systems that can operate on end-to-end encrypted data.

Encrypted databases. A key problem in this area is the problem of designing *encrypted databases* (EDB); that is, of building practical database management systems (DBMS) that can operate on end-to-end encrypted databases.

The problem of encrypting relational databases is one of the "holy grails" of database security. It was first explicitly considered by Hacigümüs, Iyer, Li and Mehrotra [16] who described a quantization-based approach which leaks the range within which an item falls. In [23], Popa, Redfield, Zeldovich and Balakrishnan described a system called `CryptDB` that can support a non-trivial subset of SQL without quantization. `CryptDB` achieves this in part by making use of property-preserving encryption (PPE) schemes like deterministic and order-revealing (ORE) encryption [1, 6, 8], which reveal equality and order, respectively. Because `CryptDB`'s PPE-based approach was efficient and legacy-friendly, it was quickly adopted by academic systems like Cipherbase [3] and commercial systems like SEEED [24], and Microsoft's SQL Server Always Encrypted [12].

Unfortunately, this design paradigm was never formally analyzed (e.g., using standard methods from cryptography) or subject to any cryptanalytic evaluation. As a result, in 2015, Naveed, Kamara and Wright showed that PPE-based EDBs could be attacked in practice with very high success rates in the weakest possible adversarial model [21]. In the setting of electronic medical records, for example, sensitive attributes of up to 99% of patients could be recovered with a snapshot attack (i.e., without even seeing any queries). Since then, several follow-up works [15, 14] have improved on the original NKW attacks.

General-purpose approaches. Besides using PPE, there are several ways to design secure relational EDBs but each solution achieves trade-offs between efficiency, query expressiveness and leakage. General-purpose primitives like fully-homomorphic encryption (FHE) and secure multi-party computation (MPC) can be used to support all of SQL without any leakage but at the cost of exceedingly slow query execution due to linear-time asymptotic complexity and very large constants. Similarly, oblivious RAM (ORAM) can also be used to handle all of SQL with very little leakage (i.e., mostly volume leakage) but at the cost of a poly-logarithmic multiplicative overhead in the size of the database.

Trusted hardware. Another approach to designing relational EDBs is to use trusted hardware such as se-

[*]zheguang.zhao@alumni.brown.edu
[†]seny@brown.edu
[‡]tarik@aroki.com
[§]sbz@cs.brown.edu

11th Annual Conference on Innovative Data Systems Research (CIDR '21) January 10-13, 2021, Chaminade, USA.

cure coprocessors or Intel SGX. Several systems, most notably TrustedDB [4] and StealthDB [28] take this direction. Though our system could leverage trusted hardware by running our client proxy in an enclave[1], we do not investigate this direction given the security concerns around SGX.

Towards a new generation of EDBs. Given the high level of interest in encrypted database technology from academia, industry and government; the inherent weaknesses of quantization- and PPE-based solutions; and the impracticality of general-purpose EDBs, the design of practical and secure encrypted database systems remains an important open problem. A first step towards achieving this was taken by systems such as `ESPADA` [10], `Blind Seer` [22] and `Stealth` [17] which, roughly speaking, use structured encryption (STE) to index the columns of a database. While these systems achieve much better security than the quantization- and PPE-based solutions, they can only support a very restricted subset of SQL; specifically, filter operations and no joins or projections. In this work, we tackle the key challenges that impede the development and deployment of encrypted databases. Our contributions can be summarized as follows:

- *(design principles)* we identify and discuss five key principles for the design of practical and secure relational EDBs. These include a reasonable leakage profile, efficiency, legacy friendliness, optimization friendliness and expressiveness. Achieving any strict subset of these four requirements is insufficient.

- *(construction)* we describe the first encrypted database scheme that follows all the design principles outlined above. To achieve this we make two important technical contributions: (1) we show, for the first time, how to design *optimization-friendly* STE schemes; and (2) we introduce a new technique called *emulation* that makes STE-based solutions legacy-friendly. This new scheme is called OPX and is an extension of the SPX construction of Kamara and Moataz [18].

- *(architecture)* we propose an architecture for encrypted database management systems that integrates the needed cryptographic components into a traditional DBMS architecture. This is, in part, done by introducing a *crypto engine* that is responsible for providing end-to-end encryption and an *emulation engine* that is responsible for making the encrypted databases and queries "comprehensible" to a standard and unmodified DBMS.

- *(prototype)* we describe the implementation and evaluation of a new system called `KafeDB` based on our architecture, our OPX construction and our emulators. `KafeDB` runs on top of an *unmodified* `PostgreSQL` server. Our initial prototype demonstrates the feasibility of our architecture and approach. We evaluate its performance empirically using the `TPC-H` benchmark and report promising initial results: about an order of magnitude query and storage overhead over standard `PostgreSQL`, but offering considerably stronger security guarantees than `CryptDB` and much more expressiveness than `ESPADA`, `Blind Seer` and `Stealth`. To improve upon the initial results, we will need more sophisticated techniques in both cryptography and database systems.

[1]SGX currently allows 90MB of working memory and our proxy is around $350KB$ in size.

2. RELATED WORK

We already discussed related work on PPE-based and STE-based relational encrypted databases so we focus here on work in encrypted search and on other types of EDBs.

Encrypted search. Encrypted search is the area in cryptography that focuses on the design, cryptanalysis and implementation of protocols and systems that support search on encrypted data. Encrypted search was first considered explicitly by Song, Wagner and Perrig in [25] which introduced the notion of searchable symmetric encryption (SSE). Curtmola et al. introduced and formulated the notion of adaptive semantic security for SSE [13] together with the first sub-linear and optimal-time constructions. Chase and Kamara introduced the notion of structured encryption which generalizes SSE to arbitrary data structures [11].

Federated EDBs. Federated EDBs are systems that are composed of multiple autonomous encrypted databases. Most federated EDBs use secure multi-party computation (MPC) to query the constituent EDBs securely. In this model, multiple parties hold a piece of the database (either tables or rows) and a public query is executed in such a way that no information about the database is revealed beyond what can be inferred from the result and some additional leakage. Examples include SMCQL [5] and Conclave [29], which store the databases as secret shares and encryptions, respectively, and use MPC to execute the sensitive parts of a SQL query on the shared/encrypted data. We note that standard EDBs like `KafeDB` can be combined with MPC to yield a federated EDB.

3. DESIGN PRINCIPLES

Designing a relational EDB is, arguably, the most challenging problem in encrypted search. Existing RDBMs are the result of over 40 years of research and development so competing with the performance of commercial DBMSs over encrypted data is a tall order. To achieve this level of performance, it stands to reason that EDBs need to inherit as many of these advances as possible. With this in mind, we outline five principles that are necessary for the design of practical EDBs.

Adversarial models & leakage. There are two main adversarial models considered in encrypted search: (1) *persistent adversaries* which have access to the encrypted database and can view all the query operations that are executed on it; and *snapshot* adversaries which only have access to the encrypted database. Persistent adversaries model attackers that corrupt the server and stay long enough to view some number of queries. Snapshot adversaries model attackers that corrupt the server and exfiltrate a snapshot of its memory and disk. All cryptographic solutions that support search on encrypted data in sub-linear time leak information against persistent adversaries. This is true of PPE-based, STE-based and ORAM-based solutions. However, it is known that both STE and ORAM can lead to solutions with no leakage against snapshot adversaries [2].

The security of an encrypted search solution is characterized by its *leakage profile* which is a formal description of the information an adversary learns from observing and interacting with the scheme. More precisely, in the the persistent model, a leakage profile consists of: (1) *setup leakage* which describes the information the adversary learns by just observing the encrypted database; and (2) *query leakage* which describes the information the adversary learns by observing the execution of queries. For dynamic schemes, which support the addition and deletion of data, the leakage profile also includes *add leakage* and *delete leakage*. This approach to characterizing leakage was introduced in [13, 11] and we refer to [19] for additional details.

Principal #1: minimal leakage. An important design goal for any encrypted database is to minimize the information a persistent adversary is able to recover. At a minimum, this means that there should be no known practical attack against the scheme. Furthermore, the scheme's leakage profile should have the following characteristics:

- *(minimal setup leakage)* the setup leakage of the scheme should include at most the "shape" of the database; i.e., the number of columns and rows of each table.
- *(output-dependent query leakage)* when a query is executed, the adversary should, at most, learn information related to result of the query and not to the entire database or column. Furthermore, the information that is leaked should, at most, be statistical information about the query or result like whether a query has been queried in the past, or the number of rows that contain a similar value.

To be clear, leakage profiles with these characteristics are not provably immune to attacks. But, given our current understanding and the state-of-the-art results in cryptanalysis [7], these leakage profiles seem difficult to attack in practice. For more discussion about leakage attacks we refer the reader to [7] and the discussions and references therein. [2]

Principle #2: low asymptotic overhead. The system should be competitive with a standard plaintext DBMS with respect to query execution and storage. High performance imposes efficiency requirements on the system's database encryption scheme. Specifically, it should achieve the same asymptotic complexity as a plaintext database and preferably with small constants. As an example, schemes that add a linear or even polylogarithmic multiplicative overhead over a plaintext query are unacceptable in practice. The same applies for the scheme's round and storage complexities.

Principle #3: optimization friendliness. In addition to low asymptotic overhead, the underlying database encryption scheme should be *optimization-friendly* in the sense that it should support the execution of optimized query plans and, in particular, of plans that are optimized by commercial query optimizers.

Principle #4: rich query expressiveness. The system should support a non-trivial subset of SQL and, at a minimum, the class of *conjunctive* SQL queries (or the SPC algebra) which have the form:

$\texttt{SELECT}$ attributes $\texttt{FROM}$ tables
$\texttt{WHERE}\ \mathsf{att}_1 = a\ \texttt{AND}\ \ldots\ \texttt{AND}\ \mathsf{att}_2 = \mathsf{att}_3\ \texttt{AND}\ \ldots$

This requires being able to handle select, project and join operators.

[2]Since our current design and system does not yet handle range queries, we do not consider cryptanalytic work focused on encrypted range schemes.

Principle #5: legacy friendliness. While building an entirely new encrypted DBMS from the ground up is an interesting technical question, designing a scheme that can work on top of an existing, unmodified DBMSs is more appealing from a practical standpoint. If the resulting system is competitive with plaintext systems, achieves the required security and provides rich query expressiveness, there is almost no reason to build a new DBMS from scratch and lose over 40 years of advances in database research and engineering. Ideally, the design should be database-agnostic in the sense that it should not depend on a particular DBMS.

4. SYSTEM OVERVIEW

`KafeDB` has a three-tier architecture composed of the application, the client and the server, as shown in Figure 1. Both the application and the client are assumed to operate in a trusted environment, whereas the server is untrusted. The client encrypts the application's database and queries and sends them to the server who executes them. The key material is stored by the client so the server never sees the data or queries in plaintext. Note that the client is stateless and only stores the schema of the plaintext database and the cryptographic keys.

The client is central to the `KafeDB` architecture, and most of its modules have to be carefully designed such that any given standard relational database can be used on the server. The most important modules are the *crypto engine* and the *emulation engine*, which are used throughout the data management cycle for data setup, query optimization and execution.

Crypto engine. In `KafeDB`, end-to-end encryption is handled by a crypto engine that implements the database encryption scheme. It is responsible for encrypting the database and queries and for decrypting the results. Currently, `KafeDB`'s crypto engine implements our OPX construction but future versions could be based on new and improved schemes.

Emulation engine. Once a database or query is encrypted it is handed to the emulation engine which is responsible for transforming them into relational tables and SQL queries to be processed by the server. Note that the tables and SQL queries output by the emulation engine *are not* the same as the tables and SQL queries produced by the application. In fact they are completely different since the latter are representations of the OPX-encrypted tables and queries of the application. Again, `KafeDB`'s emulation engine currently implements a specific emulator designed for this work but it could be replaced in the future with a different emulator.

Setup. Our current focus is on analytical workloads, therefore we design `KafeDB` to bulk load new data through the setup module at the client[3]. The setup module invokes the crypto engine to encrypt the data into encrypted structures, and then it uses the emulation engine to reshape them into tables and indexes.

Query optimizer. Due to encryption, the `KafeDB` server cannot maintain statistics over the tables and is, therefore, limited in how much it can optimize queries. In fact, since its underlying database encryption scheme achieves minimal setup leakage, the only information the server learns at setup time is the size of the database. Because of this, `KafeDB` does

[3]We defer the extension on secure fine-grained updates that are ACID-compliant to follow-up work.

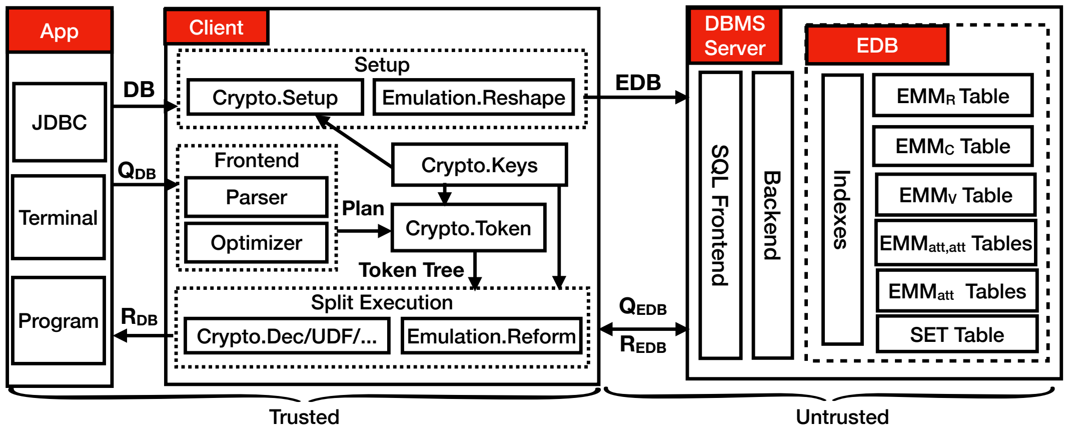

Figure 1: The KafeDB system architecture.

most of its query optimization at the client. One of the main technical contributions of OPX is its ability to support any SPC query plan. More precisely, the encrypted structures and query protocols used by OPX are carefully designed so that the supported operations (i.e., joins, filters, projections) can be queried in any order. This flexibility results in KafeDB being optimization-friendly since it can process query plans that are produced by standard query optimizers. Besides filter pushdown, we also identified two additional optimizations that are particularly effective in reducing some of the costs introduced by OPX. These include: *many-to-many join factorization* and *multi-way join flattening*. The former transforms each many-to-many join into two many-to-one joins (this requires an additional encrypted table at setup) and the latter transforms a sequence of multiple joins (i.e. a left deep tree) into separate pairs of joins (i.e. a bushy tree).

Split execution. KafeDB's OPX-based crypto engine currently handles conjunctive SQL queries on encrypted data. For more complex queries, we use split execution as introduced in [27]. Given a query, the client splits it into two kinds of subqueries: conjunctive subqueries, which are supported by the OPX crypto engine, and other subqueries which are not. The conjunctive subqueries are processed by the crypto engine and the others are executed locally using the results of the conjunctive subqueries.

5. THE OPX SCHEME

In this section, we give an intuitive description of OPX and of our emulation techniques, and refer to [20] for the formal treatment and analysis. First, however, we provide some technical background that is necessary to understand our construction.

5.1 Technical Background

OPX is a relational database encryption scheme. It is based on, and considerably improves on, the SPX construction of Kamara and Moataz [18]. The most important difference between the two schemes is that OPX is optimization-friendly whereas SPX is not.

Structured encryption (STE). A structured encryption scheme encrypts a data structure in such a way that it can be privately queried. Intuitively, an STE scheme is secure if the encrypted structure reveals nothing about the data structure beyond a given setup leakage and the query operation reveals nothing about the structure and/or query beyond a given query leakage. The setup and query leakage of a scheme constitute its *leakage profile*.

Encrypted multi-maps. A central building block in the design of most STE-based solutions are encrypted multi-maps. Briefly, an encrypted multi-map stores label/tuple pairs (where the tuples can be of different sizes) and supports Get and Put operations. OPX makes black-box use of EMMs which means that they can be instantiated with using any concrete construction. In our implementation, we make use of a variant of the Pibase scheme of Cash et al. [9] which has optimal storage and query complexity. Its setup leakage is the size of all the tuples of the multi-map, and its query leakage consists of the query equality pattern and the response identity pattern. The query equality reveals if and when the same query was made in the past, whereas the response identity reveals the response of the query [4]

SPX. Kamara and Moataz proposed the first STE-based database encryption solution to handle a non-trivial fraction of SQL [18]. This scheme, called SPX, has complexity linear in the query output size and (provably) leaks considerably less than known PPE-based solutions like CryptDB and Monomi. Roughly speaking, the scheme represents of the database using various multi-maps that are then encrypted with a multi-map encryption scheme. The main limitation of SPX, however, is that it can only support queries in a specific normal form. That is, all SPC queries have to be written in the SPC normal form and then translated into a form called the heuristic normal form. Due to this restriction, SPX cannot handle many optimized query trees (e.g., ones that result from filter pushdown or filter/join reordering).

5.2 Details of the OPX Scheme

We divide the scheme's description in two parts: (1) a setup phase during which the client outputs the encrypted database; and (2) a query phase during which the client sends an encrypted query. Here, we focus on the important ideas behind the scheme and refer the necessary formal treatment to [20].

Setup. The setup takes as input the plaintext database DB and a security parameter (i.e., the length of the cryp-

[4]It is trivial to make Pibase response-hiding but in the OPX construction it is used as a building block and the response-revealing variant is used by design in a manner that does not reveal the query responses.

tographic keys). It then creates six data structures that capture different representations of the database:

- *(row representation)* MM_R is a multi-map that maps each row identifier to a tuple composed of the cells of the row;
- *(column representation)* MM_C is a multi-map that maps each column identifier to a tuple composed of the cells of the the column;
- *(filter representation)* MM_V is a multi-map that maps the unique values in every column to the rows that contain that value;
- *(partial join representation)* $\{\mathsf{MM}_{\mathbf{c},\mathbf{c}'}\}_{\mathbf{c},\mathbf{c}'}$ is a set of multi-maps that correspond to a pair of *joinable* columns in the database. Each multi-map maps row identifiers in one column to the row identifiers in the other column that have equal cell value;
- *(full join representation)* $\{\mathsf{MM}_{\mathbf{c}}\}_{\mathbf{c}}$ is a set of multi-maps, each of which corresponds to a column $\mathbf{c}$ of the database. Each multi-map maps a column identifier $\mathbf{c}'$ to the pairs of row identifiers that have the same value in both $\mathbf{c}$ and $\mathbf{c}'$;
- *(partial filter representation)* SET is a set-membership structure that checks whether a cell in a specific row contains a specific value.

All the multi-maps are encrypted using a multi-map encryption scheme. The set structure is encrypted using a custom encrypted set scheme that we detail in the full version of this work. As mentioned, an important aspect of OPX is that it has the ability to make use of any combination of these structures to answer any conjunctive query. To do so, it leverages a key design technique in structured encryption called *structural chaining*. In its simplest form, it works as follows: the client sends an encrypted query that can only be used with one of the encrypted structure. Once the server executes this encrypted query, it reveals an intermediary response which is composed of other precomputed encrypted queries that were stored in the encrypted structure during setup. The server can then use these encrypted queries to query other encrypted structures. As a concrete example, in OPX, the client sends an encrypted SQL query to retrieve all rows that are equal to some specific value. The server will take this encrypted query and run it against EMM_V, the encrypted multi-map of MM_V, which outputs all the necessary encrypted queries to run against EMM_R.

Query. The query phase of OPX takes as input an optimized SQL query tree whose nodes are relational operators. OPX then replaces each node with a corresponding token for a specific encrypted structure. The emulation engine then emulates the token tree as an encrypted SQL query and sends it to the server. The server executes the encrypted query leveraging the structural chaining discussed above.

Efficiency. OPX's query complexity can be shown to asymptotically match the plaintext's when measured in terms of query output size. Furthermore, it is the first STE-based scheme that can support query optimization such that its encrypted query can incur lower complexity at execution. Its storage overhead over plaintext comes from the creation of multiple (encrypted) database representations. While most of these representations do not significantly increase the asymptotic storage overhead, the partial and full join representations can incur a worst-case quadratic blowup in database size. We use an idea of query rewrite called many-to-many join factorization to circumvent this blowup, and our results in `TPC-H` shows significant improvement in both storage and query time in Section 6. At a high level, this new query rewrite rule factors a many-to-many join (e.g. foreign key to foreign key) into two many-to-one joins (e.g. foreign key to primary key) such that each only incurs linear complexity.

Security. Because OPX uses encrypted multi-maps extensively, its leakage profile depends the profile of its underlying encrypted multi-map constructions. In our current instantiation, OPX leaks a combination of query equality and response identity patterns. At a high level, it reveals frequency information on how the client accesses the database such as if and when the client sends the same query. The server can also learn which query touches which rows, as well as the rows touched rows that are common between different encrypted queries. Note, however, that OPX—and therefore `KafeDB`—*provably* leaks significantly less than PPE-based schemes and systems like `CryptDB` and `Monomi`. Furthermore, its leakage profile is not prone to any known practical attack. We refer the formal security proofs to [20].

5.3 Emulation

While STE-based solutions are efficient and more secure than PPE-based solutions they have an important limitation: they require a custom server and, therefore, are not legacy-friendly. To address this, we introduce a new technique called *emulation* that can make STE-based schemes like OPX legacy-friendly. While the notion of emulation is generally applicable, in this project we focused on designing *SQL emulators*; that is emulators to make OPX run on any unmodified relational RDBMS. The main advantages of our emulator are: (1) it does not impact OPX's efficiency; (2) it preserves its security; and (3) it is agnostic to the underlying relational DBMS. An emulator consists of two algorithms: a *reshape* algorithm and a *reform* algorithm.

Reshape. This algorithm transforms the encrypted database, which consists of a set of EMMs, into a set of relational tables. It relies on sub-emulators that transform the individual EMMs into tables. In our current `KafeDB` implementation, we use the Pibase EMM from [9] so our sub-emulator is designed for that particular construction. At a high level, the sub-emulators parse each EMM into a set of label/tuple pairs which it then inserts as a row into a table. It then creates a plaintext index on the (encrypted) label column. In Figure 1, we summarize all the generated tables.

Reform. This algorithm transforms the (encrypted) query tree into a normal SQL query. Here, we use Common Table Expressions (CTEs) to capture the recursive nature of Pibase EMM queries. We will provide details on our SQL emulators in follow-up work.

6. EVALUATION

We implemented `KafeDB` and compared its performance against `CryptDB` and (standard) `PostgreSQL`. We want to stress that the current version of `KafeDB` should be viewed as as a first step towards designing practical and secure EDBs and is the first system to achieve the five principles outlined in Section 3. Previous systems like `CryptDB` failed to achieve the first principle: it has non-trivial setup leakage and is

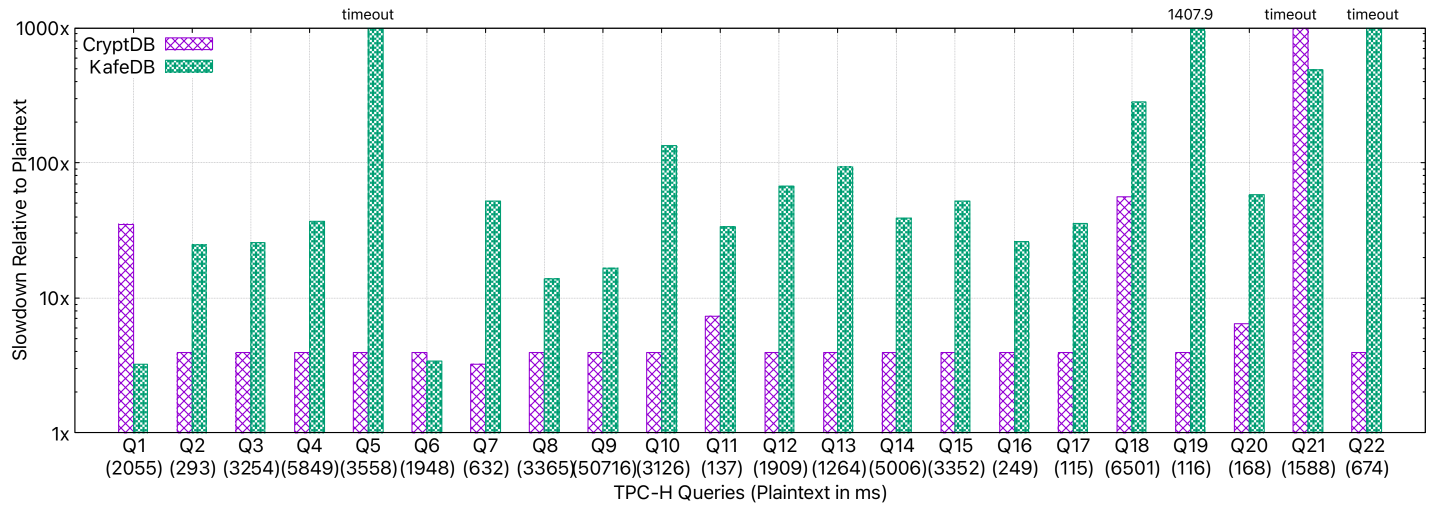

Figure 2: `TPC-H` Benchmark with scale factor 1.

vulnerable to a series of practical attacks. Other systems like `ESPADA`, `Blind Seer` and `Stealth` are not optimization-friendly or legacy-friendly and only support filters. The main purpose of our evaluation is to demonstrate that one can design an EDB systems without giving up completely on security, functionality and performance.

Implementation. The `KafeDB` client uses and extends `Spark SQL` 's Catalyst for query parsing and planning, and its executor to facilitate split execution over the `KafeDB` server which runs `PostgreSQL 9.6.2`. Our implementation is availalbe for download [26]. For the cryptographic primitives, we use `AES` in `CBC mode` with `PKCS7` padding for symmetric encryption, and `HMAC-SHA-256` for pseudo-random functions. Both primitives are provided by `Bouncy Castle 1.64` in the Client and by the `pgcrypto` module in `PostgreSQL 9.6.2`.[5]

Testing environment. We conducted our experiments on Amazon (EC2) with instance type `t2.2xlarge`, which has 8 CPUs, 32GB RAM and 1TB of Elastic Block Store. Following the typical hardware setting in the research literature, we chose to keep high memory capacity ratio to the database size, which amounts to 0.5× for `KafeDB` and 7.2× for plaintext `PostgreSQL`.

Data generation. We use the `TPC-H` benchmark with a scale factor 1, which leads to about 8.6 million rows and 1GB of data. We only index the filtered and joined attributes for `KafeDB` and plaintext `PostgreSQL`. This indexing strategy helps to ensure the best possible query performance for both `PostgreSQL` and `KafeDB`.

Comparisons. For the purpose of efficiency evaluation, we also compare `KafeDB` to `CryptDB` from [27]. The original version of `CryptDB` [23] only supports 4 out of the 22 `TPC-H` queries, so the results we recall here are from a modified version of `CryptDB` in [27] that supports the full `TPC-H`. The hardware setup differs slightly in [27] where most noticeably the authors used a machine with slightly less RAM of 24GB compared to the 32GB of RAM in our setting[6]. Since the code of [27] is not open-source, and in order to draw fair comparisons, we only report the query and storage multiplicative overheads incurred by these systems over a plaintext `PostgreSQL`.

Overview of our results. As an initial study, we provide below a summary of our `TPC-H` results for `KafeDB` with OPX scheme for scale factor one:

- `KafeDB` was about an order of magnitude slower than CryptDB. For `KafeDB`, excluding the three queries that timed out, the median slowdown relative to plaintext was 45.6× with a range of 3.2×-1407.9×; whereas for CryptDB the median was 3.92× with a range of 1.04×-55.9×;
- All queries performed better with encrypted query optimizations applied. With **`selection pushdown`**, the speedup varied from 4× to 53×. Without **`many-to-many join factorization`**, the single join between `Customer` and `Supplier` timed out after 24 hours, whereas the factorized joins with additionally `Nation` took only 12 minutes. With **`multi-way join flattening`**, the speedup was around 20×.
- `KafeDB` incurred an order of magnitude size blowup over plaintext due to both ciphertext expansion and the complexity of the encrypted structures with a multiplicative factor of 13.17×. CryptDB appears to incur a smaller 4.21× size blowup;
- `KafeDB` requires about an order of magnitude more time to set up than to load the plaintext into `PostgreSQL` with a multiplicative factor of 10.37×.
- At scale factor 10, `KafeDB` showed signs of limited scalability where the overhead for most queries exceeded three orders of magnitude compared to plaintext.

Note however that this initial prototype of `KafeDB` did not undergo system-level optimization, and the OPX scheme still has much room for improvement beyond the new support for query optimization. Here we focus on functionality and security at first, and defer more sophisticated improvement on the efficiency and security to the future work.

[5]The only secure mode provided by `PostgreSQL 9.6.2`.
[6]The authors in [27] stated their evaluation numbers were similar across different hardware setup.

7. CONCLUSION

The problem of building end-to-end encrypted relational databases is an important topic in database security. To help guide the design of practical and secure encrypted databases, we identified five principles that revolve around security, functionality and efficiency. As an initial step, we designed the first STE-based encrypted relational database scheme that supports query optimization and that is legacy-friendly. We also built a system to illustrate our scheme's practical viability. We hope that our initial results will motivate and guide future work to improve the functionality, efficiency and security of encrypted relational database systems.

8. REFERENCES

[1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *ACM SIGMOD International Conference on Management of Data*, pages 563–574, 2004.

[2] G. Amjad, S. Kamara, and T. Moataz. Breach-resistant structured encryption. In *Proceedings on Privacy Enhancing Technologies (Po/PETS '19)*, 2019.

[3] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *CIDR*, 2013.

[4] S. Bajaj and R. Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *IEEE Trans. Knowl. Data Eng.*, 26(3):752–765, 2014.

[5] J. Bater, G. Elliott, C. Eggen, S. Goel, A. Kho, and J. Rogers. Smcql: secure querying for federated databases. *Proceedings of the VLDB Endowment*, 10(6):673–684, 2017.

[6] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In A. Menezes, editor, *Advances in Cryptology – CRYPTO '07*, Lecture Notes in Computer Science, pages 535–552. Springer, 2007.

[7] L. Blackstone, S. Kamara, and T. Moataz. Revisiting leakage abuse attacks. In *Network and Distributed System Security Symposium (NDSS '20)*, 2020.

[8] A. Boldyreva, N. Chenette, Y. Lee, and A. O'neill. Order-preserving symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2009*, pages 224–241, 2009.

[9] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[10] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology - CRYPTO '13*. Springer, 2013.

[11] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT '10*, volume 6477 of *Lecture Notes in Computer Science*, pages 577–594. Springer, 2010.

[12] M. Corp. Always Encrypted. https://msdn.microsoft.com/en-us/library/mt163865(v=sql.130).aspx.

[13] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *ACM Conference on Computer and Communications Security (CCS '06)*, pages 79–88. ACM, 2006.

[14] F. B. Durak, T. M. DuBuisson, and D. Cash. What else is revealed by order-revealing encryption? In *ACM Conference on Computer and Communications Security (CCS '16)*, 2016.

[15] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *IEEE Symposium on Security and Privacy (S&P '17)*, 2017.

[16] H. Hacigümücs, 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*, pages 216–227, 2002.

[17] Y. Ishai, E. Kushilevitz, S. Lu, and R. Ostrovsky. Private large-scale databases with distributed searchable symmetric encryption. Lecture Notes in Computer Science. Springer, 2016.

[18] S. Kamara and T. Moataz. SQL on Structurally-Encrypted Data. In *Asiacrypt*, 2018.

[19] S. Kamara, T. Moataz, and O. Ohrimenko. Structured encryption and leakage suppression. In *Advances in Cryptology - CRYPTO '18*, 2018.

[20] S. Kamara, T. Moataz, S. Zdonik, and Z. Zhao. Opx: An optimal relational database encryption scheme. Technical report, IACR ePrint Cryptography Archive, 2020.

[21] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '15, pages 644–655. ACM, 2015.

[22] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S.-G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 359–374. IEEE, 2014.

[23] R. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100, 2011.

[24] SAP Software Solutions. SEEED. https://www.sics.se/sites/default/files/pub/andreasschaad.pdf.

[25] D. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *IEEE Symposium on Research in Security and Privacy*, 2000.

[26] The KafeDB Team. KafeDB. https://github.com/zheguang/encrypted-spark, 2020.

[27] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *Proc. VLDB Endow.*, 6:289–300, 2013.

[28] D. Vinayagamurthy, A. Gribov, and S. Gorbunov. Stealthdb: a scalable encrypted database with full SQL query support. *PoPETs*, 2019(3):370–388, 2019.

[29] N. Volgushev, M. Schwarzkopf, B. Getchell, M. Varia, A. Lapets, and A. Bestavros. Conclave: secure multi-party computation on big data. In *EuroSys'19*. ACM, 2019.