Waldo: A Private Time-Series Database from Function Secret Sharing

Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica
University of California, Berkeley

***Abstract*—Applications today rely on cloud databases for storing and querying time-series data. While outsourcing storage is convenient, this data is often sensitive, making data breaches a serious concern. We present Waldo, a time-series database with rich functionality and strong security guarantees: Waldo supports multi-predicate filtering, protects data contents as well as query filter values and search access patterns, and provides malicious security in the 3-party honest-majority setting. In contrast, prior systems such as Timecrypt and Zeph have limited functionality and security: (1) these systems can only filter on time, and (2) they reveal the queried time interval to the server. Oblivious RAM (ORAM) and generic multiparty computation (MPC) are natural choices for eliminating leakage from prior work, but both of these are prohibitively expensive in our setting due to the number of roundtrips and bandwidth overhead, respectively. To minimize both, Waldo builds on top of function secret sharing, enabling Waldo to evaluate predicates non-interactively. We develop new techniques for applying function secret sharing to the encrypted database setting where there are malicious servers, secret inputs, and chained predicates. With 32-core machines, Waldo runs a query with 8 range predicates over 2^{18} records in 3.03s, compared to 12.88s for an MPC baseline and 16.56s for an ORAM baseline. Compared to Waldo, the MPC baseline uses 9 – 82× more bandwidth between servers (for different numbers of records), while the ORAM baseline uses 20 – 152× more bandwidth between the client and server(s) (for different numbers of predicates).**

I. Introduction

Organizations today rely on the ability to continuously collect and analyze time-series data. To cheaply store and query this data, organizations turn to cloud databases [7, 52, 101]. However, many systems produce time-series data that is not only useful, but also *sensitive*. For example, remote patient monitoring systems and smart homes both generate time-series data that users might not want to store in the cloud due to the danger of data breaches [68, 73, 77, 90].

One solution to this problem is to perform queries over encrypted time-series data, as Timecrypt [21] and Zeph [22] do. These systems have two serious limitations. The first is that they only support aggregation by time over a data stream (e.g. average heart rate over a week). Many modern time-series databases [9, 52, 76, 101] support *multidimensional* data and allow users to filter based on different predicates that are not predefined. Multi-predicate queries are critical for some applications. For example, a doctor might want to run the following query to assess congestive heart failure risk without revealing the filter values or query result to the server [97]:

```
SELECT COUNT(*) FROM MedicalHistory
WHERE (systolic < 90 OR diastolic < 50 OR
weight_gain > 2 OR heart_rate < 40
OR heart_rate > 90) AND (time BETWEEN
2021:07:01:00:00 AND 2021:08:01:00:00)
```

The second limitation is that Timecrypt and Zeph reveal the query time interval to the server, which could be problematic for some applications. For example, if a doctor is querying for a patient's heart rate, the queried time period could reveal when the patient had a heart attack or started a new medication. To address the first limitation (functionality), we could leverage techniques from encrypted databases [41, 51, 80–82, 85, 93, 103, 113]. While many of these systems can support multi-predicate queries, they generally achieve good performance by permitting some leakage, which an attacker can exploit to learn information about the query and the database contents (e.g. the attacker could learn the patient's blood pressure) [23, 46–48, 56, 58, 61, 62, 64, 79, 84, 111].

Both oblivious RAM (ORAM) [45, 78] and general-purpose secure multiparty computation (MPC) [44, 108, 110] are natural tools for this problem. ORAM is suited to the trusted proxy setting (common in encrypted databases [80, 82, 85]), and MPC works in the distributed-trust setting where servers are deployed in different trust domains (the same setting used in Zeph [22]). Unfortunately, both are prohibitively expensive for the time-series setting. Storing a multidimensional tree in ORAM makes it possible to execute queries in polylogarithmic time, but appends are just as or more expensive than executing queries and require many round-trips, which results in poor throughput due to the append-intensive nature of time-series workloads (§VII). General-purpose MPC is also a poor fit, as existing tools require massive amounts of communication (§VII). In a distributed-trust setting in which servers are likely deployed in different clouds to minimize the chance that multiple servers are compromised, many round-trips and large bandwidth imply high latency and high monetary cost.

We present Waldo, an oblivious, maliciously secure time-series database that leverages distributed trust. Waldo provides:

- **Multi-predicate functionality:** Waldo provides two types of indices: one that supports additive aggregates (e.g. sum, count, mean, variance, standard deviation) based on multiple predicates (§IV) and another that supports arbitrary aggregates (e.g. max, min, top-k) over a time interval (§V). Prior work [21, 22] only supports aggregation over time.
- **Obliviousness with malicious security:** Waldo distributes trust to protect not only the data contents, but also the query

filter values and search access patterns (§III). Our design uses three servers and provides malicious security when at least two servers are honest.

- **Efficiency:** We implement and evaluate Waldo (§VI,§VII) on a set of 32-core machines. With features modulo 2^8, Waldo (specifically our index with multi-predicate support in §IV) runs a query with 8 range predicates for 2^{10} records in 0.22s, compared with 1.75s for an MPC baseline and 9.60s for an ORAM baseline, and 2^{20} records in 11.82s, compared with 45.72s for MPC and 16.70s for ORAM. The MPC baseline uses $9-82\times$ more bandwidth between servers than Waldo for 2^{10} to 2^{20} records, and the ORAM baseline uses $20-152\times$ more bandwidth between the client and server(s) than Waldo for 1-10 predicates. Waldo is also highly parallelizable.

A. Summary of techniques

As we show in §VII, ORAM and general-purpose MPC are poorly suited to the time-series database setting due to the many rounds of interaction (ORAM) and substantial communication overhead (MPC) they require. We design Waldo to overcome these shortcomings and be efficient when servers are in different trust domains: we need to rely less on communication, which is limited and expensive [1], and instead take advantage of compute resources, which are significantly cheaper and easy to increase. With this goal in mind, we turn to function secret sharing (FSS) [17, 18], a recent cryptographic tool that allows the client to generate compact shares of a function that the servers can then use to evaluate the corresponding equality or range predicate without learning what the predicate is (critical for security). Crucially, the servers can evaluate their shares of the predicate without interaction (in contrast, other state-of-the-art MPC techniques require interaction proportional to the number of comparisons).

At its core, FSS is a simple primitive designed for semihonest servers with public inputs where efficient implementations exist for a limited class of functions [17, 18]. The high-level challenge in Waldo is to adapt this fairly simple primitive to the much more complex encrypted time-series database setting where there are *malicious servers, secret inputs, and chained predicates*. Prior work has explored applying FSS in different settings that require some combination of private data, malicious security, and complex queries [16, 19, 20, 30, 33, 106], but as we discuss below (and in §IX), these techniques do not easily translate to our setting. These shortcomings motivate the techniques we develop in Waldo, which we summarize below.

FSS for private predicates (§IV-A). Using FSS to evaluate predicates on private data is not straightforward because, for correctness, servers evaluating function shares must provide the same input. Providing the input in plaintext is clearly a problem for implementing equality or range predicates where the server should not know the values being compared. To circumvent this problem, prior works on FSS for secure computation [16, 19] use additive masks to hide secret values, but as we discuss in §IV-A, this technique is highly inefficient in our setting, requiring client communication linear in the database size. To solve this problem, we develop a *shared one-hot index*, which hides the contents of the data while supporting high-throughput appends and private queries with FSS. The way in which we split our index across the servers is inspired by Bunn et al.'s distributed ORAM [20] that combines FSS with replicated secret sharing [10]. However, distributed ORAM only requires a block storage abstraction, whereas we need to evaluate different range and equality predicates on the database contents. We introduce new techniques that build on top of FSS and replicated secret sharing to obliviously evaluate predicates (§IV-A).

Combining multiple predicates (§IV-B). To support multi-predicate queries, we need a mechanism for efficiently combining the outputs of equality and range queries. There are two critical challenges here: (1) how to structure the outputs of the FSS evaluations so that they can be efficiently merged, and (2) how to perform the actual merging. To solve (1), we design our shared one-hot index such that the FSS evaluation output is a vector of zeroes and ones that can easily be combined with a vector for another predicate. Then, to address (2), we leverage the fact that our vectors are shared using replicated secret sharing to take advantage of existing communication-efficient techniques for semihonest 3-party honest-majority multiplication [10, 99]. Our techniques for combining predicates are effective for computing count, sum, and, by extension, mean, variance, and standard deviation.

Supporting complex aggregates (§V). The above protocol supports complex filtering, but a limited set of aggregates. To support more complex aggregates (e.g. max, min, top-k), we show how to build a *shared aggregate tree* that supports any user-defined aggregation function where the server does not have to know how values are aggregated. Like our shared one-hot index, our shared aggregate tree uses FSS and replicated secret sharing to hide the query and data. Our shared aggregate tree only supports aggregation over time, but notably, queries do not require any server interaction. Furthermore, server execution time is independent of the aggregation function.

Providing malicious security (§IV-C, §V). For both types of queries, we need to defend against a malicious adversary that might try to tamper with the query results. Some 3-party honest-majority MPC protocols can rely on replication coupled with cut-and-choose [42, 71] or triple sacrifice techniques [35], but these solutions don't work for us because we use 2-party FSS. We need to authenticate the results of the FSS evaluations in a way that is compatible with our techniques for combining outputs from multiple predicates. Our solution is inspired by Boyle et al.'s use of information-theoretic MACs [14, 27, 28] for authenticating FSS evaluation outputs [16]. The challenge is to make this approach compatible with multiplications: the multiplication protocol that we use for combining multiple predicate outputs is very efficient, but designed for the semihonest setting [10, 99]. We show how to use authenticated outputs from FSS evaluations to securely chain multiplications together such that the client only needs to check the integrity of the final result and a random linear combination of intermediate results.

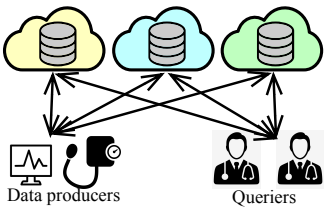


Fig. 1: System architecture. Here the ECG sensor and blood pressure monitor are data producers, and the doctors are queriers. The servers are deployed in different trust domains.

II. System overview

A. Time-series workloads

Waldo accounts for these elements of time-series workloads:

- **Append-only:** Time-series databases tend to be append-only, as records represent data captured at some timestamp [102].
- **Append-intensive:** Time-series workloads are primarily composed of appends, and so the database must be able to process a large volume of appends quickly [53, 72].
- **Multiple features, multiple predicates:** The data recorded for a timestamp often has multiple features. Therefore, queries with predicates corresponding to more than one feature are very common [54, 89].
- **Recent data is more valuable:** Even though the number of records grows rapidly over time (time-series workloads are append-only and append-intensive), the most recent data is the most relevant, and the value of data decays over time [53].
- **Aggregation:** Aggregation queries are very common in time-series workloads. Plaintext time-series databases build specialized indices to quickly aggregate data [8, 101]

B. Running example: Remote Patient Monitoring

Time-series data is critical to many applications, including smart homes [75], smart cars [95], energy conservation [96], agriculture [24], and industrial IoT [100]. We now discuss remote patient monitoring as a running example (although Waldo can support a wide variety of applications). Remote patient monitoring systems allow doctors to use sensors to monitor at-risk patients while they are home. The COVID-19 pandemic has made these tools even more critical, with more patients opting for telehealth visits and the federal government expanding Medicare coverage to remote patient monitoring [74].

RPM can be particularly valuable for at-risk patients to manage conditions such as hypertension, chronic obstructive pulmonary disease, diabetes, and asthma [86]. In some cases, doctors only need to monitor a single vital sign (e.g. glucose levels in a diabetes patient), but in a growing number of cases, doctors find it valuable to make decisions based on more biometric data (e.g. blood pressure, heart rate, and weight [98]).

One challenge for remote patient monitoring is that this data is extremely sensitive and the query itself can reveal information about a patient's condition. For example, the threshold vital signs that a doctor checks for may reveal if a patient is diabetic. Waldo ensures that the attacker only learns the database schema and the structure, origin, and timing of queries (§III).

C. System architecture

The Waldo system is composed of the following entities (Fig. 1):

- **Clients:** There are two types of clients: data producers and queriers. Some clients may be both.
  - **Data producers:** Sensors or other devices collect real-time data and update the servers' state.
  - **Queriers:** Queriers query the data collected by the data producers and stored at the servers.
- **Servers:** Three servers in different trust domains store data collected by data producers and execute queries made by queriers. If a majority of the servers are honest, the single malicious server cannot not learn the data contents, query filter values, or any search access patterns. These "logical" servers might be distributed across multiple machines.

Because Waldo leverages distributed trust (§III), each server should be deployed in a different trust domain. This could mean that the servers are hosted in different clouds, managed by different, potentially competing organizations, and/or deployed in different jurisdictions. Clients send messages directly to each of the three servers. This is in contrast to prior works that use a trusted proxy [80, 82, 85]: with a trusted proxy, users route their queries through a computationally powerful machine that interacts with the server on behalf of the clients. The proxy model has the disadvantage that the clients must set up a powerful, trusted-by-all machine rather than outsourcing computation to the cloud. In Waldo, both storage and computation are outsourced, and clients interact directly with the untrusted servers.

D. Waldo API

We now describe the API that clients use to interact with Waldo. Waldo exposes two types of indices to clients: WaldoTable and WaldoTree. WaldoTable stores multiple features for a single timestamp and supports multi-predicate queries. WaldoTree, on the other hand, stores a single feature for a timestamp and only supports queries over a time range. While WaldoTable supports more complex multi-predicate filtering, WaldoTree supports a larger class of aggregation functions and is more performant. WaldoTree is also useful for queries with predefined filters (as it is faster and uses less storage than WaldoTable), whereas WaldoTable is useful when queries are unpredictable. Both types of indices support Init, Append, and Query operations where Init and Append are invoked by data producers, and Query is invoked by the queriers. All routines trigger execution at the servers. We describe the API for both below.

WaldoTable**:**

- $\mathsf{Init}(1^\lambda, 1^{\tilde{s}}, \mathsf{schema})$: Initialize a table index given computational security parameter λ, statistical security parameter $\tilde{s}$, and a schema layout parameter $\mathsf{schema} = (N, F, 2^{\ell_1}, \ldots, 2^{\ell_F})$ where N is the number of records in the window, F is the number of features associated with a timestamp, and 2^{ℓ_i} is the feature size for feature i.
- $\mathsf{Append}(t, v_1, \ldots, v_F)$: Update the table index to store record with timestamp t and values $v_1 \in \mathbb{Z}_{2^{\ell_1}}, \ldots, v_F \in \mathbb{Z}_{2^{\ell_F}}$.
- $\mathsf{Query}(P_1 \wedge \cdots \wedge P_n, \mathsf{feature}, \mathsf{type}) \rightarrow x$: Aggregate by type for feature over the boolean formula composed of predicates $P_1, \ldots, P_n$, where each P_i implicitly conveys the feature it applies to. Here $\mathsf{type} \in$ {`count, sum, mean, variance, stdev`}. Output the

aggregate of values in feature filtered by the query predicates (or abort if integrity checks fail).

WaldoTree**:**

- $\mathsf{Init}(1^\lambda, 1^{\tilde{s}}, \mathsf{schema})$. Initialize a tree index given computational security parameter λ, statistical security parameter $\tilde{s}$, and a schema layout parameter $\mathsf{schema} = (2^\ell, \mathsf{type})$, where 2^ℓ is the feature size of the values in the tree index and type is any user-defined aggregation function (§V).
- $\mathsf{Append}(t, v)$: Update the tree index to store record with timestamp t and value $v \in \mathbb{Z}_{2^\ell}$.
- $\mathsf{Query}(t_1, t_2) \to x$: Aggregate over time interval (t_1, t_2) by type (set during initialization) and output the result (or abort if integrity checks fail).

In WaldoTable, the schema parameter takes a number of records in the window, N. Because records are constantly appended, N is not the total number of records in the table, but rather the number of most recent records that the client can query (recall that the most recent data is typically the most valuable). The parameter N represents a tradeoff between performance (smaller N values result in better performance) and query expressiveness (clients might want access to data further in the past). WaldoTable can easily be extended to support multiple window sizes N if the client is willing to reveal which window size is being used for the current query.

WaldoTable supports both equality ($x = a$) and range ($a < x < b$) predicates. Clients can chain predicates together using AND operations. The servers can easily compute NOTs (§IV-B), and so we can express any combination of ANDs, ORs, and NOTs via De Morgan's laws in a WaldoTable query.

Access control. Waldo enforces different permissions for clients across different tables. Because the servers know the identity of the client, access control is straightforward: all three servers must participate to compute a query, and we allow at most one server to be malicious, so we can restrict the types of queries that different users are allowed to make using a standard database access control list [31]. Each server checks a client's permissions, and if a client doesn't have permission for that operation, the honest servers will simply refuse to participate. Note that the servers can only enforce permissions for the parts of the query that are public (e.g. the data that the query is executing on and the query structure), but not the parts that remain private. Also, permission to make some types of queries (e.g. mean) implicitly gives permission to view some intermediate values (e.g. sum and count to compute mean). Access control can be at the record level in WaldoTable and at the database-table level in WaldoTree. For simplicity, when describing the design of Waldo, we focus on the case of a single table, as it is straightforward to extend this design to multiple tables with access control.

E. Notation

In Waldo, we consider all database values $\in \mathbb{Z}_{2^\ell}$ where ℓ depends on the feature size defined in schema. Waldo uses secret-sharing to split a value $x \in \mathbb{Z}_{2^\ell}$ into parts $[x]_1, \ldots, [x]_p \in \mathbb{Z}_{2^\ell}$ where $p \in \{2, 3\}$ such that $x = (\sum_{i=1}^{p} [x]_i) \bmod 2^\ell$. Note that we can sample shares in $\mathbb{Z}_{2^m}$ for $m \geq \ell$ to represent values in $\mathbb{Z}_{2^\ell}$. We sometimes use $x_1, \ldots, x_p$ and $x^{(1)}, \ldots, x^{(p)}$ to also refer to the secret shares of x, and within a server's context, we sometimes drop the share subscript altogether. All arithmetic operations such as $(+, -, \cdot)$ correspond to ring operations. Arithmetic operations on vectors refers to their component-wise application in the underlying ring. $[N]$ denotes the set $\{0, \ldots, N-1\}$. We use $a \leftarrow b$ to denote assignment of b's value to a, and $a \xleftarrow{R} \mathbb{R}$ denotes randomly sampling a from ring $\mathbb{R}$. We denote the computational security parameter as λ and the statistical security parameter as $\tilde{s}$.

III. Threat model and security guarantees

We describe Waldo's security guarantees and, due to space constraints, delegate Waldo's formalism (detailing its guarantees) to §A. Waldo operates in the malicious three-party honest-majority setting, meaning that it provides security with abort if at most one server is malicious. In the malicious threat model, the attacker can influence the server's behavior arbitrarily. If a server is malicious, the client does not receive output and only learns that an error occurred.

If at most one server is corrupted, then Waldo guarantees that the attacker does not learn the record contents, query filter values, or any search access patterns and only learns public information. The public information available to the attacker is: (1) the database schema (i.e. for each table, the number of records, number of features, and the size of each feature); (2) the structure of the query (i.e. the number of predicates, the type of each predicate, the structure of conjunctions and negations, the feature being aggregated, and, in the case of WaldoTable, the aggregation function); and (3) when a query is performed and which client performed the query. The predicate type includes whether the predicate is an equality or range (single-sided or interval) predicate and the feature corresponding to the predicate. To make the query structure leakage more concrete, we consider the congestive heart failure query in §I: the attacker learns that the query is computing `COUNT(*)` for `(RANGE(f1) OR RANGE(f2) OR RANGE(f3) OR RANGE(f4) OR RANGE(f5)) AND (RANGE(f6) AND RANGE(f6))` where `RANGE` implies a single-sided range predicate and the mapping of features to feature ID is consistent across queries. Expressing queries in terms of some query "normal form" with dummy predicates could eliminate leakage due to query structure, although this would negatively impact performance and query expressiveness. As discussed in §II-D, for some types of queries, clients are able to learn intermediate values (e.g. the client learns sum and count when running a mean query).

Notably, Waldo does not reveal any information about the filter values, which records are selected in a query, or how many records are selected, among other potential sources of leakage; protecting access patterns and volume leakage defends against a large class of leakage-abuse attacks [23, 46–48, 56, 58, 61, 62, 64, 79, 84, 111]. Because Waldo is maliciously secure, the client can check the integrity of the query result. If at most one server is corrupted, Waldo ensures that only clients granted permission to make queries or updates to a given table are able to perform those operations. Waldo does not provide

availability if any one server refuses to provide service.

We formally model the end-to-end security guarantees of Waldo by defining an ideal functionality $\mathcal{F}$ that specifies the behavior of an ideal system, capturing the properties discussed above. $\mathcal{F}$ additionally captures the fact that the client can verify the integrity of the result. In §A, we present a formal definition of security using $\mathcal{F}$, which we use in the following theorem:

Theorem 1: *Using Definition 1 (§A), Waldo securely evaluates (with abort) the ideal functionality $\mathcal{F}$ (§A) when instantiated with secure distributed point and comparison functions and a pseudo-random function, all with a computational security parameter of λ.*

We include the full proof in §A.

IV. Multi-predicate queries

In this section, we describe how to implement the WaldoTable API to filter on multiple predicates. We will start with a strawman that provides limited functionality and incomplete security and show how to modify our scheme to support the full query functionality and security guarantees we want.

A. Single predicate with semihonest security

Our first step is to choose a building block to help us obliviously filter by predicates. As we discussed previously, ORAM and generic MPC are natural candidates, but these solutions perform poorly in the time-series setting (§VII). We instead identified two-party function secret-sharing to be an excellent fit for equality and range predicates.

Tool: Function Secret Sharing (FSS). Two-party function secret sharing (FSS) makes it possible to split a function f into succinct function shares such that any strict subset of the shares doesn't reveal anything about the function f, but when the evaluations at a given point x are combined, the result is $f(x)$. A two-party FSS scheme is defined by the following algorithms:

- $\mathsf{Gen}(1^\lambda, f) \rightarrow K_1, K_2$: Given the security parameter 1^λ and a function description f, output keys K_1 and K_2.
- $\mathsf{Eval}(K_i, x) \rightarrow y_i$: Given the key K_i and input x, output value y_i, corresponding to this party's share of $f(x)$. We assume that key K_i implicitly contains the party index i.

Adding together the two outputs of Eval y_1, y_2 yields $f(x)$.

We identify two FSS constructions as a natural fit for Waldo: distributed point functions and distributed comparison functions [16–18]. Distributed point functions (DPFs) are FSS schemes for the point function $f_{\alpha,\beta}$ where $f_{\alpha,\beta}(\alpha) = \beta$ and $f_{\alpha,\beta}(\alpha') = 0$ for all $\alpha' \neq \alpha$ [17, 18]. Similarly distributed comparison functions (DCFs) are for functions $g_{\alpha,\beta}$ where $g_{\alpha,\beta}(x) = \beta$ if $x < \alpha$ and $g_{\alpha,\beta}(x) = 0$ otherwise [16]. Analogously, DCFs can also describe predicates $x > \alpha$. Constructions for interval containment (IC) build on DCFs to express functions of the form $a < x < b$ [19]. Throughout the paper, we will use $\mathsf{Gen}^=(1^\lambda, \alpha, \beta)$ and $\mathsf{Gen}^<(1^\lambda, \alpha, \beta)$ to refer to FSS generator algorithms for DPFs and DCFs respectively. For $a < x < b$ predicates, we use the IC construction from Boyle et al. [19] that requires 2 DCF keys per IC, and we refer to its generator as $\mathsf{Gen}^{\mathsf{IC}}(1^\lambda, a, b, \beta)$. For all these cases, we refer to the evaluation algorithms as Eval, and we assume the keys implicitly convey the type of algorithm being invoked.

DPFs are a natural fit for equality queries, and DCFs are a natural fit for range queries.

FSS for private data. Applying FSS to filter *public* data based on an equality or range predicate is fairly straightforward [16, 18, 106]. Two servers store identical copies of a public database (here the database is just a list of values). To privately query the database for the number of records matching a predicate, the client generates FSS keys with $\beta = 1$ for the equality or range predicate using a DPF or a DCF and sends a key to each server. Each server evaluates its key on each value in the database, sums the evaluations together, and sends the results back to the client, computing $\sum_{i=1}^{N} \mathsf{Eval}(K, d_i)$ for a database composed of values $d_i, \ldots, d_N$ with FSS key K on server s. When the client sums the results from each server, it obtains the number of records matching the predicate.

Leveraging FSS to search over *private* data, however, introduces a new challenge: the server cannot simply evaluate its FSS key on the database contents because the server should not be able to view the database contents. At the same time, the servers need to evaluate their keys on identical copies of the database to produce correct outputs. Prior works on using FSS for secure computation [16, 19] keep values secret by ensuring that the servers hold additively masked versions of the secret. To output shares of $f(x)$ instead of $f(x+r)$ (where f is the shared function, x is the secret input, and r is the mask), they rely on sharing the matching function $\tilde{f}_r = f(x-r)$. In the database setting, each entry x_i must be masked with an independently sampled r_i. Thus we would need a different $\tilde{f}_{r_i}$ for each database entry x_i even though the servers only need to evaluate a single function f. This practically means that the size of the function shares would match the size of the database, defeating the purpose of using FSS to minimize communication. Therefore, we need different techniques for the encrypted time-series database setting.

Our solution to this problem is inspired by that of Dory [30]. For each feature, we build a table of size $N \times 2^\ell$ where N is the number of records that can be queried (the window size from §II-D) and 2^ℓ is the feature size (i.e. the number of possible values for that feature). For each record, the corresponding row in the table is set to "1" at the location corresponding to record value and "0" elsewhere. We call this structure a *one-hot index*, as it is a table of "one-hot" vectors, and we use this tool as a building block to construct a *shared one-hot index*. While the construction of the core one-hot index is very similar to the data structure in Dory, the shared one-hot index we construct from it provides more powerful functionality and guarantees confidentiality and integrity using different techniques, as we discuss later.

Now we can leverage FSS using the *structure* rather than *content* of the search index. We want to compute the number of records matching the predicate. For every entry $d_{i,j}$ in the table D for record $i \in [N]$ and feature value $j \in [2^\ell]$, the server s evaluates its FSS key K on the current value j, multiplies

the evaluation by the table entry $d_{i,j}$, and then computes

$$\mathsf{EvalPred}(s, K, D) \leftarrow \sum_{j=0}^{2^\ell - 1} \left(\mathsf{Eval}(K, j) \cdot \sum_{i=0}^{N} d_{i,j} \right) \quad (1)$$

There are two remaining challenges here. First, we need to understand how to encode different types of record values using a small feature size 2^ℓ, as the computation required is $O(N \cdot 2^\ell)$. Second, while this clearly works if $d_{i,j} \in \{0, 1\}$, if the values $d_{i,j}$ are encrypted, then the summation will not produce the correct result. We address both below.

Encoding values with a small feature size. Choosing a small feature size 2^ℓ is critical for good performance in WaldoTable. For the remote patient monitoring applications we examine, we find that all sensitive fields with a predicate computed over them are already from a small domain (size 2^8) or can easily be mapped to one (§VII-B). Notably only the values being compared in predicates need to use small feature sizes; the values being aggregated are not subject to these restrictions. We summarize three techniques for encoding values in a large domain using a small feature size below.

One way to represent a large set of values using a small feature size is by bucketing intervals in $\mathbb{Z}_{2^\ell}$, improving performance at the expense of precision. Bucketing preserves ordering for range predicates and is used in prior work [6, 22, 26] to efficiently compute aggregate statistics. Candidates for bucketing include attributes such as weight, blood glucose level, salary, GPA, or percentages.

Hash maps or Bloom filters can compress large values for point queries where high precision is required (for Bloom filters, the client needs to check that each bit at each hash location is 1). One example of a field that can be represented in this way and is only used in point queries is an identifier (e.g. a client ID, company ID, social security number, or phone number).

Large domains can also be represented via a conjunction of predicates. For example, a time can be represented as a timestamp or as a conjunction of the year, month, day, hour, and minute. The number of predicates can leak information about the resulting filter, although this leakage can be eliminated by always using the maximum number of predicates.

Identifying replicated secret sharing (RSS). To solve the second problem (protecting the database contents while computing the correct sum), we turn to secret sharing. In standard 2-out-of-2 secret sharing, to secret share a value $x \in \mathbb{Z}_{2^k}$, we sample shares $[x]_1, [x]_2 \xleftarrow{R} \mathbb{Z}_{2^k}$ such that $x = [x]_1 + [x]_2$. Since the output of Eval is 2-out-of-2 shares of $f(x)$, if the table D is also shared in the same way, then each of the 2^ℓ multiplications in Equation (1) will require expensive MPC tools [13]. While 2^ℓ secure multiplications might seem feasible, to chain together predicates, we will show in §IV-B that we need to perform $N \cdot 2^\ell$ multiplications to evaluate a single predicate, which is impractical if these multiplications must use Beaver triples [13]. To overcome this challenge, we leverage replicated secret sharing, which hides the data contents while only requiring *local multiplications*. Replicated secret sharing requires switching from two servers to three servers, but this third server allows us to significantly improve performance. This combination of RSS with FSS makes single-depth multiplications essentially free (no communication required).

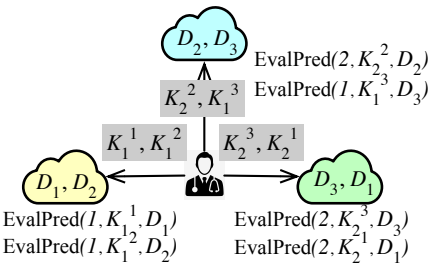


Fig. 2: Query predicate evaluation with FSS and RSS.

We use the 2-out-of-3 replicated secret sharing from Araki et al. [10]. To secret share a value $x \in \mathbb{Z}_{2^k}$, sample shares $[x]_1, [x]_2, [x]_3 \xleftarrow{R} \mathbb{Z}_{2^k}$ such that $x = [x]_1 + [x]_2 + [x]_3$. Each server gets a pair of shares: S_1 has $([x]_1, [x]_2)$, S_2 has $([x]_2, [x]_3)$, and S_3 has $([x]_3, [x]_1)$.

Layering RSS with FSS. RSS provides the replication necessary for FSS without sacrificing confidentiality. If the database is split into shares $[D]_1, [D]_2, [D]_3$, then the client can generate three pairs of FSS keys (K_1^1, K_2^1), (K_1^2, K_2^2), and (K_1^3, K_2^3). Each server's share of the database is a pair $(D.\mathsf{first}, D.\mathsf{second})$, where S_1 has $([D]_1, [D]_2)$, S_2 has $([D]_2, [D]_3)$, and S_3 has $([D]_3, [D]_1)$. The client sends S_1 (K_1^1, K_1^2), S_2 (K_2^2, K_1^3), and S_3 (K_2^3, K_2^1)). Then

- S_1 computes $x_1^{(1)} \leftarrow \mathsf{EvalPred}(1, K_1^1, [D]_1)$ and $x_1^{(2)} \leftarrow \mathsf{EvalPred}(1, K_1^2, [D]_2)$,
- S_2 computes $x_2^{(2)} \leftarrow \mathsf{EvalPred}(2, K_2^2, [D]_2)$ and $x_1^{(3)} \leftarrow \mathsf{EvalPred}(2, K_1^3, [D]_3)$, and
- S_3 computes $x_2^{(3)} \leftarrow \mathsf{EvalPred}(3, K_2^3, [D]_3)$ and $x_2^{(1)} \leftarrow \mathsf{EvalPred}(3, K_2^1, [D]_1)$.

Client can fetch these and compute $x^{(1)} \leftarrow x_1^{(1)} + x_2^{(1)}$, $x^{(2)} \leftarrow x_1^{(2)} + x_2^{(2)}$, $x^{(3)} \leftarrow x_1^{(3)} + x_2^{(3)}$, and $x \leftarrow x^{(1)} + x^{(2)} + x^{(3)}$. This way, RSS allows us to hide contents of the data from servers and evaluate a single predicate without communication between servers (Fig. 2). We call this data structure (and the corresponding API for appending to and querying it) a *shared one-hot index*.

Appends to this shared one-hot index are straightforward: to append value v_i corresponding to feature i with feature size 2^{ℓ_i}, the data producer generates a one-hot vector $V \in \mathbb{Z}_{2^k}^{2^{\ell_i}}$ where $V_j = 1$ if $j = v_i$ and $V_j = 0$ if $j \neq v_i$. Then, the data producer splits V component-wise into RSS shares and sends a pair of shares to each server. Each server appends each share to its corresponding table. Note that we do *not* support private updates to existing records, and so we only consider append-only workloads like time-series. Bunn et al. [20] also explore how FSS and RSS compliment each other in the distributed ORAM setting. They show how to provide a private key-value store interface, whereas we build a data structure that can handle more complex queries while only requiring a small number of FSS keys.

B. Multiple predicates with semihonest security

So far, we have focused on evaluating a single predicate, but our goal is to filter records based on a combination of multiple predicates. We have also focused only on counting the number of records matching a predicate, but in practice we additionally want to compute sums (below we describe how to use sum and count as a foundation for other aggregates).

To support both of these, we transition from each server computing a single value (the number of records matching the predicate) to each server computing a filter (a vector of size N where the value at index $i = 1$ if record $i \in [N]$ matches the predicate). Server s with FSS key K and table D with table entry $d_{i,j} \in \mathbb{Z}_{2^\ell}$ can compute

$$\mathsf{FilterPred}(s, K, D) \leftarrow \Big(\sum_{i=0}^{2^\ell - 1} \big(\mathsf{Eval}(K, i) \cdot d_{0,i}\big), \ldots, \sum_{i=0}^{2^\ell - 1} \big(\mathsf{Eval}(K, i) \cdot d_{N-1,i}\big) \Big) \quad (2)$$

From this filter, it is easy to count the number of matching records as before: simply sum the elements in the filter. Computing the sum of values for records matching the filter is a bit trickier, as we need to compute the dot product of the filter F and the values W, where W is a vector of database values corresponding to the feature being added (not in one-hot form), and both are secret-shared.

Combining filters using logical ANDs also requires multiplication. Given two filters F_1 and F_2, we can compute $F_1 \wedge F_2$ by multiplying F_1 and F_2 together because all the elements are in $\{0, 1\}$. The NOT operator can be easily computed locally (one pair of shares is set to $[x]_i \leftarrow 1 - [x]_i$ and the others are set to $[x]_i \leftarrow -[x]_i$), and the OR operator can be written as a combination of ANDs and NOTs. Thus the problems of aggregating by sum and combining filters both reduce to the problem of multiplying secret-shared values.

Tool: Multiplying RSS shares. Generic MPC tools for multiplication are particularly efficient in the three-party honest-majority setting. Given shares $x_i, x_{i+1}, y_i, y_{i+1}$, we can compute $z_i = x_i y_i + x_{i+1} y_i + x_i y_{i+1}$. Then $z_1 + z_2 + z_3 = xy$, yielding a 3-out-of-3 additive sharing. We refer to this existing technique as Mult, which takes in RSS shares of operands x, y and outputs 3-out-of-3 shares of xy. To obtain a replicated secret-sharing, S_i can send a blinded share $z_i + \alpha_i$ to S_{i+1} where $(\alpha_1, \alpha_2, \alpha_3)$ is a fresh secret-sharing of zero. To generate fresh sharings of zero, we rely on a pseudorandom function (PRF) keyed during initialization. During setup, server S_i samples PRF key k_i and sends k_i to server S_{i+1}. The jth share of zero is (z_1, z_2, z_3) where $z_i = \mathsf{PRF}(k_i, j) - \mathsf{PRF}(k_{i-1}, j)$. This general approach is used in CryptGPU [99], and the technique for generating fresh sharings of zero is from Araki et al. [10]. We refer to this technique for resharing 3-out-of-3 shares to get RSS shares as $\mathsf{Reshare}$, and Mult followed by $\mathsf{Reshare}$ as $\mathsf{MultAndReshare}$, which takes RSS shares of x, y and outputs RSS shares of xy.

Supporting multiple predicates. This multiplication tool simplifies the problem of summation and combining multiple filters. The local computation costs are small, but now each server must send N values where N is the number of records for each multiplication. Generating filters using FSS with RSS produces filters that are 3-out-of-3 shared, and so before multiplying them, we use the same share conversion trick to get replicated secret shares. To reduce the cost of share conversion, we can precompute the PRF evaluations and store them until we receive a query, making the online cost of multiplication and resharing depend almost entirely on the cost of communication.

Supporting different types of aggregates. So far, we have described how to compute count (summing elements in the filter F) and sum (computing the dot product of F and the values W). We now discuss how to use these building blocks to compute other functions using previously known techniques. A well-known technique for computing the mean is to run a sum and a count query and divide locally. By storing not just the value X but also X^2, it is also straightforward to compute the variance of X as $\mathsf{Var}(X) = E[X^2] - (E[X])^2$ using the above technique to compute the mean and then squaring and subtracting locally. For standard deviation, the client locally computes the square root of variance. When using the same filter with different aggregation functions, we can reuse the filter to save computation.

C. Multiple predicates with malicious security

Up to this point, we have only considered a semihonest adversary, but to defend against a malicious adversary, the client needs to be able to check that the servers performed the computation correctly. We leverage information-theoretic MACs from MPC [27] and show how to provide integrity when chaining together predicates evaluated using FSS.

Tool: Information-theoretic MACs. To authenticate a value x in the ring $\mathbb{Z}_{2^k}$, we use the information-theoretic MACs from $\mathrm{SPD}\mathbb{Z}_{2^k}$ [27]. The servers hold shares of x over the ring $\mathbb{Z}_{2^{k+s}}$ where s is the statistical security parameter. For some MAC key $\alpha \in \mathbb{Z}_{2^{k+s}}$ not known to the servers, the servers also hold shares of $\alpha x \in \mathbb{Z}_{2^{k+s}}$. These MACs are additively homomorphic: $\alpha x_1 + \alpha x_2 = \alpha(x_1 + x_2)$. All computation is now performed in $\mathbb{Z}_{2^{k+s}}$ over both the value and the MAC, and the protocol aborts if the output y and the MAC tag σ do not satisfy $\alpha y = \sigma$. The probability that the attacker can forge the MAC is the probability that the attacker can guess s lower bits of α, which is $1/2^s$. We choose to use rings rather than fields even though this means that we need a larger ring because it allows us to take advantage of native instructions for addition and multiplication in our implementation.

Encoding information-theoretic MACs. We would like to apply information-theoretic MACs to Waldo such that the client chooses a value $\alpha \xleftarrow{R} \mathbb{Z}_{2^{k+s}}$ and then receives a query result of the form $(x, \sigma = \alpha x)$. Existing RSS-based 3PC works in the honest-majority setting provide malicious security for multiplications by replication combined with either cut-and-choose techniques [42, 71] or triple sacrifice [35]. Our setting is different because we are interleaving 2-party FSS with 3-party RSS. Using 2-party FSS requires us to use dishonest-majority techniques to provide malicious security, and information-theoretic MACs are a natural candidate here [14, 16, 27, 28]. Because we use MACs to authenticate the FSS evaluations,

we can take advantage of authenticated input shares to provide malicious security when we combine filters via multiplication.

To authenticate multiplications, we use linear combinations of all intermediate values x_i and their MAC tags σ_i with random coefficients χ_i, i.e., $\sum \chi_i x_i$ and $\sum \chi_i \sigma_i$ where $\sigma_i = \alpha x_i$ [27]. This technique allows us to safely compute the MAC tag for xy by multiplying $\sigma = \alpha x$ directly with y because the batch check via random linear combinations ensures that xy is bound to the resulting MAC tag. The servers can compute shares of a random value χ_i using the same technique with PRFs that we use to generate random sharings of zero [10]; we refer to this as RandCoeff, which outputs RSS shares of random values and takes as input server index $\in \{1, 2, 3\}$ and the number of shares needed. With RSS shares of χ_i values, servers invoke the Mult function with RSS shares of χ_i and x_i to get 3-out-of-3 shares of $\chi_i x_i$ and similarly for $\chi_i \sigma_i$. We refer to this function as RandComb. Prior MPC work that computes random linear combinations of MACs requires the servers to perform a two-round commit-and-open protocol [27], but we can avoid this cost by taking advantage of the client. Each server simply sends the client its share of $y = \sum \chi_i x_i$ and $z = \sum \chi_i \sigma_i$ and the client assembles the shares and checks that $\alpha y = z$. Prior work [27] shows that this check catches any introduced errors with high probability; however, we only achieve $\tilde{s} = s - \log(s+1)$ parametric statistical security (see Lemma 2).

The only remaining challenge is how to encode α. A natural choice would be to keep two versions of the database where if the first version contains x, the second version contains αx. Then, the servers execute queries on both versions. This solution is secure but requires twice the amount of storage and makes revocation challenging: every client knows α, so if access is revoked, the *entire* table must be rebuilt with a new α'.

Instead, in our approach, the client chooses an α value for every query and encodes α *in the FSS key itself*. Instead of sending one key per predicate, the client now sends two FSS keys per predicate: one that evaluates to 1 on the desired input (as in the semihonest case), and another that evaluates to α on the desired input. The servers execute the query for the keys that evaluate to 1 to produce x and then for the keys that evaluate to α to produce σ, and the client checks that $\alpha x = \sigma$. This has two key benefits: (1) we do not need to expand the size of the index for malicious security, and (2) clients do not need to share α values (no recomputation necessary when access is revoked). Boyle et al. [16] first explored the idea of supporting malicious security by encoding α values directly in FSS keys, but for the MPC setting rather than the client-server setting in databases. A key difference between these settings is that, in the MPC setting, the parties must perform a joint verification protocol, whereas in our setting, the client can verify the MAC directly from the shares produced by each server.

D. Putting it together

We present the final protocol for the client in Fig. 3 and the server in Fig. 4 and provide a high-level overview below.

In Fig. 3, the client begins by sampling the MAC key α. For each predicate P_i, the client samples three pairs of FSS keys, which evaluate to shares of 1 for inputs satisfying P_i. In addition, for malicious security, the client samples three extra key pairs, which evaluate to shares of α when P_i is satisfied. The client sends the keys corresponding to the servers' RSS shares of the data, along with the predicate feature IDs, the ID of the feature being aggregated, and the type of aggregation (e.g. sum, count). Once the client receives responses from the servers, it reconstructs the query result x and the MAC tag σ, as well as the value $\hat{m}$ and its MAC tag $\hat{\sigma}$ that contain traces of malicious behavior via a random linear combination of the entire transcript. The client can then verify the MAC tags and output x if the checks pass.

In Fig. 4, the servers receive FSS keys corresponding to the query result and the query MAC tag for each predicate and each RSS share of the data. For each predicate in the query, the servers need to compute RSS shares of the intermediate filter and its corresponding MAC tag. To do this, they evaluate each FSS key on the corresponding table share to generate a 1-out-of-6 share of the resulting filter (FilterPred from §IV-B). Each server has RSS shares of the table and receives 2 FSS keys to evaluate each predicate, so it generates two 1-out-of-6 shares of the predicate filter, which it can then combine into a single 1-out-of-3 share. By running the Reshare protocol, the servers can convert their 1-out-of-3 shares to RSS shares. Then the servers run MultAndReshare (§IV-B) to combine predicates together and output an RSS sharing of the accumulated filter. They can then use shares of the final accumulated filter to compute shares of the final aggregate. By performing this process for both the FSS keys for the query result and the FSS keys for the MAC tag, the servers can compute shares of both the query result and the MAC tag. To ensure that the malicious server does not manipulate the filter shares or corresponding MAC tags during multiplication, the servers must compute a random linear combination of all the messages they received using RandCoeff (§IV-C). The servers send back shares of the final result x and the corresponding MAC tag σ, as well as the accumulated random linear combination $\hat{m}$ and its corresponding MAC tag $\hat{\sigma}$.

V. Complex aggregates over time ranges

While our shared one-hot index can compute a useful set of aggregates using sum and count queries, not all valuable aggregates can be expressed as a combination of dot products (e.g. min, max, top-k). In many cases, the client needs to compute a complex aggregate over a time period (e.g. a doctor might want to compute the maximum glucose level of a diabetic patient in the last week). Our WaldoTree index allows the client to compute any aggregate function over a time period without server-server interaction and without revealing the time interval being queried (as prior work does [21, 22]). Because it is more efficient than WaldoTable and doesn't require interaction between the servers, WaldoTree is also valuable in cases where the query predicates are predefined.

We call our solution to this problem a *shared aggregate tree*. Our core data structure is inspired by ideas from authenticated data structures [65], Faber et al. [38], Arx [82], and Timecrypt [21]: each leaf node contains a (private) record value, and

Client.WaldoTable.Query$^{S_1,S_2,S_3}(P_1 \wedge \cdots \wedge P_n, p, \text{type})$

$\alpha \xleftarrow{R} \mathbb{Z}_{2^{k+s}}, \mathcal{K}_i, \mathcal{K}'_i \leftarrow \{\}$ for $i \in \{1,2,3\}$
$\forall i \in \{1,\ldots,n\}$, $p_i \leftarrow \mathsf{idx}(P_i)$
for $i = 1$ to n **do**
  for $j = 1$ to 3 **do**
    $(K_1^j)_i, (K_2^j)_i \leftarrow \mathsf{Gen}(1^\lambda, P_i, 1)$
    $(K_1'^j)_i, (K_2'^j)_i \leftarrow \mathsf{Gen}(1^\lambda, P_i, \alpha)$
  end for
  $\mathcal{K}_1 \leftarrow \mathcal{K}_1 \bigcup (K_1^1)_i, (K_1^2)_i$ and $\mathcal{K}'_1 \leftarrow \mathcal{K}'_1 \bigcup (K_1'^1)_i, (K_1'^2)_i$
  $\mathcal{K}_2 \leftarrow \mathcal{K}_2 \bigcup (K_2^2)_i, (K_1^3)_i$ and $\mathcal{K}'_2 \leftarrow \mathcal{K}'_2 \bigcup (K_2'^2)_i, (K_1'^3)_i$
  $\mathcal{K}_3 \leftarrow \mathcal{K}_3 \bigcup (K_2^3)_i, (K_2^1)_i$ and $\mathcal{K}'_3 \leftarrow \mathcal{K}'_3 \bigcup (K_2'^3)_i, (K_2'^1)_i$
end for
for $i = 1$ to 3 **do**
  $(x_i, \sigma_i, \hat{m}_i, \hat{\sigma}_i) \leftarrow S_i.\mathsf{WaldoTable.Query}(\mathcal{K}_i, \mathcal{K}'_i, \{p_j\}_j, p, \text{type})$
end for
$x \leftarrow \sum_{i=1}^3 x_i, \sigma \leftarrow \sum_{i=1}^3 \sigma_i, \hat{m} \leftarrow \sum_{i=1}^3 \hat{m}_i, \hat{\sigma} \leftarrow \sum_{i=1}^3 \hat{\sigma}_i$
if $(\alpha \cdot x \neq \sigma) \vee (\alpha \cdot \hat{m} \neq \hat{\sigma})$ **then**
  Output $\perp$ and broadcast $\perp$ to all servers
end if
Output x

Fig. 3: Client WaldoTable.Query algorithm. $\mathsf{Gen}(1^\lambda, P_i, \beta)$ refers to $\mathsf{Gen}^=(1^\lambda, a, \beta)$, $\mathsf{Gen}^<(1^\lambda, a, \beta)$, or $\mathsf{Gen}^{\mathsf{IC}}(1^\lambda, a, b, \beta)$ depending on the predicate P_i being $x = a, x < a$, or $a < x < b$, respectively. We denote P_i's feature ID as $\mathsf{idx}(P_i)$, and $\{p_i\}_i$ denotes $\{p_1, \ldots, p_n\}$.

Server.WaldoTable.Query$^{S_1,S_2,S_3}(\mathcal{K}, \mathcal{K}', \{p_i\}_i, p, \text{type})$

Parse $\mathcal{K}$ as $(K_1)_1, (K_2)_1, \ldots, (K_1)_n, (K_2)_n$
Parse $\mathcal{K}'$ as $(K'_1)_1, (K'_2)_1, \ldots, (K'_1)_n, (K'_2)_n$
$\hat{m} \leftarrow 0$, $\hat{\sigma} \leftarrow 0$, $z_1 \leftarrow 1$ if $\mathsf{ID} = 2$ and 0 otherwise, and $z_2 \leftarrow 1$ if $\mathsf{ID} = 1$ and 0 otherwise.
$\tilde{F} \leftarrow (z_1^N, z_2^N), \tilde{F}' \leftarrow (z_1^N, z_2^N) \in \mathbb{Z}_{2^{k+s}}^N \times \mathbb{Z}_{2^{k+s}}^N$.
for $i = 1$ to n **do**
  $G_1 \leftarrow \mathsf{FilterPred}(\mathsf{ID}, (K_1)_i, T_{p_i}.\mathsf{first})$
  $G_2 \leftarrow \mathsf{FilterPred}(\mathsf{ID}, (K_2)_i, T_{p_i}.\mathsf{second})$
  $G'_1 \leftarrow \mathsf{FilterPred}(\mathsf{ID}, (K'_1)_i, T_{p_i}.\mathsf{first})$
  $G'_2 \leftarrow \mathsf{FilterPred}(\mathsf{ID}, (K'_2)_i, T_{p_i}.\mathsf{second})$
  $G \leftarrow G_1 + G_2$ and $G' \leftarrow G'_1 + G'_2$
  $\tilde{H} \leftarrow \mathsf{Reshare}^{S_1,S_2,S_3}(G)$ and $\tilde{H}' \leftarrow \mathsf{Reshare}^{S_1,S_2,S_3}(G')$
  $\tilde{F} \leftarrow \mathsf{MultAndReshare}^{S_1,S_2,S_3}(\tilde{F}, \tilde{H})$
  $\tilde{F}' \leftarrow \mathsf{MultAndReshare}^{S_1,S_2,S_3}(\tilde{F}', \tilde{H})$
  $\{\tilde{\chi}_1, \ldots, \tilde{\chi}_{2N}\} \leftarrow \mathsf{RandCoeff}(\mathsf{ID}, 2N)$
  $\hat{m} \leftarrow \hat{m} + \mathsf{RandComb}(\tilde{F}||\tilde{H}, \{\tilde{\chi}_1, \ldots, \tilde{\chi}_{2N}\})$
  $\hat{\sigma} \leftarrow \hat{\sigma} + \mathsf{RandComb}(\tilde{F}'||\tilde{H}', \{\tilde{\chi}_1, \ldots, \tilde{\chi}_{2N}\})$
end for
if type is `sum` **then**
  $x \leftarrow \sum_{i=1}^N \mathsf{Mult}(\tilde{F}_i, (D_p)_i)$, $\sigma \leftarrow \sum_{i=1}^N \mathsf{Mult}(\tilde{F}'_i, (D_p)_i)$
end if
if type is `count` **then**
  $x \leftarrow \sum_{i=1}^N \tilde{F}_i.\mathsf{first}$, $\sigma \leftarrow \sum_{i=1}^N \tilde{F}'_i.\mathsf{first}$
end if
Output $(x, \sigma, \hat{m}, \hat{\sigma})$

Fig. 4: Server WaldoTable.Query algorithm. We use N for the window size, n for the number of query predicates, and $\mathsf{ID} \in \{1,2,3\}$ for the server id. Variables with "tilde" denote RSS shares. The variables D, T refer to RSS shares of database and its corresponding shared one-hot index, respectively. We use $(D_p)_i$ to denote the ith record's pth feature in D (not in one-hot form), T_{p_i} refers to the shared one-hot index for p_ith feature, and $\tilde{F}_i$ denotes ith RSS share in $\tilde{F}$. We use $\mathsf{first}, \mathsf{second}$ to access the respective component from an RSS share. For simplicity we assume that predicates are chained via conjunctions; disjunctions can be expressed by adding negations.

each internal node contains the (private) aggregate of its two children. Each leaf node has a public timestamp, and the n leaf nodes are ordered by time so that each internal node has a public time interval. In this way, the client can compute an aggregate over some time interval by retrieving at most $2\log n + 1$ nodes. This set of nodes represents the *covering set*, as it covers the time range that the client is querying (see Fig. 5). Once the client has retrieved the nodes in the covering set, the client can locally aggregate the intermediate aggregates to compute the query result. To hide the contents of the tree from the server, we can again use RSS to share the aggregate value of each node.

Oblivious queries. The client cannot directly request the covering set from the server, as this set reveals to the server the time period being queried. To hide this set, we can once again leverage FSS. We use the same techniques for combining FSS with RSS discussed in §IV-A and so do not describe the interplay between FSS and RSS. As a strawman, the client could send $2\log n + 1$ logical DPF keys to the servers (here we refer to each "logical" FSS key as corresponding to 3 "physical" FSS keys, one for each of the 3 pairs of servers), each of which corresponds to a node in the covering set. Note that this strawman solution requires the timestamps at leaf nodes to be at regular intervals (we fix this issue in our final solution). To prevent query leakage, clients always need to send $2\log n + 1$ keys to the servers.

We can reduce the number of FSS keys that the client needs to send to just two logical keys per server pair by instead using DCFs for the two intervals $a < x$ and $x < b$. The client generates the DCF keys for the leaf level of the tree, sends the keys to the server, and then each server evaluates its DCF keys on the timestamp for each leaf separately for both $a < x$ and $x < b$. We say that a leaf node is "activated" if its DCF evaluation is a

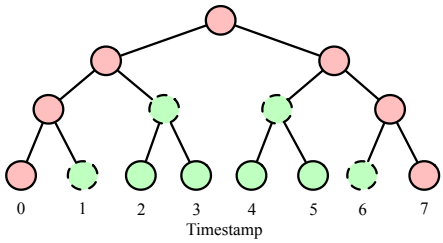


Fig. 5: Node activation for the query $0 < x < 7$. Green nodes are activated, and nodes with dashed boundary are in the covering set.

secret-share of one, meaning that the node is within the range that the client is querying for (note that to protect the client's query, the server does not know whether or not a node is activated). Each server then *projects* the DCF evaluation for each single-sided range at the leaf nodes to the internal nodes. Projection maintains the invariant that an internal node is "activated" only if both of its children are. We can perform this projection by copying the value of the left or right child to the parent depending on the direction of the single-sided range being evaluated. For example, in Fig. 5, the value of node 0 is copied to the parent of node 0 and node 1 (as this is the left side of the range), and the value of node 7 is copied to the parent of node 6 and node 7 (as this is the right side of the range). This projection operation allows us to correctly copy the DCF evaluations (se-

cret shares of the activation status) from the leaf nodes to the internal nodes (1) without knowing which nodes are actually activated, while (2) maintaining the invariant that a node is only activated if its time interval covers part of the queried time range.

However, we only want to retrieve the nodes in the covering set (i.e. the nodes where the parent is not also activated); we can't retrieve *all* activated nodes because the number of total activated nodes is large and depends on the queried range. To filter out nodes where the parents are also activated, we compute two sums for each level ℓ: (1) the sum of all the activated nodes X_ℓ, and (2) the sum of the *children* of the activated nodes Y_ℓ. Then for each level ℓ, we return $X_\ell - Y_{\ell-1}$. This ensures that we return at most one node per level (we compute each single-sided range separately). At the end of the protocol, the client receives $\log n$ values for each of the single-sided ranges, which the client can then use to recover the covering set for the intersection of the two ranges.

High-throughput appends. Because the leaves are ordered by time, appends are fairly straightforward. Nothing about the append is unpredictable or secret except for the value that the client is appending, and so the server simply sends the path from the root to the right-most leaf node (the tree will populate leaves moving to the right). The client uses these nodes to compute a path that incorporates the new value being appended at the internal node and aggregates and sends this path (secret-shared) to the servers. The servers use this path to update the tree to incorporate the new node. To keep the tree balanced, the servers can periodically rotate the tree (if all servers rotate the tree in the same way, the RSS sharings remain correct). Because access control is only at the table level for WaldoTree, clients can view the aggregates in the upper levels of the tree without a problem. Like WaldoTable, WaldoTree does not support privates updates to existing records (this functionality would require privately writing an arbitrary path in the tree).

One way to reduce the append overhead for resource-constrained data producers is to batch appends: the client retrieves the path along the right edge of the tree and then sends back the nodes for paths for the new values in one round trip. This greatly reduces not only round trips, but also total bandwidth because (1) only one path needs to be fetched, and (2) many of the new paths sent to the servers overlap.

Malicious security. Malicious security for our shared aggregate tree is straightforward. As in the shared one-hot index, we use information-theoretic MACs for queries by encoding them directly into the FSS key, as initially proposed by Boyle et al. [16]. For our appends, we need to ensure that the servers send the correct version of the right-most path in the tree. Here, we can rely on the fact that at most one server is malicious and each secret share is stored at two servers: if a pair of shares don't match, the client knows a server is corrupt.

Summarizing old data. We need to ensure that the tree size (and query execution time) stays bounded as the number of records increases. Our approach is inspired by Timecrypt [21]. When our tree reaches a maximum size, we rotate the tree, causing the left-most leaves to exceed the maximum depth. We then prune these leaves, leaving previously internal nodes as leaf nodes to summarize the pruned data. The client can no longer make fine-grained queries over old data, but can make coarse-grained queries that include this old data. Summarizing older data is common in modern time-series databases [52, 101].

Aggregation functions. The aggregation function does *not* need to be based on addition and can include min, max, top-k, bottom-k, histograms, and quantiles (some functions, like top-k, require storing multiple aggregate values per node). Our shared aggregate tree can also in principle support sketch algorithms [70], as well as aggregation-based encodings that allow private training of linear models [26, 57]. Notably, aggregation functions with the same output size will require the same amount of server execution time.

Final protocol. We present the final protocol for the client in Fig. 12 and the server in Fig. 13 (included in the appendix).

VI. Implementation

We implemented Waldo in ~6,200 lines of C/C++ code (excluding tests and benchmarking infrastructure). We used the libPSI [3] DPF implementation (with some minor modifications), the cryptoTools library [2] for cryptographic primitives, and gRPC for communication. We configured Waldo to aggregate values of up to size 2^{32} and set our statistical security parameter $\tilde{s} = 80$ and computational security parameter $\lambda = 128$. This allows us to use a 128-bit ring, which makes the additions and multiplications used to evaluate predicates very fast. Our implementation is available at https://github.com/ucbrise/waldo.

EvalAll for DCFs. We extend the state-of-the-art DCF construction from Boyle et al. [16] to perform full domain evaluations more efficiently. Waldo's protocols rely on evaluating FSS key K on each point in the domain of size 2^ℓ, referred to as EvalAll(K) [18]. Boyle et al. [18] proposed an EvalAll optimization for DPFs that reduces pseudorandom generator (PRG) invocations by a factor of ℓ. We apply this same insight to DCFs, providing the first EvalAll implementation for DCFs that we are aware of. This optimization improves single-threaded execution time for DCF EvalAll by 7.5× for $\ell = 20$.

Parallelism. We parallelize most of the query computation in WaldoTable and WaldoTree across 32 threads. Waldo achieves parallelism both within and across the evaluation of predicates. We can additionally parallelize the PRF evaluations for share conversion and malicious security.

VII. Evaluation

In evaluating Waldo, we ask the following questions:

1) How does the performance of Waldo compare to that of an ORAM and generic MPC baseline in terms of latency (§VII-C, §VII-D), throughput (§VII-E), bandwidth (§VII-F), and monetary cost (§VII-G)?
2) How do the individual components of Waldo perform, and how are they affected by different parameter settings (§VII-C, §VII-D)?

Experimental setup. We evaluate Waldo on AWS EC2 instances. For the servers, we use r5n.16xlarge instances

with 32 physical cores and 512 GB memory running on a 3.1 GHz Intel Xeon scalable processor. For the client, we use an r4.2xlarge instance with 4 physical cores and 61 GB memory running on a 2.3 GHz Intel Xeon E5-2686 v4 processor. To model the cost of transferring data between trust domains where the servers are geographically close but located in different data centers, server machines have a network bandwidth of 3 Gbps (max bandwidth to an external IP address in Google Cloud [25]) with 20ms RTT (the ping time we measured between AWS regions `us-west-1` and `us-west-2`).

A. Baselines

We now describe the two baselines that we compare Waldo to: a multidimensional tree in ORAM, and our functionality executed in a generic MPC framework. We do not compare against Timecrypt [21] or Zeph [22] because they do not support multi-predicate queries and provide less security (only semi-honest security and they leak the query).

Oblivious multidimensional tree. Prior work shows how to achieve obliviousness for multidimensional queries by layering oblivious tools like private information retrieval with a multidimensional tree [43, 60]. Because we need to support both searches and updates, we store a multidimensional tree in ORAM. We use an R-tree [49] because it handles updates well (k-d trees cannot be rebalanced [15]). However, R-trees are poorly suited to oblivious exact query execution (prior work uses them for approximate queries). While the average-case search complexity is logarithmic in tree size, the worst-case search complexity is linear. Our baseline does not pad the number of node accesses to the worst-case (this is impractical), and so its security guarantees are weaker than Waldo's. An interesting direction for future work would be to design a multidimensional tree for time-series data compatible with ORAM.

We implement our oblivious R-tree by taking an existing R-tree implementation [49] and replacing reads and writes to nodes in local memory with ORAM accesses. We use SEAL-ORAM's implementation of PathORAM [5], which relies on MongoDB for block storage. Because Waldo uses an in-memory index, we configure MongoDB to use a memory-mapped file. A full-fledged implementation would use techniques from oblivious data structures (ODS) [107] to encode data about the position map in the R-tree itself to minimize client local storage. For simplicity, we store the entire position map locally, as using ODS techniques would likely only add overhead (at the bare minimum, we would need to keep pointers in ORAM blocks). We use random data and random predicate values for our ORAM baseline, which adds a small amount of noise to our experiments (in contrast, Waldo and our MP-SPDZ baseline are fully oblivious, and so their performance is not affected by the data or query values). Point and range queries are executed in the same way, and we set tree dimension to the number of query predicates (for WaldoTable, the dimension is 1).

MP-SPDZ. For the MPC baseline, we implement Waldo's functionality in MP-SPDZ [4, 59], a state-of-the-art framework for general-purpose MPC. For WaldoTable queries, servers first check which records match the predicate(s), select the values for the matching records, and then aggregate. For WaldoTree queries, to give our baseline the advantage, we assume that the client encodes the query as indices of nodes in the covering set (padded to the worst-case size). Servers securely select nodes from secret shares of these indices and then aggregate by depth.

MP-SPDZ offers implementations of many different 3-party honest-majority protocols. We tested each and found that the post-processing variant of the RSS-based maliciously secure protocol from Eerikson et al. [35] was best for our setting. We used the library's mixed-mode circuit support, as well as loop decorators to parallelize computation and reduce the number of round trips. Values are aggregated modulo 2^{32} with 80-bit statistical security. All comparisons and equality checks are done only over ℓ bits for feature size of 2^{ℓ}, as in Waldo. For simplicity, we only implement the server processing and so only report server execution time (the client only has to submit a query).

B. Queries for real-world applications

We measure the performance of WaldoTable by evaluating sum queries with conjunctions of 2, 4, and 8 point and range predicates. A count query is a slightly cheaper version of the sum query (does not include a final multiplication by the values being summed), and mean, standard deviation, and variance are simply combinations of sum and count queries. For simplicity, we measure the case where all predicates are either point or range predicates, although real queries would likely contain a mix. Because Waldo is oblivious, performance does not depend on the query values or data contents. However, to make our results more concrete, we describe real-world applications where doctors need to examine relationships between measurements that correspond to two, four, and eight predicates. Throughout our evaluation, we use feature size 2^8 as it supports the applications we describe below (to the best of our knowledge as we are not medical experts).

Queries with two predicates can be used to compute the number of times an asthma patient's peak exipatory flow rate exceeded some patient-specific threshold in the last week [87]. Queries with four predicates can help identify at-risk pregnancies: doctors need to check for elevated blood pressure (`systolic < 120 AND diastolic < 80`) and sudden weight change over a time period [69]. Queries with eight predicates are useful for predicting heart failure decompensation: the success of the HeartLogic index has shown that the relationship between the first and third heart sounds, intrathoracic impedance, respiration rate, the ratio of respiration rate to tidal volume, heart rate, and patient activity over a time period can help identify at-risk heart failure patients [91].

WaldoTree queries can be used to compute a patient's maximum or minimum heart rate (for patients with heart conditions) or maximum or minimum glucose levels (for diabetic patients) over some time period. The execution time is independent of the time interval and aggregation function (we do not measure the time for the client to aggregate nodes in the covering set, as this should be very fast).

		WaldoTable latency (s)					
		– *Point* –			– *Range* –		
	log N	$P=2$	$P=4$	$P=8$	$P=2$	$P=4$	$P=8$
Malicious	10	0.08	0.13	0.23	0.07	0.12	0.22
	12	0.09	0.14	0.24	0.08	0.13	0.24
	14	0.11	0.18	0.31	0.10	0.17	0.32
	16	0.22	0.40	0.79	0.21	0.39	0.77
	18	0.78	1.53	3.11	0.80	1.56	3.03
	20	3.10	6.00	11.94	3.03	6.18	11.82
Semi-honest	10	0.07	0.12	0.21	0.07	0.11	0.21
	12	0.08	0.12	0.22	0.07	0.12	0.21
	14	0.09	0.14	0.23	0.08	0.13	0.23
	16	0.11	0.16	0.28	0.10	0.15	0.27
	18	0.19	0.28	0.49	0.18	0.27	0.47
	20	0.57	0.94	1.70	0.55	0.90	1.65

TABLE 6: WaldoTable query latency for P predicates and N records.

C. *Latency:* WaldoTable

Understanding Waldo's performance. In Table 6, we show WaldoTable query latency for different numbers of records N and different numbers of predicates P. As expected in Waldo, after a certain point ($N = 2^{16}$), the latency grows linearly with N and P, as both computation and communication costs are $O(NP)$ (fixed ℓ). Range and point predicates perform very similarly (small performance differences are due to implementation differences). For larger values of N, the overhead of malicious security is $6-7\times$ that of semihonest security: we can use 32-bit integers rather than 128-bit integers if we only need semihonest security ($4\times$ saving), and we don't need MACs ($2\times$ saving).

Fig. 7 illustrates the breakdown in query execution time for 2^{10} and 2^{20} records with different numbers of predicates (P). The majority of the overhead is due to network latency, particularly for smaller N and larger P. This is due to the fact that the number of Waldo round-trips is linear in P. For larger numbers of records, the computation and the bandwidth increase, but the number of round trips does not, and so the ratio of compute time to network time increases. Note that the PRF evaluation time (for re-sharing after multiplications and random linear combinations of MACs) could be moved into a separate preprocessing step to reduce online latency.

Comparison to baselines. In Fig. 8a, we show how WaldoTable's query latency compares to that of the two baselines for different numbers of records N with 8 predicates. The latency of WaldoTable and MP-SPDZ increase at roughly the same rate, as expected, as both incur costs linear in N. Although they grow at similar rates, WaldoTable remains substantially faster than MP-SPDZ, $7.8\times$ for 2^{10} records for both point and range queries, and for 2^{20} records, $2.5\times$ for point queries and $3.9\times$ for range queries (range predicates are more expensive to evaluate in MP-SPDZ as they require two comparisons rather than one). On a slightly slower 1Gbps connection between the servers, Waldo performs $7.8\times$ better than MP-SPDZ for 2^{20} records with 8 range predicates. On the other hand, the ORAM baseline latency starts high but grows slowly, as expected due to its polylogarithmic complexity: for 2^{10} records, Waldo is approximately $37\times$ faster, and for 2^{20} records, approximately $1.4\times$ faster. Recall that our ORAM baseline does not pad to the maximum number of accesses and so has an advantage over Waldo.

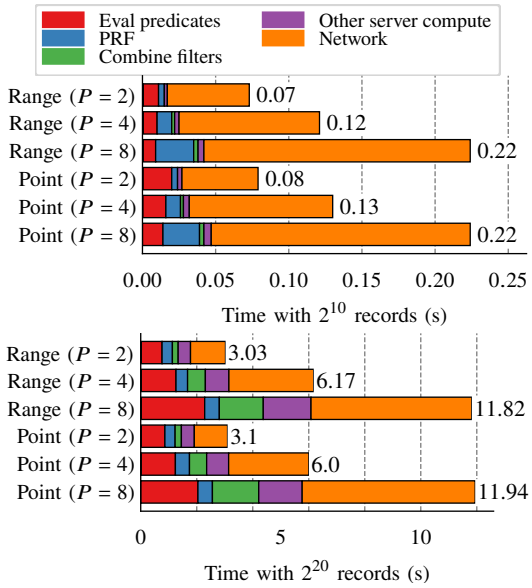
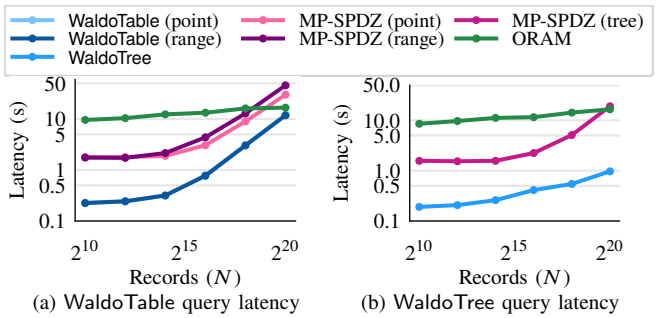


Fig. 7: Breakdown of WaldoTable query latency with different numbers of predicates (P).



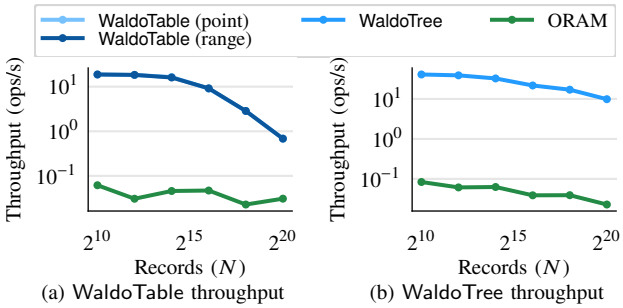
(a) WaldoTable query latency (b) WaldoTree query latency

Fig. 8: WaldoTable query latency is for 8 predicates, and latency for point and range predicates is almost identical.

Parallelism across servers. WaldoTable is parallelizable not just across cores, but also across servers without introducing new trust domains. We can split each "logical" server into n "physical" servers. The client divides its index into n equally-sized sub-indexes and delegates a sub-index to a triple of servers split across trust domains. The client can run its query on all n sub-indexes and locally aggregate the results. Because each triple processes its query chunk independently, parallelism is trivial. By using 12 servers instead of 3, we estimate that 8-predicate range queries take 3.0s, whereas with 3 servers they take 11.82s.

D. *Latency:* WaldoTree

In Fig. 8b, we show that WaldoTree queries are much faster than queries in our two baselines. WaldoTree achieves an $8-20\times$ improvement in query latency over MP-SPDZ, with the gap increasing for larger N. This gap is due to the fact that WaldoTree does not require server interaction, whereas MP-SPDZ requires a substantial amount of communication for comparisons. WaldoTree achieves a $45\times$ improvement over ORAM for 2^{10} records and a $17\times$ improvement for 2^{20} records. Again, this improvement is due to the fact that WaldoTree



(a) WaldoTable throughput (b) WaldoTree throughput

Fig. 9: WaldoTable is configured with 8 predicates and has similar throughput for point and range predicates. ORAM throughput fluctuates due to the fact that it is not fully oblivious (doesn't make the max number of accesses) and we randomly sample the data and queries.

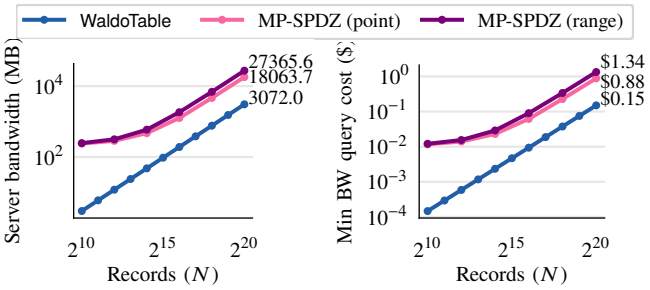
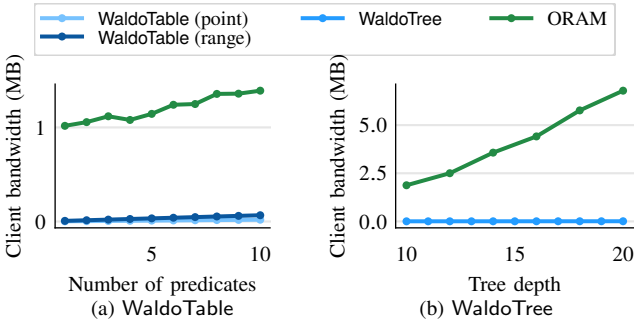


Fig. 10: Overhead and cost of total bandwidth between servers for a WaldoTable query with 8 predicates. We use the minimum AWS egress bandwidth cost of $0.05/GB [11] to compute the cost (bandwidth pricing is based on total egress bandwidth per month).



(a) WaldoTable (b) WaldoTree

Fig. 11: WaldoTable is configured with 2^{10} records and has almost identical bandwidth for point and range predicates.

does not require any network overhead to execute the query whereas the client must perform many ORAM accesses to traverse the tree, resulting in many round trips.

E. Throughput

In Fig. 9a and Fig. 9b, we compare Waldo's throughput to that of our ORAM baseline for a 90% append, 10% query workload (time-series workloads are append-heavy, see §II-A). Waldo's throughput is orders of magnitude higher: with 2^{10} records, WaldoTable's throughput is roughly 303× that of ORAM, and with 2^{20} records, roughly 22× that of ORAM. For WaldoTree, this gap is even more pronounced: with 2^{10} records, the throughput difference is approximately 488×, and with 2^{20} records, approximately 431×. The throughput gap is much larger than the latency gap due to the differing cost of updates. ORAM baseline updates require multiple ORAM accesses for inserting into and potentially rebalancing the R-tree. In contrast, Waldo clients only send a secret-shared one-hot vector to the three servers.

We don't report MP-SPDZ throughput numbers because our baseline only uses the MP-SPDZ framework for query execution and we do not implement updates. However, it is easy to compute a reasonable upper bound on MP-SPDZ's throughput. Appends in a system with MP-SPDZ would only require sending a small amount of data to each server, and so we can use 10ms as a lower bound for append latency (20ms round-trip time). From this, we can upper-bound MP-SPDZ's throughput for 90% appends and 10% searches: for WaldoTable functionality, with 2^{10} records, MP-SPDZ can achieve at most 5 ops/sec for point and range predicates (Waldo's throughput is 3.7× larger) and with 2^{20} records, 0.33 ops/sec for point predicates (Waldo's is 2× larger) and 0.22 ops/sec for range predicates (3× larger). For WaldoTree functionality, with 2^{10} records, MP-SPDZ can reach at most 5.7 ops/sec (Waldo's throughput is 7× larger), and with 2^{20} records, at most 0.5 ops/sec (Waldo's is 19× larger).

F. Communication

Server communication. In Fig. 10, we compare the bandwidth between servers for Waldo and MP-SPDZ (ORAM is single-server). MP-SPDZ uses $80-82\times$ more server bandwidth for 2^{10} records, and $5.8-8.9\times$ more for 2^{20} records. This is due to the fact that MP-SPDZ uses communication to perform comparisons to evaluate predicates, whereas Waldo only uses compute for these evaluations. The MP-SPDZ bandwidth for 2^{10} records is inflated due to how MP-SPDZ batches comparisons; bandwidth grows linearly starting at 2^{14} records.

Client communication. In Fig. 11a and Fig. 11b, we show how the client bandwidth of Waldo compares to that of our ORAM baseline. Again, we do not include MP-SPDZ here because we did not implement a client, although the client bandwidth cost would likely be small. While Waldo is in the range of tens of kilobytes, our ORAM baseline is clearly in the range of megabytes (even without the ORAM baseline padding the number of accesses to match Waldo's security). In Waldo, the client only has to send 4 FSS keys to each server for every predicate for a WaldoTable query, and 8 FSS keys total for a WaldoTree query. In contrast, the ORAM baseline requires many roundtrips, performing multiple ORAM accesses for queries.

G. System cost

Server bandwidth has serious implications for system cost. Clouds typically charge steep prices for transferring data out of the cloud to incentivize customers to keep their data in the cloud. This cost is challenging for a distributed trust setting where servers routinely need to send information between clouds. For example, AWS charges $0.09/GB if communication is less than 10TB each month and $0.05/GB if communication is greater than 150TB per month [11]. Executing a query with 8 range predicates for 2^{20} records with MP-SPDZ costs $1.34-$2.41 (depending on communication cost). In contrast, a Waldo query with 8 predicates only costs $0.15-$0.27 (depending on communication cost) because Waldo uses much less communication. Each server (r5n.16xlarge instance) only costs $4.77 per hour.

VIII. Limitations and future work

Small feature sizes are critical for good performance in WaldoTable. In §IV-A, we discuss techniques for encoding values in a large domain using a small feature size. While Waldo supports richer functionality than prior work [21, 22], it does not support all features provided by some modern plaintext time-series databases (e.g. retrieving individual records, sorts, group by, or joins [9, 101]). Also, Zeph [22] supports differential privacy [34] but Waldo does not because it provides malicious security: the servers would need to add noise in a verifiably correct way and without causing MAC verification to fail. Supporting more expressive queries and providing differential privacy are valuable directions for future work. Finally, our ORAM baseline has weaker security guarantees than Waldo (recall that we do not pad to the maximum number of accesses), and so our baseline has a performance advantage. We expect that at some point for a very large number of records, the ORAM baseline will outperform Waldo because of its asymptotic complexity, but because the ORAM baseline has a performance advantage, it is not clear exactly where an ORAM solution of comparable security overtakes Waldo.

IX. Related work

Private time series queries. Timecrypt [21] and Zeph [22] both support queries over encrypted time-series data. Unlike Waldo, they do not support filtering and leak the queried time interval. Also unlike Waldo, both systems also focus on allowing a third-party service fine-grained access to data, with Zeph considering aggregation across users. Other works explore similarity range queries over encrypted time-series data [113] and queries over encrypted and compressed time-series data [51], but these use specialized encryption schemes that have leakage that can be leveraged in statistical attacks.

FSS for private queries and secure computation. Splinter [106] uses FSS to allow users to make a variety of private queries, but unlike Waldo, the data is public and the servers are assumed to be semihonest. Dory [30] uses DPFs to enable clients to privately search for keywords in encrypted files without leaking search access patterns. Dory's queries are much simpler than Waldo's, and while Dory defends against malicious adversaries, its techniques don't easily extend to our setting because there isn't a clear way to combine its MAC tags for different predicates in the same query. Durasift [39] uses n servers to support at most $n-3$ conjunctions of arbitrary boolean expressions of keywords using private information retrieval, implemented with a DPF. Unlike Waldo, Durasift operates in the semihonest setting, can only combine a limited number of predicates, does not support range predicates, and operates on lists of documents rather than aggregates. Floram [33] and Bunn et al. [20] use DPFs for private reading and writing in the distributed ORAM setting; these works only need to provide a block storage abstraction, whereas Waldo must evaluate predicates and combine filters. Boyle et al. first explored how FSS can be used to implement secure computation [19], with subsequent work improving on these constructions providing malicious security [16].

Encrypted databases. Encrypted databases [32, 41, 80–82, 85, 103] execute expressive queries on encrypted data, but often achieve good performance by permitting some leakage. This leakage can be exploited in statistical attacks to learn the query and database contents [23, 46–48, 56, 58, 61, 62, 64, 79, 84, 111]. SisoSPIR [55] improves performance and reduce leakages by splitting trust, but only shows how to traverse a B-tree obliviously, which is not enough to compute multi-predicate aggregates. Some encrypted databases achieve good performance by using secure hardware [37, 40, 88, 104, 112]. These solutions require additional trust assumptions due to known side-channel attacks. Another set of encrypted databases are tailored to the IoT setting, but these systems do not provide the same security and functionality as Waldo: they use encryption schemes that leak information about the database contents, reveal the query to the server, or do not support filtering [50, 92, 93, 109].

Collaborative analytics. Collaborative analytics, a related line of work, allows mutually distrusting parties to run analytics queries over their combined data. Although the setting is different than ours, like Waldo, these works split trust across parties and leverage MPC techniques to run database queries. Senate [83], Secrecy [66], SMCQL [12], and Conclave [105] support more complex analytics queries than Waldo, although they use more heavyweight tools to do so.

Secure Aggregation. Prior work also explores aggregating data across many users without a single trusted server. Some of these systems split trust across multiple servers using secret sharing [6, 26, 29, 36, 63, 94]. Like Waldo, they support aggregation on private data, but unlike Waldo, they do not support private queries.

X. Conclusion

We presented Waldo, a private time-series database that operates in the malicious three-party honest-majority setting. While prior work [21, 22] only supports time-based filtering and reveals the queried time intervals, Waldo enables multi-predicate filtering while hiding the filter values and search access patterns. Waldo contributes new techniques that build on top of function secret sharing to enable Waldo to evaluate predicates non-interactively. Our MPC baseline uses $9-82\times$ more bandwidth between servers than Waldo (for different numbers of records), and our ORAM baseline uses $20-152\times$ more bandwidth between the client and server(s) than Waldo (for different numbers of predicates).

Acknowledgments. We thank the anonymous reviewers and shepherd for their helpful feedback. We also thank Ashley Zhang for participating in early stages of this work, as well as Henry Corrigan-Gibbs and students in the RISELab security group for giving feedback that improved the presentation of the paper. This work is supported by NSF CISE Expeditions Award CCF-1730628, NSF CAREER 1943347, and gifts from the Sloan Foundation, Alibaba, Amazon Web Services, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware. This work is also supported by a NSF Graduate Research Fellowship and a Microsoft Ada Lovelace Research Fellowship.

References

[1] Amazon EC2 instance network bandwidth. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-network-bandwidth.html. Accessed: 2021-08-08.

[2] cryptoTools. https://github.com/ladnir/cryptoTools/tree/master.

[3] libPSI. https://github.com/osu-crypto/libPSI.

[4] Multi-Protocol SPDZ: Versatile framework for multi-party computation. https://github.com/data61/MP-SPDZ. Accessed: 2021-07-30.

[5] SEAL-ORAM. https://github.com/InitialDLab/SEAL-ORAM.

[6] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. Prio+: Privacy preserving aggregate statistics via boolean shares. *IACR Cryptol. ePrint Arch.*, 2021.

[7] Amazon Timestream. https://aws.amazon.com/timestream/.

[8] Michael P Andersen and David E Culler. Btrdb: Optimizing storage system design for timeseries processing. In *USENIX FAST*, pages 39–52, 2016.

[9] Apache Druid. https://druid.apache.org/.

[10] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *ACM CCS*, pages 805–817, 2016.

[11] Amazon AWS. Amazon EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/.

[12] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. SMCQL: secure querying for federated databases. *VLDB*, 2017.

[13] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, 1991.

[14] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT*, pages 169–188, 2011.

[15] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[16] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. Function secret sharing for mixed-mode and fixed-point secure computation. In *EUROCRYPT (2)*, pages 871–900, 2021.

[17] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *EUROCRYPT*, pages 337–367, 2015.

[18] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *ACM CCS*, pages 1292–1303, 2016.

[19] Elette Boyle, Niv Gilboa, and Yuval Ishai. Secure computation with preprocessing via function secret sharing. In *TCC*, pages 341–371, 2019.

[20] Paul Bunn, Jonathan Katz, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient 3-party distributed oram. In *International Conference on Security and Cryptography for Networks*, pages 215–232. Springer, 2020.

[21] Lukas Burkhalter, Anwar Hithnawi, Alexander Viand, Hossein Shafagh, and Sylvia Ratnasamy. Timecrypt: Encrypted data stream processing at scale with cryptographic access control. In *USENIX NSDI*, pages 835–850, 2020.

[22] Lukas Burkhalter, Nicolas Küchler, Alexander Viand, Hossein Shafagh, and Anwar Hithnawi. Zeph: Cryptographic enforcement of end-to-end data privacy. In *USENIX OSDI*, pages 387–404, 2021.

[23] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, pages 668–679. ACM, 2015.

[24] Clevabit. https://www.clevabit.com/.

[25] Google Cloud. Network bandwidth. https://cloud.google.com/compute/docs/network-bandwidth.

[26] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *USENIX NSDI*, pages 259–282, 2017.

[27] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. $\text{SPD}\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In *CRYPTO (2)*, pages 769–798, 2018.

[28] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662, 2012.

[29] George Danezis, Cédric Fournet, Markulf Kohlweiss, and Santiago Zanella-Béguelin. Smart meter aggregation via secret-sharing. In *Proceedings of the first ACM workshop on Smart energy grid security*, pages 75–80, 2013.

[30] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. DORY: An Encrypted Search System with Distributed Trust. In *USENIX OSDI*, pages 1101–1119, 2020.

[31] Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Access control: principles and solutions. *Software: Practice and Experience*, 33(5):397–421, 2003.

[32] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. SEAL: Attack mitigation for encrypted databases via adjustable leakage. In *USENIX Security*, 2020.

[33] Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In *CCS*, pages 523–535. ACM, 2017.

[34] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3-4):211–407, 2014.

[35] Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. Use your brain! arithmetic 3pc for any modulus with active security. In *ITC*, volume 163 of *LIPIcs*, pages 5:1–5:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[36] Tariq Elahi, George Danezis, and Ian Goldberg. Privex: Private collection of traffic statistics for anonymous communication networks. In *ACM CCS*, pages 1068–1079, 2014.

[37] Saba Eskandarian and Matei Zaharia. ObliDB: oblivious query processing for secure databases. *VLDB*, 13(2):169–183, 2019.

[38] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. Rich queries on encrypted data: Beyond exact matches. In *European symposium on research in computer security*, pages 123–145. Springer, 2015.

[39] Brett Hemenway Falk, Steve Lu, and Rafail Ostrovsky. Durasift: A robust, decentralized, encrypted database supporting private searches with complex policy controls. In *WPES*, pages 26–36, 2019.

[40] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. HardIDX: Practical and secure index with SGX. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 386–408. Springer, 2017.

[41] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K Cunningham. Sok: Cryptographically protected database search. In *IEEE S&P*, pages 172–191. IEEE, 2017.

[42] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *EUROCRYPT (2)*, pages 225–255, 2017.

[43] Gabriel Ghinita, Panos Kalnis, Ali Khoshgozaran, Cyrus Shahabi, and Kian-Lee Tan. Private queries in location based services: anonymizers are not necessary. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 121–132, 2008.

[44] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *ACM STOC*, pages 218–229, 1987.

[45] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 1996.

[46] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *ACM CCS*, pages 315–331, 2018.

[47] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *IEEE S&P*, pages 1067–1083. IEEE, 2019.

[48] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. Encrypted databases: New volume attacks against range queries. In *ACM CCS*, pages 361–378, 2019.

[49] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, 1984.

[50] Subir Halder and Mauro Conti. Crypsh: A novel iot data protection scheme based on bgn cryptosystem. *IEEE Transactions on Cloud Computing*, 2021.

[51] Matúš Harvan, Samuel Kimoto, Thomas Locher, Yvonne-Anne Pignolet, and Johannes Schneider. Processing encrypted and compressed time series data. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1053–1062. IEEE, 2017.

[52] InfluxDB. https://www.influxdata.com/.

[53] InfluxData. Time series database explained. https://www.influxdata.com/time-series-database/.

[54] InfluxDB. https://www.influxdata.com/.

[55] Yuval Ishai, Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. Private large-scale databases with distributed searchable symmetric encryption. In *Cryptographers' Track at the RSA Conference*, pages 90–107. Springer, 2016.

[56] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, volume 20, page 12. Citeseer, 2012.

[57] Alan F Karr, Xiaodong Lin, Ashish P Sanil, and Jerome P Reiter. Secure regression on distributed databases. *Journal of Computational and Graphical Statistics*, 14(2):263–279, 2005.

[58] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. Generic attacks on secure outsourced databases. In *ACM CCS*, pages 1329–1340, 2016.

[59] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *ACM CCS*, pages 1575–1590. ACM, 2020.

[60] Ali Khoshgozaran and Cyrus Shahabi. Private information retrieval techniques for enabling location privacy in location-based services. In *Privacy in Location-Based Applications*, pages 59–83. Springer, 2009.

[61] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. The state of the uniform: attacks on encrypted databases beyond the uniform query distribution. In *IEEE S&P*, pages 1223–1240. IEEE, 2020.

[62] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Response-hiding encrypted ranges: Revisiting security via parametrized leakage-abuse attacks. *IACR Cryptol. ePrint Arch.*, 2021:93, 2021.

[63] Klaus Kursawe, George Danezis, and Markulf Kohlweiss. Privacy-friendly aggregation for the smart-grid. In *PETS*, pages 175–191. Springer, 2011.

[64] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *IEEE S&P*, pages 297–314. IEEE, 2018.

[65] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Authenticated index structures for aggregation queries. *ACM Transactions on Information and System Security (TISSEC)*, 13(4):1–35, 2010.

[66] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. Secrecy: Secure collaborative analytics on secret-shared data. *arXiv preprint arXiv:2102.01048*, 2021.

[67] Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*, pages 277–346. Springer International Publishing, 2017.

[68] Tammy Lovell. Swedish healthcare advice line stored 2.7 million patient phone calls on unprotected web server, February 20 2019. https://www.healthcareitnews.com/news/swedish-healthcare-advice-line-stored-27-million-patient-phone-calls-unprotected-web-server.

[69] Kathryn I Marko, Jill M Krapf, Andrew C Meltzer, Julia Oh, Nihar Ganju, Anjali G Martinez, Sheetal G Sheth, and Nancy D Gaba. Testing the feasibility of remote patient monitoring in prenatal care using a mobile app and connected devices: a prospective observational trial. *JMIR research protocols*, 5(4):e200, 2016.

[70] Luca Melis, George Danezis, and Emiliano De Cristofaro. Efficient private statistics with succinct sketches. *arXiv preprint arXiv:1508.06110*, 2015.

[71] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *ACM CCS*, pages 35–52, 2018.

[72] MongoDB. Time Series Data and MongoDB. https://www.mongodb.com/blog/post/time-series-data-and-mongodb-part-1-introduction.

[73] Ellen Nakashima. Russian government hackers penetrated DNC, stole opposition research on Trump, June 14 2016. https://www.washingtonpost.com/world/national-security/russian-government-hackers-penetrated-dnc-stole-opposition-research-on-trump/2016/06/14/cf006cb4-316e-11e6-8ff7-7b6c1998b7a0_story.html.

[74] Department of health and human services. Medicare and Medicaid Programs; Policy and Regulatory Revisions in Response to the COVID-19 Public Health Emergency. https://www.cms.gov/files/document/covid-final-ifc.pdf.

[75] OpenHAB. https://www.openhab.org/.

[76] OpenTSDB. http://opentsdb.net/.

[77] Charlie Osborne. Fortune 500 company leaked 264gb of client, payment data, June 7 2019. https://www.zdnet.com/article/veteran-fortune-500-company-leaked-264gb-in-client-payment-data/.

[78] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *STOC*. ACM, 1990.

[79] Simon Oya and Florian Kerschbaum. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In *USENIX Security*, 2021.

[80] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big data analytics over encrypted datasets with seabed. In *USENIX OSDI 16*, pages 587–602, 2016.

[81] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. Blind seer: A scalable private dbms. In *IEEE S&P*, pages 359–374. IEEE, 2014.

[82] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: an encrypted database using semantically secure encryption. *VLDB*, 12(11):1664–1678, 2019.

[83] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M Hellerstein. Senate: A maliciously-secure MPC platform for collaborative analytics. In *USENIX Security*, 2021.

[84] Rishabh Poddar, Stephanie Wang, Jianan Lu, and Raluca Ada Popa. Practical volume-based attacks on encrypted databases. 2020.

[85] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.

[86] Prevounce. Examples of remote patient monitoring: 9 top patient applications. https://blog.prevounce.com/examples-of-remote-patient-monitoring-9-top-patient-applications.

[87] Prevounce. Remote Monitoring of Peak Expiratory Flow. https://blog.prevounce.com/remote-monitoring-of-peak-expiratory-flow.

[88] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *IEEE S&P*, pages 264–278. IEEE, 2018.

[89] Prometheus. https://prometheus.io/.

[90] Corinne Reichert. Payroll data for 29,000 facebook employees stolen, December 13 2019. https://www.cnet.com/news/payroll-data-of-29000-facebook-employees-reportedly-stolen/.

[91] Luca Santini, Karim Mahfouz, Valentina Schirripa, Nicola Danisi, Michelangelo Leone, Gloria Mangone, Monica Campari, Sergio Valsecchi, and Fabrizio Ammirati. Preliminary experience with a novel multisensor algorithm for heart failure monitoring: The heartlogic index. *Clinical case reports*, 6(7):1317, 2018.

[92] Hossein Shafagh, Anwar Hithnawi, Lukas Burkhalter, Pascal Fischli, and Simon Duquennoy. Secure sharing of partially homomorphic encrypted iot data. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, pages 1–14, 2017.

[93] Hossein Shafagh, Anwar Hithnawi, Andreas Dröscher, Simon Duquennoy, and Wen Hu. Talos: Encrypted query processing for the internet of things. In *Proceedings of the 13th ACM conference on embedded networked sensor systems*, pages 197–210, 2015.

[94] Elaine Shi, TH Hubert Chan, Eleanor Rieffel, Richard Chow, and Dawn Song. Privacy-preserving aggregation of time-series data. In *Proc. NDSS*, volume 2, pages 1–17. Citeseer, 2011.

[95] Smartcar. https://smartcar.com/.

[96] SolarNetwork. https://solarnetwork.github.io/.

[97] Myung-kyung Suh, Chien-An Chen, Jonathan Woodbridge, Michael Kai Tu, Jung In Kim, Ani Nahapetian, Lorraine S Evangelista, and Majid Sarrafzadeh. A remote patient monitoring system for congestive heart failure. *Journal of medical systems*, 35(5):1165–1179, 2011.

[98] Myung-kyung Suh, Chien-An Chen, Jonathan Woodbridge, Michael Kai Tu, Jung In Kim, Ani Nahapetian, Lorraine S Evangelista, and Majid Sarrafzadeh. A remote patient monitoring system for congestive heart failure. *Journal of medical systems*, 35(5):1165–1179, 2011.

[99] Sijun Tan, Brian Knott, Yuan Tian, and David J Wu. Cryptgpu: Fast privacy-preserving machine learning on the gpu. 2021.

[100] Timescale. How Everactive powers a dense sensor network with virtually no power at all. https://blog.timescale.com/blog/how-everactive-powers-a-dense-sensor-network-with-virtually-no-power-at-all/.

[101] Timescale. https://www.timescale.com/.

[102] Timescale. What is time-series data? https://docs.timescale.com/timescaledb/latest/overview/what-is-time-series-data/#what-is-time-series-data.

[103] Stephen Lyle Tu, M Frans Kaashoek, Samuel R Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. 2013.

[104] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. Stealthdb: a scalable encrypted database with full sql query support. *PETS*, 2019(3):370–388, 2019.

[105] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–18, 2019.

[106] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical private queries on public data. In *USENIX NSDI*, pages 299–313, 2017.

[107] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *ACM CCS*, pages 215–226, 2014.

[108] Avi Wigderson, MB Or, and S Goldwasser. Ben-or, michael and goldwasser, shafi and wigderson, avi. In *STOC*, 1988.

[109] Wanli Xue, Chenwen Luo, Guohao Lan, Rajib Rana, Wen Hu, and Aruna Seneviratne. Kryptein: a compressive-sensing-based encryption scheme for the internet of things. In *2017 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 169–180. IEEE, 2017.

[110] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

Client.WaldoTree.Query$^{S_1,S_2,S_3}(0,t)$

$\alpha \xleftarrow{R} \mathbb{Z}_{2^{k+s}}$
for $i = 1$ to 3 **do**
  $K_1^i, K_2^i \leftarrow \mathsf{Gen}^<(1^\lambda, t, 1)$
  $K_1'^i, K_2'^i \leftarrow \mathsf{Gen}^<(1^\lambda, t, \alpha)$
end for
$\mathcal{K}_1 \leftarrow K_1^1, K_2^2$ and $\mathcal{K}_1' \leftarrow K_1'^1, K_1'^2$
$\mathcal{K}_2 \leftarrow K_2^2, K_1^3$ and $\mathcal{K}_2' \leftarrow K_2'^2, K_1'^3$
$\mathcal{K}_3 \leftarrow K_2^3, K_2^1$ and $\mathcal{K}_3' \leftarrow K_2'^3, K_2'^1$
for $i = 1$ to 3 **do**
  $\{x_i^{(j)}\}_{j=0}^{\log n}, \{\sigma_i^{(j)}\}_{j=0}^{\log n} \leftarrow S_i.\mathsf{WaldoTree.Query}(\mathcal{K}_i, \mathcal{K}_i')$
end for
for $j = 0$ to $\log n$ **do**
  $x^{(j)} \leftarrow \sum_{i=1}^3 x_i^{(j)}, \sigma^{(j)} \leftarrow \sum_{i=1}^3 \sigma_i^{(j)}$
  if $\alpha \cdot x^{(j)} \neq \sigma^{(j)}$ **then**
    Output $\perp$ and broadcast $\perp$ to all servers
  end if
end for
Output $x \leftarrow \mathsf{Agg}(\{x^{(0)}, \ldots, x^{(\log n)}\})$

Fig. 12: Client WaldoTree.Query algorithm. n is the number of leaves in the current shared aggregate tree. Here the aggregate is computed over time range 0 to t. General case follows similarly by using double the keys and servers returning $2\log n + 1$ values. Aggregation function Agg is defined by clients during call to Init procedure; it takes in a list of values and outputs their aggregate. $\{x^i\}_{i=a}^b$ denotes $\{x^a, \ldots, x^b\}$.

Server.WaldoTree.Query$^{S_1,S_2,S_3}(\mathcal{K}, \mathcal{K}')$

Parse $\mathcal{K}$ as K_1, K_2 and $\mathcal{K}'$ as K_1', K_2'
Initialize empty trees P, Q with n leaves each.
$\{x_0, \ldots, x_{\log n}\} \leftarrow \{0, \ldots, 0\}$, $\{\sigma_0, \ldots, \sigma_{\log n}\} \leftarrow \{0 \ldots, 0\}$
for $i \in n$ **do**
  $P_{\log n}^{(i)} \leftarrow \big(\mathsf{Eval}(K_1, i), \mathsf{Eval}(K_2, i)\big)$
  $Q_{\log n}^{(i)} \leftarrow \big(\mathsf{Eval}(K_1', i), \mathsf{Eval}(K_2', i)\big)$
end for
for $d \in \{\log n - 1, \ldots, 0\}$ **do**
  for $i \in \{1, \ldots, 2^d\}$ **do**
    $P_d^{(i)} \leftarrow P_d^{(i)}.\mathsf{right}$ and $Q_d^{(i)} \leftarrow Q_d^{(i)}.\mathsf{right}$
  end for
end for
for $d \in \{0, \ldots, \log n\}$ **do**
  for $i \in \{1, \ldots, 2^d\}$ **do**
    $x_d \leftarrow x_d + P_d^{(i)} \odot \big(T.\mathsf{first}_d^{(i)}, T.\mathsf{second}_d^{(i)}\big)$
    $\sigma_d \leftarrow \sigma_d + Q_d^{(i)} \odot \big(T.\mathsf{first}_d^{(i)}, T.\mathsf{second}_d^{(i)}\big)$
  end for
end for
for $d \in \{0, \ldots, \log n - 1\}$ **do**
  for $i \in \{1, \ldots, 2^d\}$ **do**
    $c \leftarrow T.\mathsf{first}_d^{(i)}.\mathsf{left} + T.\mathsf{first}_d^{(i)}.\mathsf{right}$
    $\hat{c} \leftarrow T.\mathsf{second}_d^{(i)}.\mathsf{left} + T.\mathsf{second}_d^{(i)}.\mathsf{right}$
    $x_{d+1} \leftarrow x_{d+1} - P_d^{(i)} \odot (c, \hat{c})$
    $\sigma_{d+1} \leftarrow \sigma_{d+1} - Q_d^{(i)} \odot (c, \hat{c})$
  end for
end for
Output $(\{x_0, \ldots, x_{\log n}\}, \{\sigma_0, \ldots, \sigma_{\log n}\})$

Fig. 13: Server WaldoTree.Query algorithm. n is the number of leaves, $\mathsf{ID} \in \{1, 2, 3\}$ is the server id. T denotes the shared aggregate tree corresponding to WaldoTree object. $T_d^{(i)}$ denotes the ith node on dth level of the tree and $i.\mathsf{left}, i.\mathsf{right}$ access the value stored on the left and right child of a node i, respectively. $(a, b) \odot (c, d) = ac + bd$. Here the aggregate is computed over time range 0 to t, and so on line 10, the right child's activation status is used at parent. General case follows similarly by using twice as many keys and returning $2\log n + 1$ values.

[111] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security*, pages 707–720, 2016.

[112] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *USENIX NSDI*, pages 283–298, 2017.

[113] Yandong Zheng, Rongxing Lu, Yunguo Guan, Jun Shao, and Hui Zhu. Efficient and privacy-preserving similarity range query over encrypted time series data. *IEEE Transactions on Dependable and Secure Computing*, 2021.

APPENDIX

A. Security analysis

We use the simulation paradigm [67] of multiparty computation (MPC) to prove Waldo's security guarantees. We consider a probabilistic polynomial time (PPT) adversary $\mathcal{A}$ who *statically* corrupts at most one of the three servers. Under $\mathcal{A}$'s control, the corrupted server is allowed to be *malicious*, meaning that it can deviate arbitrarily from the protocol specification. To keep the proof description simple, we first assume that client behaves honestly. At the end of this section, we extend to the case of malicious clients as well as collusion between client and a malicious server.

Proving security under the simulation paradigm requires defining two worlds: the real world where the actual protocol is run by honest parties, and an ideal world where an ideal functionality $\mathcal{F}$ takes inputs from the parties and directly outputs the result to the concerned party. $\mathcal{A}$ controls the behavior of the corrupted server as well as observes its view during the protocol run. To make the ideal world view of $\mathcal{A}$ indistinguishable from the real world, we need to define a simulator $\mathcal{S}$ whose job is to "simulate" messages to $\mathcal{A}$ that are similar to what the client and honest servers send to the corrupted server in the real world. However, $\mathcal{S}$ can only interact with the corrupted server and $\mathcal{F}$. If the ideal functionality $\mathcal{F}$ correctly models the leakage that our protocols claim to have and if $\mathcal{A}$ cannot distinguish between the two worlds, then we deem our protocols as being secure. To account for the case where an adversary is able to influence the operations issued by clients, and see their final result, we allow $\mathcal{A}$ to freely choose the queries issued by clients and see the result. For simplicity, we assume that predicates are only connected via conjunctions (see §IV-B for a discussion of negations, which are necessary to support disjunctions).

Ideal Functionality $\mathcal{F}$. We first define our stateful ideal functionality $\mathcal{F}$ which stores the current time series database and responds to requests in the following way:

1) $\mathsf{Init}(1^\lambda, 1^{\tilde{s}}, \mathsf{schema})$: $\mathcal{F}$ runs initialization given security parameters $\lambda, \tilde{s}$ and a schema layout parameter schema, where for WaldoTable, $\mathsf{schema} = (N, F, 2^{\ell_1}, \ldots, 2^{\ell_F})$ with N number of records in the window, F is the number of features and ℓ_i the feature size for ith feature, and for WaldoTree, $\mathsf{schema} = (2^\ell, \mathsf{type})$ with ℓ being the feature size and type is a user-defined aggregation function.
2) $\mathsf{WaldoTable.Append}(t, v_1, \ldots, v_F)$: $\mathcal{F}$ updates the current index to store record with timestamp t and value $v_1 \in \mathbb{Z}_{2^{\ell_1}}, \ldots, v_F \in \mathbb{Z}_{2^{\ell_F}}$.

3) WaldoTree.Append(t, v): $\mathcal{F}$ updates the tree index with the new entry for timestamp t and value $v \in \mathbb{Z}_{2^\ell}$.
4) WaldoTable.Query$(P_1 \wedge \cdots \wedge P_n, \text{feature}, \text{type}) \rightarrow x$: $\mathcal{F}$ aggregates by type for feature over the boolean formula composed of predicates $P_1, \ldots, P_n$ and outputs the result x *only* to the client.
5) WaldoTree.Query$(t_1, t_2) \rightarrow x$: $\mathcal{F}$ aggregates by type over time interval (t_1, t_2). It finally outputs the result x *only* to the client.

We allow $\mathcal{F}$ to leak $\mathsf{Leak}(\mathcal{F}) = (\text{schema}, \text{struct}_Q)$, where query structure struct_Q is defined as: (1) $(\text{Init}, \lambda, \tilde{s})$ for Init, (2) (Append, t) for Append, (3) $(\text{Query}, n, \text{feature}, \text{type}, \text{kind}, \text{fid})$ for WaldoTable.Query with type denoting the aggregation function (e.g. sum, count), kind denoting point or range predicates, and fid denoting a vector of feature ids corresponding to each predicate, and (4) (Query) for WaldoTree.Query.

Definition 1. Let Π be a protocol for an encrypted time-series database which takes as input requests from clients, say Q. $\mathcal{F}$ as defined above models the functionality provided by Π as a trusted party. Let $\mathcal{A}$ be an adversary who observes the view of a statically corrupted server during the protocol run and gets the final client output. Let $\mathsf{View}^{\text{Real}}_{\Pi(Q)}$ denote $\mathcal{A}$'s view in the real world experiment. In the ideal world, a simulator $\mathcal{S}$ generates a simulated view $\mathsf{View}^{\text{Ideal}}_{\mathcal{S},\mathsf{Leak}(\mathcal{F}(Q))}$ to $\mathcal{A}$ given only the leakage of $\mathcal{F}$. Then, $\forall$ non-uniform algorithms $\mathcal{A}$ PPT in $\lambda, \tilde{s}$, where $\lambda, \tilde{s}$ are computational and statistical security parameters, respectively, $\exists$ a PPT algorithm $\mathcal{S}$ s.t.

$$\Pr[Q \leftarrow \mathcal{A}(1^\lambda, 1^{\tilde{s}}); b \xleftarrow{R} \{0,1\}; \mathcal{A}(\mathsf{View}_b, Q) = b] \leq \frac{1}{2} + \text{negl}(\lambda) + \text{negl}(\tilde{s})$$

$$\text{where } \mathsf{View}_0 = \mathsf{View}^{\text{Real}}_{\Pi(Q)}, \mathsf{View}_1 = \mathsf{View}^{\text{Ideal}}_{\mathcal{S},\mathsf{Leak}(\mathcal{F}(Q))}$$

Theorem 1: *Using Definition 1 (§A), Waldo securely evaluates (with abort) the ideal functionality $\mathcal{F}$ (§A) when instantiated with secure distributed point and comparison functions and a pseudo-random function, all with a computational security parameter of λ.*

Proof. We begin by first providing a construction for our simulator $\mathcal{S}$ for the ideal world. We work in the hybrid model [67], where invocations of sub-protocols can be replaced with that of the corresponding functionalities, as long as the sub-protocol is proven to be secure. We will operate in the $\mathcal{F}_{\text{Correlated}}$-hybrid model, where we assume the existence of a secure protocol $\Pi_{\text{Correlated}}$ (described in prior works [10, 99]), which realizes the ideal functionality $\mathcal{F}_{\text{Correlated}}$ that generates 3-party RSS shares of the value 0 (used in Reshare) or a random value (used in RandCoeff).

Simulator Construction. Without loss of generality, let S_1 be the corrupted party. Depending on the current request Q and given access to $\mathsf{Leak}(\mathcal{F}(Q))$, $\mathcal{S}$ does the following:

1) *On receiving* $(\text{Init}, \text{schema}, \lambda, \tilde{s})$ *from* $\mathcal{F}$: $\mathcal{S}$ stores it locally and forwards it to $\mathcal{A}$ (who we assumed corrupts S_1).
2) *On receiving* (Append, t) *from* $\mathcal{F}$: $\mathcal{S}$ samples RSS shares of a *randomly* sampled record satisfying the structure dictated by the schema and appends them to the local database. $\mathcal{S}$ then forwards S_1's share to $\mathcal{A}$.
3) *On receiving* (Query) *from* $\mathcal{F}$: $\mathcal{S}$ samples a *random* query Q satisfying the structure dictated by schema and generates corresponding DCF keys. $\mathcal{S}$ then sends the keys for S_1 to $\mathcal{A}$. At the end, $\mathcal{S}$ receives the final output shares of S_1 from $\mathcal{A}$. If received shares aren't exactly as expected ($\mathcal{S}$ has the RSS shares and FSS keys of S_1 to check this), then output $\perp$ to $\mathcal{A}$.
4) *On receiving* $(\text{Query}, n, \text{feature}, \text{type}, \text{kind}, \text{fid})$ *from* $\mathcal{F}$: $\mathcal{S}$ samples a *random* query Q satisfying the structure $(\text{schema}, n, \text{feature}, \text{type}, \text{kind}, \text{fid})$. $\mathcal{S}$ generates FSS keys (DPF keys if kind = 0 and DCF otherwise) for Q and stores them locally. It then sends the keys for S_1 to $\mathcal{A}$. Given access to $\mathcal{F}_{\text{Correlated}}$, $\mathcal{S}$ follows the rest of the steps (for multiplication) emulating the actions performed by S_2, S_3 in the real protocol. Since $\mathcal{S}$ has access to the FSS keys as well as all the database shares, it can generate the expected versions of messages from $\mathcal{A}$ that it should see. If any of the messages deviate, $\mathcal{S}$ sets an abort flag and continues. At the end, $\mathcal{S}$ receives the final output share of S_1 from $\mathcal{A}$. If the abort flag is set, then output $\perp$ to $\mathcal{A}$.

We now prove that the view generated by $\mathcal{S}$ in the ideal world is indistinguishable from the real world for a computationally (in λ, s) bounded $\mathcal{A}$ through a sequence of hybrids $\mathcal{H}_0, \ldots, \mathcal{H}_5$.

Hybrid 0. We start with the ideal world as our initial hybrid.

Hybrid 1. Simulator $\mathcal{S}$ replaces FSS keys for the random query Q with the outputs of FSS simulators $\mathcal{S}_{\text{DPF}}$ and $\mathcal{S}_{\text{DCF}}$ [16, 19]. Since $\mathcal{S}$ has access to the database shares of all the three servers and the simulated FSS keys, it can check if the messages received from $\mathcal{A}$ are exactly as *expected* or not. From the security of FSS schemes for DPF and DCF, it follows that $\mathcal{A}$'s advantage in distinguishing $\mathcal{H}_0$ from $\mathcal{H}_1$ is $\text{negl}(\lambda)$.

Hybrid 2. We let our ideal functionality $\mathcal{F}$ forward the real queries WaldoTable.Query$(P_1 \wedge \cdots \wedge P_n, \text{feature}, \text{type})$, WaldoTree.Query$(t_1, t_2)$ to $\mathcal{S}$. Recall that we allow $\mathcal{A}$ to specify these queries. $\mathcal{S}$ generates FSS keys for the real query and replaces the simulated keys from $\mathcal{H}_1$ with the real ones. From the security of aforementioned FSS schemes, it holds that $\mathcal{A}$'s advantage in distinguishing $\mathcal{H}_1$ from $\mathcal{H}_2$ remains $\text{negl}(\lambda)$.

Hybrid 3. We now allow $\mathcal{F}$ to also forward the real append queries WaldoTable.Append$(t, v_1, \ldots, v_F)$ and WaldoTree.Append(t, v) to $\mathcal{S}$. $\mathcal{S}$ generates and distributes RSS shares of the real incoming record and uses that instead of RSS shares for randomly sampled record. Although $\mathcal{A}$ specifies the append query and knows the incoming record's value, it only sees RSS shares of at most one server. These limited shares are independent of the actual record values. Therefore, $\mathcal{A}$'s view in $\mathcal{H}_2$ and $\mathcal{H}_3$ is identical.

Hybrid 4. This hybrid corresponds to the real world with calls to $\mathcal{F}_{\text{Correlated}}$. The only difference in $\mathcal{H}_4$ over $\mathcal{H}_3$ is the reliance on MACs to detect malicious behavior. As mentioned earlier,

we allow the adversary to observe the client's final output for every query that it issues. $\mathcal{A}$'s view is distinguishable from $\mathcal{H}_3$ when the client's output is erroneous and it still doesn't abort. In Lemma 2, we prove that the probability of $\mathcal{A}$ cheating and not getting caught during MAC verification step is $\text{negl}(\tilde{s})$, for statistical security parameter $\tilde{s}$. Thus the distinguishing advantage of $\mathcal{A}$ between $\mathcal{H}_3$ and $\mathcal{H}_4$ is $\text{negl}(\tilde{s})$.

Hybrid 5. In our final hybrid, we replace the calls to $\mathcal{F}_{\text{Correlated}}$ with the PRF calls to realize $\Pi_{\text{Correlated}}$ for Reshare and RandCoeff. From the security of $\Pi_{\text{Correlated}}$ [10] (relies on PRF security), we have that $\mathcal{A}$ cannot tell $\mathcal{H}_4$ and $\mathcal{H}_5$ apart with probability any better than $\text{negl}(\lambda)$ over a random guess.

This concludes our proof that given views of either the real or ideal world, $\mathcal{A}$ cannot correctly guess which view it is, except with probability $\leq \frac{1}{2} + \text{negl}(\tilde{s}) + \text{negl}(\lambda)$. □

Lemma 2. *In our proposed protocols, a cheating server is caught by the client during MAC verification step with probability* $1 - \text{negl}(\tilde{s})$*, where* $\tilde{s} = s - \log(s+1)$ *is the statistical security parameter.*

Proof. Any locally introduced error cascades down in the subsequent computation across all servers due to exchange of malformed messages from the adversary. This can be modeled as an equivalent additive error of adversary's choosing in the protocol messages it sends. In particular, the only time servers communicate with each other is during Reshare, and any error introduced by $\mathcal{A}$ at this point, translates to the same error in the RSS shares that are next established. We formalize this using a non-uniform PPT algorithm $(e_{1,j}, \ldots, e_{N,j}) \leftarrow \mathsf{ChooseError}(j, t)$, where j denotes the communication round in the protocol and t is the current state available to $\mathcal{A}$. t includes all the information that $\mathcal{A}$ has about the distribution of secrets, initial protocol state as well as the transcript so far.

Our MAC check is: $\alpha \cdot \sum_{i,j} \chi_{i,j} x_{i,j} \stackrel{?}{=} \sum_{i,j} \chi_{i,j} \sigma_{i,j}$, where $\chi_{i,j}$ are random coefficients sampled from $\mathbb{Z}_{2^{k+s}}$ and $x_{i,j}$ (resp. $\sigma_{i,j}$) are all messages (resp. their MACs) whose secret shares are exchanged during jth communication round in the protocol. No server knows the values $\chi_{i,j}$. Given that $\exists i, j$ such that $e_{i,j} \neq 0$ (otherwise, there is no tampering), then passing the MAC check requires: $\alpha \cdot (\sum_{i,j} \chi_{i,j}(x_{i,j} + e_{i,j})) - \sum_{i,j} \chi_{i,j}\sigma_{i,j} + \Delta = 0$, where Δ is the corrective error $\mathcal{A}$ needs for the check to pass. It was shown in [27] that given α is sampled uniformly from $\mathbb{Z}_{2^{k+s}}$, the probability of passing the check $2^{-s+\log(s+1)}$. □

Client and Server Collusion. In the case of an adversarial client, the only goal of collusion can be to learn information about database entries that aren't accessible to the client based on the set access control policies. If the client tries to access databases outside its clearance, the two honest servers will immediately abort the protocol before even sending their first protocol message. Given that shares of the database with the remaining server (colluding with the client) are independent of the actual contents of the database, and neither the client nor the server receive any message from the two honest servers except an abort, it is straightforward to see that the case of a malicious client is securely dealt with. Note that the case of a malicious non-colluding client is subsumed in this preceding argument.