Multi-Party Database for Secure Query Processing

Xiaoyuan Liu
UC Berkeley

Karen Tu
UC Berkeley

John Yang
UC Berkeley

Abstract

Data collaboration has become essential in many modern digital applications. Financial accounting, cyber threat intelligence gathering, and even the contact tracing of pandemics require systems that protect both security and privacy. However, many known solutions often provide efficiency at the cost of deployment difficulties, requiring non-collusion assumptions or hardware TEE. Targeting specific application scenarios, these schemes often lack generalizability and are obscure to engineers without security knowledge.

In this project, we develop a concise and engineering-friendly abstraction named Multi-Party Database , allowing secure data query processing from multiple mutually-untrusted data providers, which could power most data-access related privacy-preserving applications. Using LWE-based homomorphic encryption, our proposed protocol is deployment-friendly and requires only one round of communication. By using the same SQL interface as traditional DBMS's, the system is friendly to engineers without cryptographic knowledge. Existing applications can be migrated with minimum effort to achieve better privacy. Applying our technique of grouped binary matching and collision-tolerant folding to cut down both communication and computation costs, our system is efficient for real-world applications.

1 Introduction

Data collaboration has become essential in many modern digital applications. Today's settings often involve multiple collaborative parties looking to combine intelligence as a means for greater insight. Of the myriad of examples, for this particular project, we are particularly interested in projects where these parties are mutually untrusting, but are still interested in combining data and running queries on the conglomerated datasets. In cybersecurity, government agencies often collaborate with one another or hire private contractors to analyze such data. Hospitals and medical practitioners would benefit greatly from jointly measuring and predicting the probability of illness based on a patient's data that may be distributed across multiple institutions. In the financial world, systemic risk or individual credit can be assessed without compromising the confidentiality of the entities being investigated. In these three key scenarios and many additional practical use cases, there is a clear need for collaborative databases that can preserve the privacy of not only data, but also the queries being run on top of them. Databases that are capable of hiding both the content of an object in addition to compromising behavior such as access patterns are necessary to defend against a wide variety of vulnerabilities and exploits towards breaking private data access.

While many scenarios discussed above have different workflows, most of them can be formalized as the problem of processing secure queries. The participants are one client and many data providers. The data providers have some sensitive data and want to share it in a controllable way, while the client tries to analyze that data without revealing access patterns. Having the obliviousness in running queries, the client is confident to provide sensitive and accurate query details without exposing their preference or interest in data.

By having privacy guarantees for all participants on both sides, we can use this abstraction to cover many applications. Considering multiple mutually-untrusted data providers, we extend the above abstraction to a multiparty setting. Different data providers can collaboratively host a distributed query service but protect their original data records from each other.

Previous privacy query systems are unmatched with the current cloud infrastructure and often suffer from deployment difficulties. To make their protocol efficient, some require non-colluding servers[12, 7] and some rely on hardware Trusted Execution Environments [13]. In addition, many protocols in these solutions require multi-

ple rounds of communication and have sophisticated details. Engineers without cryptographic or security knowledge might find them hard to understand, making the industry-level implementation even more challenging.

In this work, we investigate a concise and engineering-friendly abstraction named Multi-Party Database . The design goals of the system are three-fold. First, to make our system easy to understand, we borrow the abstractions about tables and queries from traditional DBMS's. Clients directly use SQL to run queries to retrieve records and aggregate values from multiple data providers. Second, our system is deployment-friendly and does not require non-colluding servers or hardware enclaves. Such a design is better matched with the current cloud infrastructure and provides potential opportunities for scalable distributed deployment. Last, the system should be efficient enough to solve real-world problems. By taking advantage of the recent development of homomorphic encryption schemes and applying our own optimizations, we reduce both the computation and communication costs of running secure query processing.

To achieve the goals above, we design and implement a prototype for managing Multi-Party Database . The complete solution includes server-side daemon services that provide query interface and the client-side CLI that allows running SQL queries on required data providers and tables. Note that our system is different from encrypted database like CryptDB[8]. In our setting, the data is owned by data providers, not encrypted by the client. Our main contributions are three-fold:

- We define a concise and engineering-friendly abstraction named Multi-Party Database which covers scenarios for many real-world applications. We also specify its security properties and discuss its threat model.
- We design homomorphic encryption-based protocols and optimization techniques and implement a prototype system that achieves our design goals.
- By providing micro-benchmarks and running secure queries on synthetic workload, we evaluate our prototype system and demonstrate its efficiency for applications in the real world.

The rest of the paper is organized as follows. In section 2 we give an overview of the Multi-Party Database abstraction by explaining the workflow, the query model, and the threat model. Then we describe the protocol details for supporting secure query processing in section 3. In section 4, we discuss implementation details and specify the dependencies of our work. We evaluate the system and present micro-benchmarks and case studies in section 5. We compare related work in section 6 and conclude in section 7.

2 System Overview

Figure 1 illustrates the complete workflow of the Multi-Party Database . The system includes one client that allows the user to encrypt queries and run them obliviously, and a server-side daemon service that provides a query interface on behalf of the data provider. Each provider can host its own data with proper authentication so that only the allowed client can run queries. In the setup phase, all participants start their system using configuration files that contain the same table schemas and security parameters. Then data providers can load data from their database dumps and start the web service. The client also initializes its security context and analyzes the table schemas for later encryption of the queries.

To process one query, the client generates a fresh key to encrypt the query, randomly chooses a provider as the master provider, and sends the query to the master service. That service then obliviously processes the query to get an encrypted result. It also forwards the query to other providers, obliviously merges the encrypted result received from them with its own result before sending the merged result back to the client. The client then decrypts the result to finish this query.

2.1 Query model

Multi-Party Database uses a similar data model to other relational databases. The servers organize data records with the same schema into tables. In the multi-party setting, the client and other parties use the same schema. While different data providers host their own sensitive records, the execution result will be as if the query is running on a concatenated of tables from all data providers. Note that the client does not have direct access to this concatenated table, which leaves the possibility for policy checkers discussed later.

As also shown in fig. 1, Multi-Party Database mainly supports two types of queries: record retrieval and aggregation. In record retrieval operations, the client specifies the required columns and retrieves a small set of records that matches the given predicates. In record aggregation, the client computes an aggregation function, like a sum or a count, on those matching records and only receives the aggregation value. We target to support rich predicate expressions, covering boolean formulas, negations, and range queries for different data types such as boolean, signed and unsigned integer, string, and enum.

We make an assumption that only a small fraction of the records will be retrieved in each query, which limits the size of the encrypted result and makes efficient record retrieval possible. If the application requires giving the client access to a large fraction of the original records, the data providers might consider directly sending all data

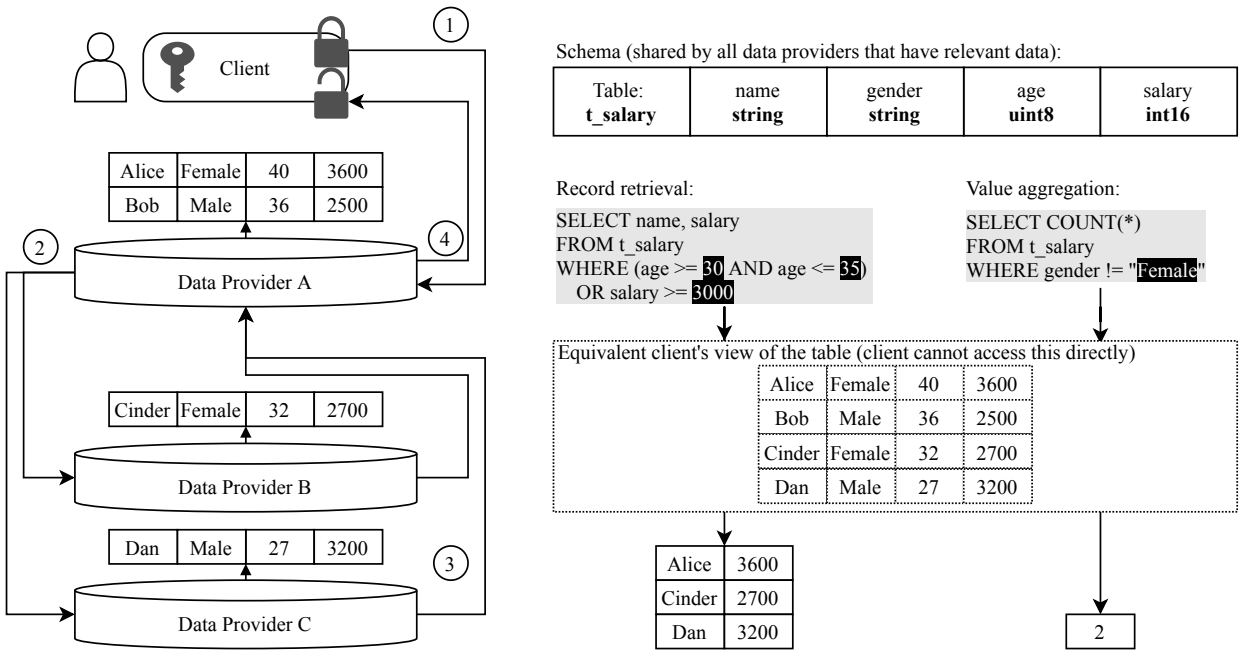

Figure 1: Multi-Party Database overview. Clients can run queries to retrieve records or get aggregation results from three different data providers that host query service for their own sensitive data. The SQL details in black background are hidden from all data providers.

to the client, which is similar to the naive solution of downloading all whole database in the PIR setting.

Similar to previous work [12], Multi-Party Database targets to hide sensitive query parameters instead of the whole SQL statement. As an example in fig. 1, the values in the two queries' predicates with a black background are hidden from all the data providers during the query processes. The result of the query will be one or more ciphertexts depending on the query type.

Although hiding the complete SQL statement seems to provide better privacy guarantees for clients, there are a few disadvantages. First, hiding the complete statement requires the protocol to be as slow as the most time-consuming query types to prevent side-channel attacks, which is computationally inefficient. It is hard to even define what the most time-consuming queries are considering the expressiveness of the predicates. Second, without knowing the query type, the encrypted query result needs to be as large as the retrieval of many records, which is communication inefficient. Although the number of retrieved records is limited, this would still add a large overhead to aggregations. Last, implementing a policy checker for a hidden query on the server-side is challenging. If the query template is revealed, the server can implement a separate policy checker to have fine-grained control over the query types. For example, it can allow queries only for aggregation values of certain columns and rejects all queries that retrieve original records. However, if the query is completely hidden, one may use more expensive techniques like zero-knowledge proofs to have the same functionality.

2.2 Threat Model

In this subsection, we describe how our system addresses four potential threats from both curious clients and providers in fig. 2. We also specify uncovered threats and explain the reason. Last, we discuss the malicious setting and side-channel attacks. We omit the discussion of normal authentication, authorization, and secure channel building procedures here since the privacy in our system is not provided by anonymity.

Multi-Party Database hides the client's access patterns from curious data providers. As illustrated in fig. 2.①, no data provider can reveal the sensitive parameters in the encrypted query or deduce which data records are selected by those parameters under the standard cryptographic assumptions.

Considering data providers might not trust each other with their sensitive data, Multi-Party Database also prevents one provider from accessing original records from other providers (fig. 2.②). Even the master service can only merge query results that are encrypted under the client's key so the procedure is still oblivious.

On the other side, Multi-Party Database also limits the access of the client to only content related to the query

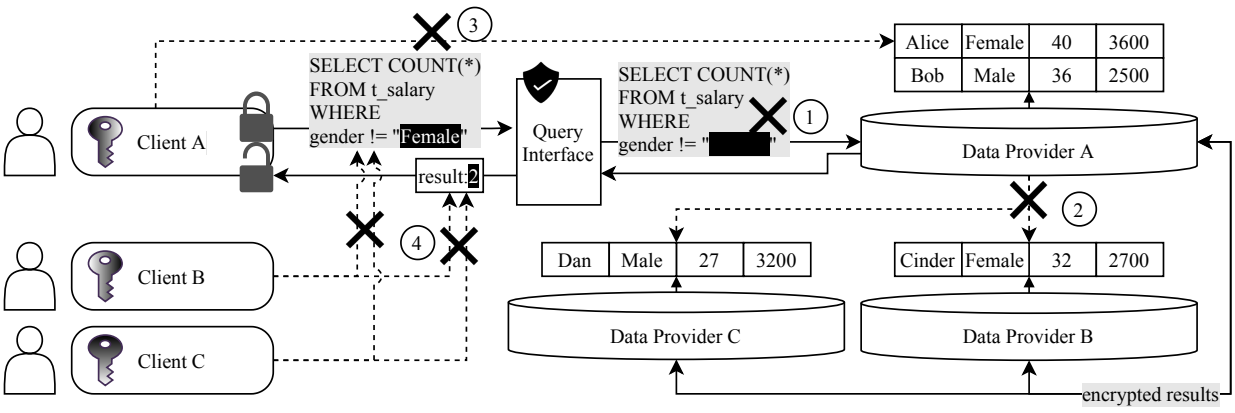

Figure 2: Threat model of Multi-Party Database . The solid lines describe the information flow specified by the secure protocol, while the dotted lines represent illegal access.

(fig. 2.③). For example, if the query type is aggregation, the client can only see the aggregated result without any original records. If the query type is retrieval, the system will not reveal records that do not match the predicates. This is similar to the privacy guarantees in symmetric PIR. Combined with an optional policy checker, such property will prevent curious clients from getting more data than authorized.

Although in the previous discussion we only mentioned one client, Multi-Party Database also supports multiple clients. Different clients use different keys for their query so that one client cannot decrypt the query result for another client (fig. 2.④). In a restricted setting where a client generates a fresh key for each query, even if the private key of a query is exposed, only the privacy of that query will be affected.

Although Multi-Party Database protects the privacy of both the client and the data providers against the previous four types of threats, the following two scenarios are not captured by our threat model. (1) If the data provider decides to modify its own data, the client will be unable to detect such modifications because the data providers are responsible for the authenticity of their own data. (2) If one data provider decides to collude with the client, that data provider will have access to the other provider's data which violates fig. 2.②. However, the functionality of the client causes this leakage and there is no way to prevent it without stopping the service of the system.

Although the previous discussions assume honest-but-curious clients and servers, our system can still provide privacy guarantees in the malicious setting. Since the server-side procedure is completely under encryption, a malicious provider cannot cause any harm other than returning the invalid results. Since the query template is in plaintext, the client cannot bypass the policy checker to retrieve more data than suggested in the query.

As many secure query systems are vulnerable to side-channels like timing analysis, our query processing protocol guarantees that the execution procedure is independent of any sensitive parameters. However, the server-side execution time does leak information about table size. One solution is to pad the execution time to a fixed value and wait before sending back the encrypted results.

3 Query processing

In this section, we describe the secure query processing protocol that supports Multi-Party Database . Our protocol is concise because unlike similar works, it only requires one round of communication between the client and the master data provider, which is exactly the same as the normal distributed DBMS. The client uses its secure query planner to generate an encrypted query based on a given SQL statement, and then sends this encrypted query to the master data provider. The service from the master data provider executes this encrypted query on its own data to get the encrypted result. It also forwards the query to other data providers and merges all encrypted results. At last, it sends this result back to the client.

In the following subsections, we first cover the preliminaries of our protocol. Then we specify basic building blocks to support the protocol, including two core techniques: grouped binary matching and range decomposition. After that, we describe how the client generates a secure execution tree with its secure query planner and explain how data providers can use that tree to execute the query. We also describe how the master data provider merges the result when there are multiple data providers.

3.1 Preliminaries

Multi-Party Database is implemented using BFV Homomorphic Encryption (HE) [5]. In this section, we will give an overview of BFV HE, as well as the PIR techniques that Multi-Party Database extends.

3.1.1 Homomorphic Encryption

A Fully Homomorphic Encryption (FHE) scheme is a powerful primitive that allows for arbitrary functions to be computed over encrypted values. Given ciphertexts

$$\text{Enc}_{pk}(p_1)...\text{Enc}_{pk}(p_n) \rightarrow$$
$$\text{Dec}_{sk}(f(\text{Enc}_{pk}(p_1),...\text{Enc}_{pk}(p_n))) = f(p_1...p_n)$$

for arbitrary f that can be computed efficiently. A homomorphic scheme is defined as a set of 4 algorithms:

- $\text{KeyGen}(\lambda) \rightarrow (pk, sk)$ where λ is the security parameter, pk is the public key, and sk is the secret key
- $\text{Enc}_{pk}(m) \rightarrow c$ where $m \in M$, the plaintext space, and $c \in C$, the ciphertext space
- $\text{Dec}_{sk}(c) \rightarrow m$ if $c = \text{Enc}_{pk}(m)$ and the provided sk matches pk. Otherwise, output $\perp$
- $\text{Eval}_{pk}(F, c)$ outputs a ciphertext, where F is the function. For any function F and $m, c = \text{Enc}_{pk}(m)$ $\text{Dec}_{sk}(\text{Eval}_{pk}(F, c)) = F(m)$

Many state-of-the-art HE libraries use BGV, BFV, and CKKS as the underlying FHE schemes. The Multi-Party Database uses BFV [5], because it allows for exact computation on encrypted integers. BFV is a lattice-based scheme and it depends on the hardness of Ring Learning with Errors (RLWE), which is the regular Learning with Errors problem specifically for polynomial rings. BFV is a leveled FHE scheme; the desired multiplicative depth (number of sequential multiplications) determines the scheme's parameters. This is because each operation incurs noise, as RLWE hardness is based on adding noise to a lattice point. Therefore, it is important to correctly select parameters that allow enough noise budget.

An important optimization of FHE computation when using BFV is batch encoding based on the Chinese Remainder Theorem (CRT). For a polynomial of degree N where the plaintext modulus parameter is T, CRT batching allows for storing N values modulo T. As long as T is prime and congruent to 1 mod $2N$, the plaintext can be seen as a matrix of size 2 by $\frac{N}{2}$, and operations on the matrix can benefit from SIMD instruction parallelism.

```
function CLIENT_QUERY(pk, record_index)
    for i = 0 to n - 1
        c[i] = Enc(pk, i == record_index ?
            ↪ 1 : 0)
    return query = c[0]...c[n-1]

function SERVER_ANSWER(query = c[0]...c[n
    ↪ -1], records)
    for i = 0 to n - 1
        a[i] = records[i] * c[i]
    return sum(a[0]...a[n-1])
```

Figure 3: XPIR using a HE scheme Gen, Enc, Dec with key pair (pk, sk). The client runs CLIENT_QUERY and the server runs SERVER_ANSWER. The database has n records.

3.1.2 PIR

Private information retrieval (PIR) is a protocol between a client and a database, where the client can retrieve one element of the database without the database owner knowing which element was retrieved. Most PIR protocols can be grouped into information theoretic PIRs (IT-PIRs) and computational PIRs (CPIRs). The security guarantees for IT-PIR protocols typically rely on multiple non-colluding servers, while CPIR protocols rely on the computational limits of servers.

XPIR [1] is a CPIR scheme. This scheme does not require database replication, but this comes at the expense of communication costs. In this scheme, a query is a vector with n (number of elements in the database) ciphertexts, where each ciphertext is encryption of an indicator value of whether the index is the desired record. We call this ciphertext the indicator vector, and will continue to use this term to discuss our own protocol. Figure 3 describes the HE-based XPIR protocol.

3.2 Building Blocks

Compared with the XPIR protocol, one major challenge of our protocol is to generate the indicator vectors from both the encrypted predicates from the client, and the plaintext records from the server. Below we discuss how we support equality check, boolean formulas, and range queries in predicates based on FHE.

3.2.1 Equality Check

Consider a simplified scenario: the client has an 8-bit unsigned integer v for column c. The data provider generates an indicator vector *ind* that has the same length as the database. All records $r[i]$ with $r[i][c]$ equal to v should

have an entry of encrypted 1's in that vector and the rest are encrypted 0's, just like in XPIR.

$$ind[i] := \begin{cases} Enc_k(1) & r[i][c] = v \\ Enc_k(0) & r[i][c] \neq v \end{cases}$$

The client can later use that indicator vector which is already stored on the server-side to retrieve records just like in XPIR, or run aggregation computation (like a sum) after masking the original records with the indicator vectors using plaintext multiplication.

Now the challenge is how the data provider can generate the indicator vector *ind* with c, r, and an encrypted v. Considering making basic operations efficient, to avoid expensive primitives like garbled circuits, we use a naive method which requires that the client generates $2^8 = 256$ ciphertexts m covering all possible values of that 8-bit unsigned integer v in advance, where

$$m[j] := \begin{cases} Enc_k(1) & j = v \\ Enc_k(0) & j \neq v, 0 \leq j \leq 255 \end{cases}.$$

Then the data provider can generate *ind* by looping through its record and setting $ind[i] := m[r[i][c]]$. Note that this only works when there is a limited number of possible values. We discuss how to support normal 32-bit integers in section 3.2.3.

3.2.2 Boolean Formulas

Now that we can generate indicator vectors for equality checking of a specific column, the next step is to support *and*, *or*, and *not* operations between indicator vectors, which is equivalent to performing these operations element-wise for 1's and 0's inside the vector. Under homomorphic encryption, we can perform these operations using additions and multiplications as follows

$$\begin{aligned} and(ind_a, ind_b) &:= ind_a * ind_b \\ not(ind) &:= 1 - ind \\ or(ind_a, ind_b) &:= not(and(not(ind_a), not(ind_b))) \\ &= ind_a + ind_b - ind_a * ind_b \end{aligned}.$$

Layered combinations of boolean operations in the predicates can be performed by processing the indicators with the same sequence. However, we do optimize the actual execution sequence during query generation. Details are discussed in section 3.3.

3.2.3 Grouped Binary Matching

To allow equality check not only for 8-bit integers but also for normal 32-bit or 64-bit integers, we present a technique called grouped binary matching, which decomposes the task of large equality check into multiple 8-bit equality check and some *and* operations. By separating the binary representation of the original 32-bit value into four 8-bit groups, we now have four 8-bit integers. The equality of the 32-bit integer holds if and only if the equality for all four 8-bit integers holds. From the indicator representation,

$$x = v \leftrightarrow and \begin{cases} and \begin{cases} x \equiv v \pmod{2^8} \\ \lfloor \frac{x}{2^8} \rfloor \equiv \lfloor \frac{v}{2^8} \rfloor \pmod{2^8} \end{cases} \\ and \begin{cases} \lfloor \frac{x}{2^{16}} \rfloor \equiv \lfloor \frac{v}{2^{16}} \rfloor \pmod{2^8} \\ \lfloor \frac{x}{2^{24}} \rfloor \equiv \lfloor \frac{v}{2^{24}} \rfloor \pmod{2^8} \end{cases} \end{cases}.$$

Since we have already supported both 8-bit equality check and the boolean operation of *and*, now we support 32-bit equality check. The procedure of 64-bit equality check is similar and requires one additional layer.

To support string matching, we use pseudo-random permutation (PRP) with block cipher like AES to generate 32-bit or 64-bit values, and the rest is the same.

3.2.4 Range Query

The range queries are challenging because they require comparison between encrypted boundaries and the value in the data record. We first consider a simplified scenario where the client has two 8-bit unsigned integers v_l and v_r and wants to generate an indicator vector that satisfies

$$ind[i] := \begin{cases} Enc_k(1) & v_l \leq r[i][c] \leq v_r \\ Enc_k(0) & otherwise \end{cases}.$$

The solution is similar to 8-bit equality check - the client generates

$$m[j] := \begin{cases} Enc_k(1) & v_l \leq j \leq v_r \\ Enc_k(0) & otherwise, 0 \leq j \leq 255 \end{cases}$$

and sends it to the data provider. Then the data provider again sets $ind[i] := m[r[i][c]]$ to get the indicator vector without any homomorphic operations.

3.2.5 Range Decomposition

The challenge is to support range queries for 32-bit/64-bit integers. We omit the discussion for 64-bit integers since it is similar to the solution for 32-bit integers.

The idea is similar to grouped binary matching, and by utilizing the property of binary representation, we can decompose one 32-bit range operation into a combination of multiple 8-bit range operations, multiple 8-bit

equality checks, and some *and* and *or* boolean operations. From the indicator representation,

$v_l \le x \le v_r \leftrightarrow$

(equality checks & range operations are under mod 2^8)

$$
or\begin{cases}
or\begin{cases}
and\begin{cases}
\lfloor \frac{x}{2^{24}} \rfloor = \lfloor \frac{v_l}{2^{24}} \rfloor \\
or\begin{cases}
\lfloor \frac{v_l}{2^{16}} \rfloor + 1 \le \lfloor \frac{x}{2^{16}} \rfloor \\
and\begin{cases}
\lfloor \frac{x}{2^{16}} \rfloor = \lfloor \frac{v_l}{2^{16}} \rfloor \\
or\begin{cases}
\lfloor \frac{v_l}{2^{8}} \rfloor + 1 \le \lfloor \frac{x}{2^{8}} \rfloor \\
and\begin{cases}
\lfloor \frac{x}{2^{8}} \rfloor = \lfloor \frac{v_l}{2^{8}} \rfloor \\
v_l \le x \le 255
\end{cases}
\end{cases}
\end{cases}
\end{cases}
\end{cases} \\
and\begin{cases}
\lfloor \frac{x}{2^{24}} \rfloor = \lfloor \frac{v_r}{2^{24}} \rfloor \\
or\begin{cases}
\lfloor \frac{x}{2^{16}} \rfloor \le \lfloor \frac{v_r}{2^{16}} \rfloor - 1 \\
and\begin{cases}
\lfloor \frac{x}{2^{16}} \rfloor = \lfloor \frac{v_r}{2^{16}} \rfloor \\
or\begin{cases}
\lfloor \frac{x}{2^{8}} \rfloor \le \lfloor \frac{v_r}{2^{8}} \rfloor - 1 \\
and\begin{cases}
\lfloor \frac{x}{2^{8}} \rfloor = \lfloor \frac{v_r}{2^{8}} \rfloor \\
0 \le x \le v_r
\end{cases}
\end{cases}
\end{cases}
\end{cases}
\end{cases}
\end{cases} \\
\lfloor \frac{v_l}{2^{24}} \rfloor + 1 \le \lfloor \frac{x}{2^{24}} \rfloor \le \lfloor \frac{v_r}{2^{24}} \rfloor - 1
\end{cases}
.
$$

By the decomposition of long ranges, we convert one 32-bit range operation into 7 8-bit range operations, 6 8-bit equality checks, 6 *and* operations, and 6 *or* operations. While this works for unsigned integers, for signed integers, we simply apply a range-preserving conversion to remove the sign by adding $0x80000000$.

3.3 Query Generation

Having all the necessary building blocks for supporting rich predicates, a secure query planner is responsible for encrypting the query at the core of the client implementation. The planner takes the SQL statement as the input and outputs a tree-like data structure called a secure execution tree (fig. 4), where all sensitive query parameters are already encrypted. Here we describe the details of the generation and encryption procedure.

The secure query planner parses the SQL statement to get the query type as well as the tree of predicates. It then converts this tree into a tree-structured execution plan named secure execution tree where each node except for the root in the tree corresponds to an indicator generation operation. Before sending it to a randomly selected master data provider, that tree needs to be revised through 4 stages: pre-expansion optimization, expansion, post-expansion optimization, and encryption.

```
SELECT name, salary
FROM t_salary
WHERE (age >= 30 AND age <= 35) OR salary >= 3000
```

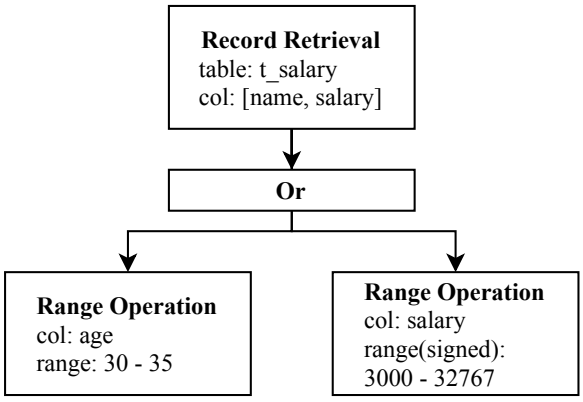

Figure 4: An example of secure execution tree. In the example, the tree has not been expanded and encrypted for better illustration.

In the pre-expansion optimization stage, the planner analyzes the high-level equality checks and range operations and tries to reduce the total number of them. For equality checks under the same *and* operation, the planner reduces the number of operations by replacing them with a "multiple equality check" operation. Instead of comparing the original concerned columns one by one, the new operation performs the equality check on the value generated by performing PRF on a concatenation of all concerned columns. The planner also analyzes the range operations to use as few range operations as possible. For example, if $age <= a$ or $age >= b$ appears in the predicates, the planner revises it to be $not(range(a,b))$ to save one range operation. Note that the procedure is independent of the query parameters. So even if $range(a,b)$ does not make sense ($a > b$), we still keep it as is and generate $2^8 = 256$ ciphertexts of zeros in the later encryption stage. In this way, the structure of the secure execution tree leaks no information about the sensitive query parameters.

In the expansion stage, the planner loads the table schema from configuration files and expand operations for long data types into basic 8-bit operations. It uses the grouped binary matching and range decomposition techniques discussed in the previous section. To simplify the execution, it converts all signed predicates, replaces *or* operations with *and* and *not* operations, and removes pairs of consequent *not* operations. After this stage, the secure execution tree will only contain *and*, *not*, and basic unsigned 8-bit operations.

In the post-expansion optimization stage, the planner rebalances the secure execution tree to save noise bud-

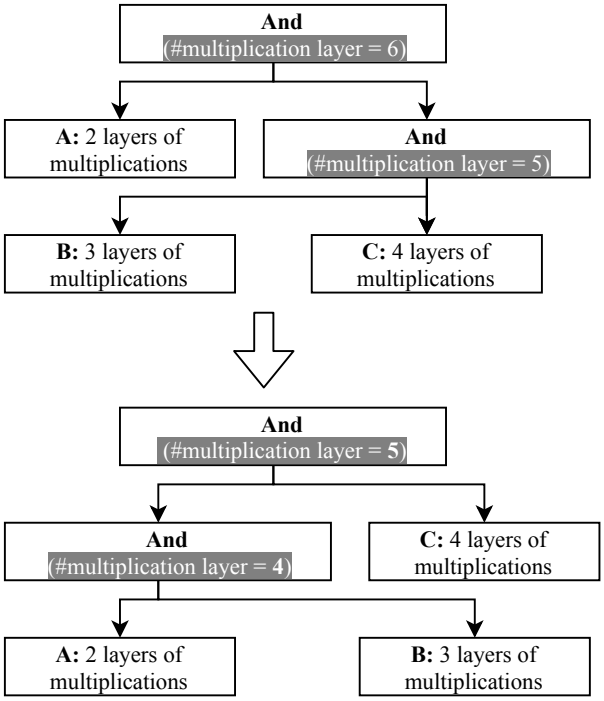

Figure 5: The execution tree rebalancing procedure for a subtree. A, B, and C are children subtrees that have been rebalanced.

gets for HE. As illustrated in fig. 5, it uses a reversed depth-first searching order to greedily rotate the subtree according to the number of multiplications in each child.

Finally, in the encryption stage, the planner replaces plaintext values in the leaf node with grouped ciphertexts for 1's and 0's. After that, the clients send the revised secure execution tree along with the Galois and relinearization keys to the master data provider for execution.

3.4 Query Execution

Once the data provider receives the keys and secure execution tree, it starts to generate the indicator vector by iterating through the tree in a depth-first searching order. After accessing all the nodes converted from predicates, the data provider gets the final indicator vector that suggests which records in the database should be selected. The final indicator vector is quite similar to the one in XPIR protocol. However, one deterministic difference is that it might contain multiple entries of 1's.

3.4.1 Record Aggregation

Record aggregation is relatively straightforward. The *count* operation is simply a sum of the indicator vector itself. The *sum* operation is the sum of the multiplication result between the indicator vector and values in the original selected column. A combination of *count* and *sum* operation can support *avg*, the average aggregation. We do identify this as an extra leakage. But considering the expensiveness of homomorphic division, we think in most cases this is a good trade-off.

To support comparison aggregation operators, *max* and *min*, we first define a new aggregation operator that is not in the original SQL syntax: *exist* (not $EXISTS$). *exist* is similar to *count*, but it only reveals whether the count is zero, not the exact value. Supporting *exist* is simple: by allowing the master provider to multiply the *count* result with a randomly selected non-zero value, the actual value of the result is masked.

With the *exist* operator, we support *max* and *min* by replacing it with multiple queries that run the binary search on the existence of column value. For example, assuming column c contains 8-bit unsigned integers,

$$\text{SELECT } max(c) \text{ FROM } t$$

can be replaced by

$$\text{SELECT } exist(*) \text{ FROM } t \text{ WHERE } c >= 128,$$
$$\text{SELECT } exist(*) \text{ FROM } t \text{ WHERE } c >= 64\ldots$$

Then we can find the maximum or minimum within $log(b)$ queries, where b is the bit width of the column data type, even in a multi-party setting.

3.4.2 Record Retrieval

Unlike aggregation, record retrieval is non-trivial. We need to retrieve multiple records, but homomorphically adding up all the multiplication results between indicator and column values like XPIR no longer works. On the other side, simply sending back all ciphertexts is privacy-preserving but not realistic, as it leads to communication that has size linear to the database. Such a communication cost is unacceptable.

To deal with this challenge, we take advantage of the assumption that only a small fraction of the record will be retrieved, and propose a simple technique called collision-tolerant folding similar to [3]. Instead of adding up all n records into 1 final value, we equally separate the original records into s slots and calculate the sum within each slot. If no slot contains more than 1 selected record, the retrieval will be successful. By allowing a certain degree of collisions, we can raise the success rate above a certain threshold with a slot ratio that is sublinear to the maximum number of the retrieved records.

We first discuss the basic solution which does not tolerate collisions. The first modification is to shuffle the data record for each query before generating any indicators. It does not affect the query result but makes it possible to rerun the query once the collision happens.

After getting the final indicator, we multiply the selected columns in records by the final indicator vector to set unrelated records to zero. We group the records according to the remainder of dividing by slot number and add up all records within the same group.

To make the basic solution collision-tolerant, we change the data representation to allow collision. In the basic solution, if two non-zero selected records are added to the same slot, the result will be unrecoverable. Assuming two records are represented by x and y, then there is no way to recover them given the sum $x+y$. However, if we represent records as x, x^2 and y, y^2, then given $x+y, x^2+y^2$, we can simply recover two original values by

$$x, y = \frac{(x+y) \pm \sqrt{2(x^2+y^2)-(x+y)^2}}{2}.$$

Although it approximately triples the size of the ciphertexts, significantly more records can be retrieved within one query compared to the basic solution with the same success rate, which is generally a good trade-off. It is also possible to allow higher degrees of collisions, but the record number cannot amend for the required 7 times extra ciphertext space.

Following the analysis in [3], to avoid any collision with a fixed high probability, the ratio $r = \frac{s}{n}$ between the number of slots s and the number of selected record m needs to be approximately linear to m. However, if we allow 2 collisions, we only need the square root of the slot ratio $\sqrt{r}$ to be approximately linear to the selected record number m. In practice, one ciphertext with the batching optimization is often enough for retrieving a few hundreds of records with a 99.99% success rate allowing 2 collisions. In summary, the collision-tolerant folding technique cuts down the retrieval communication cost in most cases to only a few ciphertexts.

3.5 Result Merging

We focused primarily on one master data provider in the previous discussion. In this subsection, we describe the merging process after the master data provider receives encrypted results from other data providers for different query types. The encrypted result will be one ciphertext for record aggregation, and a list (matrix if batching is used) of ciphertexts for record retrieval. Details of the data layout will be explained together with batching optimization in section 4.

Since all results are encrypted under the same group of keys from the client, for record aggregation, the master data provider can directly perform necessary aggregation operations on results from other providers. For record retrieval, because the encrypted results use the same slot and column structure, the master data provider continues the folding procedure by aligning the slots and directly adding them up. Since the total number of retrieved records is limited and the shuffling procedure is random, the previous discussion on success rate still holds.

Once the master data provider merges the results according to the query type, it sends the final encrypted result back to the client for decryption, which finishes the whole secure query processing procedure.

4 Implementation

Based on our protocol discussed in section 3, we implemented a prototype of Multi-Party Database with C++. Extended from our in-memory relational database, we implemented both a client-side CLI and a server-side web service. We use Microsoft SEAL[10] for homomorphic encryption. For security parameters, we use the BFV scheme with a 2^{14}-degree polynomial and CRT-batching with an 18-bit plaintext modulus, which provides a noise budget that allows at most 11 layers of multiplications. While this is enough for most of the queries, we also have a set of backup security parameters that supports at most 29 layers (2^{15}-degree, 12-bit).

Correctly using the CRT-batching mentioned in section 3.1 to amortize the multiplication cost is critical for making the implementation efficient. In fact, that key observation motivates the protocol designs like using vector operations and having a secure execution tree. In the rest of this section, we explain how we utilize batching operations to amortize both communication and computation cost. We describe the ciphertext data layout and discuss how to compress vector ciphertexts with rotations.

In the last encryption stage of query generation, instead of generating 256 ciphertexts for each basic building block node, the client creates multiple vectors with 256 entries each and batching the concatenation of them into as few ciphertexts as possible. This way, one ciphertext can cover $2^{14-8} = 64$ basic building block nodes. As a result, in most cases, one query only requires one ciphertext. Later, the data provider extracts the indicator according to the value in the data record by rotating this ciphertext to the correct location and masking it with plaintext multiplications.

While the aggregation operations do not need complicated batching, the data layout in record retrieval requires careful design. As shown in fig. 6, the ciphertext is separated internally to tolerate 2 collisions. Horizontally, multiple ciphertexts cover the complete bit representation of records in the slots, while vertically, more rows allow extra slots to retrieve more records.

The CRT-batching also simplifies the folding procedure. Instead of doing extra rotations inside ciphertext, the data provider only needs to add the complete ciphertexts for shuffled and masked data records. As long as

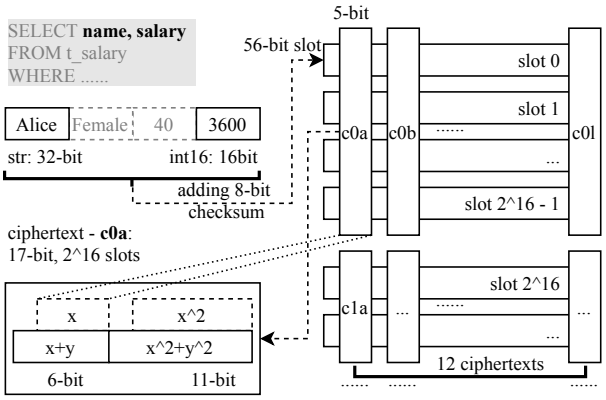

Figure 6: Data layout of ciphertext during record retrieval.

Operation		Amortized running time for each record (µs)
HE	Plaintext add	0.078
	Ciphertext add	0.014
	Plaintext mul	0.469
	Ciphertext mul	3.048
	Rotation	1.063
	Relinearization	1.102
Building block	8-bit equality check 8-bit range query	442.252
	Boolean - not	0.043
	Boolean - and	4.122
	Boolean - or	4.233
Composition	16-bit equality check	888.626
	16-bit range query	2227.970
	32-bit equality check	1781.374
	32-bit range query	5799.406

Table 1: Micro-benchmark for Multi-Party Database .

	Multi-Party Database	Splinter
Query Processing	14789 ms	450 ms

Table 2: Comparison of query processing time.

the addition is horizontally aligned, it can execute the folding procedure correctly.

5 Evaluation

In this section, we answer the following questions.

- How long does it take to run normal operations on Multi-Party Database ?
- How does it compare with other similar works?
- Is it practical enough for supporting real-world applications?

Our evaluations below are based on simulations using our implementation of the protocol. All of the experiments used 2.3 GH-z 8-Core Intel Core i9 machine with 16GB of RAM.

5.1 Micro-benchmark

Table 1 shows micro-benchmarks for the time consumption of different operations used in Multi-Party Database . The 8-bit building block operations are the most time-consuming ones since they involve many rotations (≈59%) and plaintext multiplications (≈24%) for extracting the grouped encryptions of 1's and 0's in the ciphertext according to the values in the data records. However, it still takes less than one millisecond, meaning that on average, we can process more than 1k original records per second. Considering that the time to run the equality check on one plaintext record is around 0.181 µs, our 8-bit equality check is about 2500x slower than a no-protection implementation and 32-bit about 10^4x.

Note that our solution is scalable by nature because an entity can always act as multiple data providers and split its table into shards. The master data provider sends the query to all other providers, who execute the query in a parallel fashion. It allows our system to support an even larger workload in a distributed setting.

5.2 Comparison with Splinter

Splinter [12] is a recent system with a similar threat model. Both Splinter and the Multi-Party Database focus on hiding query parameters. The major difference between Splinter and the Multi-Party Database is the underlying cryptographic primitive. The Multi-Party Database relies on homomorphic encryption, while Splinter uses function secret sharing. Because function secret sharing requires non-colluding parties to receive different shares of the function, Splinter requires duplicating the database across at least two servers, which each receives a share of the function. We use a basic version of Splinter in C++ for some preliminary comparison for a count aggregation query with one equality predicate. Although code for Splinter is not publicly available, the FSS library that it is based on is. Therefore, we run a simple count aggregation query, specifically: "SELECT COUNT(*) where salary = 2500" then report the simulated computation time on a database 2^{14} records in table 2.

In the Multi-Party Database, the processing is using HE to aggregate the indicator vector. In Splinter, the processing is computing the function shares on the corresponding record attributes. To avoid the extra non-colluding server, Multi-Party Database is about 33x slower than the Splinter solution.

As the number of predicates increases, in the Multi-Party Database, for each additional record attribute, the

client must generate another indicator vector. For Splinter, the size of the function share increases; as the size of the function share depends on the size of the input domain, which is proportional to the number of predicates. For both systems, the communication costs will increase. However, it is unclear for which system the rate of increase is lower. Therefore, for future work, we plan to test more queries with richer predicates in different scenarios for better comparison.

5.3 Evaluation on Real-world Application

One major dataset we used is focused on cyber threat intelligence, specifically for threat detection based on transactions from hashed wallet addresses. The overarching goal is to detect fraudulent or suspicious cryptocurrency transactions. Today, blockchain based financial technology and ledgers, notably Bitcoin and Ethereum, allow anonymous entities to perform online transactions without being tracked or identified as is the case via traditional financial institutions. By the design of blockchain, these transactions are recorded on a global, immutable ledger. In recent years, intelligence agencies have put together information via a variety of 3rd party information linking specific patterns of transaction behavior to malicious intentions. The goal in this situation would be to combine transaction history and 3rd party intelligence data to create a dataset that would allow analysts to identify if certain accounts are behaving suspiciously. Privacy is particularly important in this form of collaboration because while analysts should be able to run queries to identify suspicious accounts, they may not be cleared to know who or what is actually associated with the accounts. For instance, if an intelligence contractor like Palantir was hired by a U.S. agency to identify suspicious accounts, the multi-party database would ideally ensure that Palantir can carry out such analysis, but not be able to figure out who the account holders are. On the other side, the U.S. agency would receive the information it needs, but would not be able to reverse engineer Palantir's proprietary approach to identifying the accounts. We anticipate that such analysis would require heavy use of conditional clauses with aggregation.

To see whether Multi-Party Database is practical for real-world applications like multi-party cyber threat hunting, we start from an existing application: matching the suspicious wallet addresses. Existing industry solutions often use centralized infrastructure and hashing-based methods to hide the original records, which are vulnerable to brute-force attacks on the plaintext. As an alternative, our Multi-Party Database eliminates such vulnerabilities and naturally supports the distributed setting where all data providers can better protect their valuable cyber threat intelligence.

In our synthesized scenario, 10 data providers collaboratively host a service of suspicious wallet address identification that checks the existence of a given wallet address in a known wallet address list. We assume each data provider holds 10^4 records and synthesize the dataset. The analyst uses the client CLI of Multi-Party Database to query "SELECT count(*) FROM t_cti WHERE wallet_address='xxx' ".

The query execution time of Multi-Party Database is around 30 seconds and the merging among all data providers takes far less than 1 second. In terms of the communication cost, the size of both the secure query and the encrypted result is approximately 1.8 MB. Assuming 10 Mbps point-to-point network bandwidth between all nodes, the analysts should be able to receive and decrypt the final result within 40 seconds. With more optimizations like multi-threading in the implementation, this running time can be further reduced, which is one of our future direction.

6 Related Work

XPIR is a practical protocol that has the same goal as the Multi-Party Database of allowing a client to privately retrieve database records. However, one drawback of XPIR is that it is used to retrieve an entire single record, based on the index number of the record. This is not a realistic protocol for the Multi-Party Database setting because the client will not know the indices of their desired records. In addition, clients can select multiple records on arbitrary conditions for arbitrary sets of record attributes. SealPIR [2] focuses on modifying CPIR by compressing and batching queries using probabilistic batch codes (PBCs). The Multi-Party Database compresses ciphertexts instead, and uses CRT based batching.

Splinter [12] and Blind Seer [7] both protect client query privacy under non-colluding server setting, while Opaque [13] uses enclaves to run secure queries and uses oblivious sort to hide access pattern. Both of them improve efficiency at the cost of deployment difficulties. Encrypted databases such as CryptDB [8] are for protecting client data, which is different from our line of work. Conclave [11] is a secure Multi-Party Computation (MPC) oriented approach to running analysis on large datasets. As we discussed, while the general MPC approaches have proven to be quite a hit on performance, we recognize how their framing of the problem and descriptions of potential approaches are quite useful. While revealing some insensitive information discussed in their paper is a way to improve efficiency, our Multi-Party Database serves as an alternative solution to avoid using general MPC.

An important aspect of this project is hiding access patterns and prevention of side channel attacks. For in-

spiration, we looked towards Oblivious RAM's details [9, 6, 4] on the threat model and evaluation for perspectives on how to evaluate our own system. As detailed in our initial related work document, the ORAM paper provided a great first stepping stone to thinking about how to build upon the existing solutions to hiding access patterns and nullifying side-channel attacks.

7 Conclusion

In this paper, we define Multi-Party Database , a concise abstraction for secure query processing. By using our proposed techniques including grouped binary matching, range decomposition, and collision-tolerant folding, the protocol achieves practicals performance in real-world applications without the non-colluding assumption or the hardware enclave.

In the future, we will continue to evaluate our protocol in various multi-party settings to validate its design, explore new functionalities like supporting join operations, and further improve the overall performance.

References

[1] AGUILAR-MELCHOR, C., BARRIER, J., FOUSSE, L., AND KILLIJIAN, M.-O. Xpir : Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies 2016* (08 2015).

[2] ANGEL, S., CHEN, H., LAINE, K., AND SETTY, S. Pir with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), pp. 962–979.

[3] CORRIGAN-GIBBS, H., BONEH, D., AND MAZIÈRES, D. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 321–338.

[4] CROOKS, N., BURKE, M., CECCHETTI, E., HAREL, S., AGARWAL, R., AND ALVISI, L. Obladi: Oblivious serializable transactions in the cloud. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 727–743.

[5] FAN, J., AND VERCAUTEREN, F. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch. 2012* (2012), 144.

[6] MISHRA, P., PODDAR, R., CHEN, J., CHIESA, A., AND POPA, R. A. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 279–296.

[7] PAPPAS, V., KRELL, F., VO, B., KOLESNIKOV, V., MALKIN, T., CHOI, S. G., GEORGE, W., KEROMYTIS, A., AND BELLOVIN, S. Blind seer: A scalable private dbms. In *2014 IEEE Symposium on Security and Privacy* (2014), IEEE, pp. 359–374.

[8] POPA, R. A., REDFIELD, C. M., ZELDOVICH, N., AND BALAKRISHNAN, H. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), pp. 85–100.

[9] REN, L., FLETCHER, C., KWON, A., STEFANOV, E., SHI, E., VAN DIJK, M., AND DEVADAS, S. Constants count: Practical improvements to oblivious {RAM}. In *24th {USENIX} Security Symposium ({USENIX} Security 15)* (2015), pp. 415–430.

[10] Microsoft SEAL (release 3.0). http://sealcrypto.org, Oct. 2018. Microsoft Research, Redmond, WA.

[11] VOLGUSHEV, N., SCHWARZKOPF, M., GETCHELL, B., VARIA, M., LAPETS, A., AND BESTAVROS, A. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019* (2019), pp. 1–18.

[12] WANG, F., YUN, C., GOLDWASSER, S., VAIKUNTANATHAN, V., AND ZAHARIA, M. Splinter: Practical private queries on public data. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)* (2017), pp. 299–313.

[13] ZHENG, W., DAVE, A., BEEKMAN, J. G., POPA, R. A., GONZALEZ, J. E., AND STOICA, I. Opaque: An oblivious and encrypted distributed analytics platform. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)* (2017), pp. 283–298.