ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Querying Time-Series Data Privately

Bachelor Thesis

Kevin Solmssen

May 12, 2022

Advisors: Prof. Dr. Kenny Paterson, Lukas Burkhalter

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

With the increasing availability of networked devices such as the Internet of Things, services and applications increasingly collect large amounts of sensitive time-series data. Often this data is offloaded to a third-party storage for availability and analytics. While outsourcing data storage is convenient, this raises concerns about privacy and security for example, if data breaches occur. Previous systems such as TimeCrypt [7] and Zeph [8] aim to resolve this issue by encrypting records end-to-end. However, an adversary is still able to observe access patterns which can be detrimental in certain applications. For example, a malicious database provider, who observes the access patterns of a doctor retrieving encrypted heart rate measurements of a patient can deduce with high probability which time-windows contain irregular heart rates since a doctor tends to be interested in outlying data points. In this thesis, we present a time-series data storage system that hides a client's query access patterns from the database provider. Our system hides the accessed time window and stream from the server as long as two semi-honest storage providers do not collude. To achieve this, we employ a recently developed cryptographic tool called Function Secret Sharing (FSS), which allows splitting a query into compact shares to send to replicated servers without revealing the original query. With our implementation running on a 4-core system, we can support small- to medium-sized databases with acceptable latency requirements, such as querying a database with 1M records in under 1.5s.

Contents

Contents **iii**

1 Introduction **1**
1.1 Contributions 2
1.2 Outline 3

2 Background **5**
2.1 Time-Series data 5
2.2 Function Secret Sharing 6
2.2.1 Efficient Constructions of FSS Schemes 7

3 Related Work **9**
3.1 Private Time-Series Data Processing 9
3.1.1 TimeCrypt 9
3.1.2 Zeph 10
3.2 Private Queries with Function Secret Sharing 11
3.2.1 Splinter 11
3.2.2 Dory 11

4 Objectives **13**
4.1 Scenario 13
4.2 Goals 14
4.3 Threat Model 15
4.4 Access Pattern Leakage 15

5 Design **17**
5.1 Overview 17
5.1.1 Client, Server and Producer System Components . . . 18
5.2 Stream Abstractions 18
5.2.1 Client Data Serialization & Encoding 18
5.2.2 Server Storage Layout 19

5.3 Enabling Private Queries 20
5.3.1 Query Overview 20
5.3.2 Example: `SUM` Query 23
5.3.3 Further Supported Queries 25
5.4 Security Analysis 25
5.4.1 Correctness 25
5.4.2 Privacy 26
5.5 TimeCrypt Compatibility 27
5.6 System Optimizations 27
5.6.1 Parallelism 27
5.6.2 Batching 27

6 Implementation **29**
6.1 Batching & Parallelism 29

7 Evaluation **31**
7.1 Experimental Setup 31
7.2 Microbenchmarks 32
7.2.1 **Eval** Latency 32
7.2.2 **Gen** Latency 33
7.3 End-to-End Benchmarks 34
7.3.1 Single Query 34
7.3.2 Batch Sizes 36
7.3.3 Health App Dashboard 37

8 Conclusion **39**
8.1 Future Work 39

Bibliography **41**

Chapter 1

Introduction

Recent years have seen unprecedented growth in networked devices and services that continuously collect increasingly detailed time-series data about individuals. The collection of time-series data is increasingly prevalent across a wide range of systems and applications. The growth of time-series data is primarily attributed to the rising demand for instrumentation. Individuals and organizations are continuously logging various metrics that report systems' state for better diagnoses, forecasting, decision making, and resource allocation. However, with this trend comes the problem of ensuring the privacy of user data. Users today typically entrust their data to a third-party storage or application provider. However, there is growing concern that this model leaves users vulnerable to privacy violations due to misuse of their data - whether deliberate or inadvertent - by third-party providers. These concerns appear to be amply justified, given the numerous reports of recent data breaches and misuse.

A potential solution to this problem is to encrypt data end-to-end but enable the server to execute queries over encrypted data. Existing encrypted storages tailored for time-series data such as TimeCrypt [7] and Zeph [8] provide such efficient solutions to execute queries on encrypted time-series data. However, these systems have a serious limitation. Although, they protect the confidentiality of the queried data, they do not prevent the server from learning which parts of the data the client queries. In particular, they reveal the queried time interval to the server, which can be problematic in certain applications. For example, a patient stores their encrypted heart rate measurements on a server and gives their doctor access to them. A malicious database server can observe the specific time range the doctor queries and can conclude with a high probability, that the patient in question had some form of irregular heart rate at that specific time since a doctor tends to be interested in outlying data points.

In this thesis, we investigate how to eliminate this query leakage for time-series data stores. Potential general-purpose tools that could be used to eliminate query pattern leakage are oblivious RAM (ORAM) [10, 23] for encrypted data and Private Information Retrieval [9, 13] for public data. Though, applying these general-purpose tools directly to time-series workloads would incur high overheads that would render such systems impractical.

A recent cryptographic tool that can be used to hide query patterns with more efficiency is Function Secret Sharing (FSS) [4, 5]. In essence, FSS allows a client to generate compact shares of a function that the servers can use to evaluate the function (e.g., range predicate) without learning what the function is. For this hiding property, FSS assumes non-colluding servers that do not have to communicate with each other. Prior work explored how to use FSS to hide query patterns in different settings on encrypted and non-encrypted data [11, 27]. However, these techniques do not directly translate to the time-series setting. The goal of this thesis is to fill this gap and explore how FSS can be employed to hide query patterns in time-series workloads and evaluate its feasibility.

1.1 Contributions

In this thesis, we present a design and implementation of a time-series data store system, which employs FSS to hide query patterns from time-series storage providers. To hide query patterns with FSS, our system assumes that at least two non-colluding semi-honest storage providers from different trust domains exist. In the default setting, our system operates on plaintext data, but the design is compatible with TimeCrypt [7] to work on encrypted data.

We make the following contributions in our thesis:

- **Design of Access Pattern Hiding Time-Series Data Store:** We design an efficient time-series data storage system that hides the time window as well as the stream a client queries from the replicated storage providers. We propose two optimizations to improve the efficiency of the baseline design.
- **C++ Implementation:** We provide an efficient implementation of our system in C++. The implementation consists of a client and a server library, compatible to run in cloud scale infrastructure.
- **Evaluation:** We provide a thorough evaluation of our system in a realistic cloud deployment. We provide insights on the feasibility and scalability of individual components and the end-to-end system. Our evaluation shows that our system can execute realistic queries on 1M

records in under 1.5s, allowing us to handle small- to medium-sized databases within reasonable time bounds.

1.2 Outline

In this thesis, we first discuss background information relevant to the details of our system's design. In Chapter 3 we review relevant work for privacy-preserving data processing of time-series data. We proceed by discussing the core objectives of our system. We then outline the design in Chapter 5. We continue with the implementation and end with the evaluation of our system, running a variety of benchmarks to assess its feasibility and scalability.

Chapter 2

Background

In this section, we discuss the relevant background for this thesis. First, we provide an overview of time-series data and then present the concept of Function Secret Sharing.

2.1 Time-Series data

Time-series data is a sequence of data points indexed by time. For example, most sensors produce time-series data streams by collecting measurements, or in cluster monitoring where time-series logs are used to ensure availability.

Compared to other types of data, time-series workloads have unique characteristics that allow for efficient storage and processing designs:

- **Append-only:** Data is almost exclusively append-only. For example, a sensor always appends the latest measurement to its data stream [24].
- **Immutability:** Time-series data mostly consists of machine-generated observations that do not change and thus once a record has been written its value is rarely modified at a later point in time [7, 24].
- **High-volume:** Data is written at an extremely high rate and must support millions of writes per day. For example, a satellite continuously collecting and streaming telemetry produces several hundred million records a day [14].
- **Aggregation queries:** Most applications, require analysis aggregated along the time dimension as, for example, a metrics dashboard of a website showing the number of daily requests throughout a year [15].
- **Data decay:** Recent data is most valuable. This is the case when querying aggregated data as mentioned above where recently appended

data is more often queried than older ones. This can be seen for example in a stock market, where minute-granularity of a past day is preferred over the same granularity in a day a week prior [24].

These characteristics can be taken advantage of to build database management systems (DBMS) tailored to time-series data. For example, as data gets older one can replace specific entries with aggregations, decreasing the resolution of queries and thus keeping the storage footprint low [7, 18].

Many modern DBMS have been designed which add additional features to improve performance on time-series workloads, such as TimescaleDB [25] and InfluxDB [16]. These DBMSs add specialized indexes over their own data structures, chunks and shards respectively, to further increase the efficiency of statistical queries.

2.2 Function Secret Sharing

In this section, we describe the concept of Function Secret Sharing [4, 5], which is a core building block of our design.

In essence, Function Secret Sharing (FSS) provides a way for a client to split a function into function shares such that no strict subset reveals any information about the original function. The client can, however, combine the evaluation of these individual shares on a specific input to reconstruct the evaluation of the actual function on the same input.

More specifically, FSS provides a way to split a function f from a function class $\mathcal{F}$ into m succinct function shares $f_1, \dots, f_m$ such that $\sum_{i=1}^{m} f_i = f$ holds and where any strict subset of the function shares computationally hides f.

An m-party FSS scheme is a pair of algorithms (**Gen**, **Eval**) which, for a function class $\mathcal{F}$, are defined in the following way:

- $\mathbf{Gen}(1^\lambda, f) \rightarrow (k_1, \dots, k_m)$. Given a security parameter 1^λ and a function description $f \in \mathcal{F}$, returns m keys.
- $\mathbf{Eval}(i, k_i, x) \rightarrow y_i$. Given a party index i, key k_i and string x, returns the party's share y_i of $f(x)$.

Thus, any FSS scheme can be divided into a key generation and evaluation function. Both algorithms should be computationally efficient and have succinct key sizes. Furthermore, the generated shares should not reveal anything about the underlying function f but when all shares $f_i(x)$ are additively recombined $f(x)$ is revealed. The computational complexity and the key size of the scheme depend on the function class $\mathcal{F}$ and their construction, of which we discuss relevant ones in the following section.

If an FSS scheme is correct then $\mathbf{Gen}(1^\lambda, f)$ is guaranteed to generate m keys $k_1, k_2, \dots, k_m$, which when evaluated with $\mathbf{Eval}(i, k_i, x)$ and their results aggregated, return the same result as the original function. If an FSS scheme is secure then an adversary given a function description f and a strict subset p of the m keys generated by $\mathbf{Gen}(1^\lambda, f_a)$ can distinguish with negligible probability if $f = f_a$.

2.2.1 Efficient Constructions of FSS Schemes

We now discuss two FSS constructions relevant to our design: distributed point functions and distributed comparison functions. Both in the 2-party setting, which we make extensive use of in our design.

Distributed Point Function

Point functions are a family of functions where all but one input evaluate to zero. More precisely, $\mathcal{F}$ is the family of point functions $f_{a,c}$ where the following holds:

$$f_{a,c}(x) = \begin{cases} c & \text{if } x = a \\ 0 & \text{otherwise.} \end{cases}$$

A distributed point function (DPF) scheme $(\mathbf{Gen}^\bullet, \mathbf{Eval}^\bullet)$ is a way to secret share a vector of 2^n elements in which only one element at index a is c. Here, the vector represents the function table of $f_{a,c}$ where the vector indexes are the possible function inputs and the vector values are the respective function outputs. However, simply secret sharing the whole function table is highly inefficient. Boyle et al. [4] provide an efficient construction of a 2-party DPF, where they reduce the size of the secret share to $\mathcal{O}(\lambda n)$, where λ is the security parameter.

In essence, the $\mathbf{Gen}^\bullet$ algorithm returns two keys k_1, k_2 that define two binary trees of depth n with a pseudo-random string at each node, where n represents the size of the input in bits. The two binary trees are identical except for the path from the root to the selected point a, where the strings are chosen pseudo-randomly and independently from each other. $\mathbf{Eval}^\bullet(i, k_i, x)$ traverses these binary trees from the root to $x = x_1, \ldots, x_n$ computing the share's evaluation along the path, with respect to an Abelian group $\mathbb{G}$. The shares' evaluations on inputs not equal to the target a will cancel each other out when added together with respect to $\mathbb{G}$, resulting in the additive combination of the shares to equal c only if the input is equal to the target value.

Boyle et al. [5] further improve this scheme by shrinking the key size by a factor of 4.

Distributed Comparison and Interval Function

A distributed comparison function (DCF) is an FSS construction for a function that evaluates to a non-zero value if the input is smaller than a selected value and 0 otherwise. That is, $g_{a,c}$ is the family of comparison functions where the following holds:

$$g_{a,c}(x) = \begin{cases} c & \text{if } x < a \\ 0 & \text{otherwise.} \end{cases}$$

Boyle et al. [4] provide a construction similar to the one for DPFs, where, given 2 keys, when evaluating both binary trees and additively combining the results the target value c is computed when the input value is smaller than a and 0 otherwise. This construction increases the secret share size to $\mathcal{O}(\lambda n + n \log |\mathbb{G}|)$.

DCFs can trivially be extended to support the class of distributed interval functions $h_{a,b,c}$, which are intuitively defined as follows:

$$h_{a,b,c}(x) = \begin{cases} c & \text{if } a \leq x < b \\ 0 & \text{otherwise.} \end{cases}$$

The construction is achieved by computing the difference between the two distributed comparison functions $g_{b,c} - g_{a,c}$, thus resulting in a factor of 2 overhead.

Chapter 3

Related Work

In this chapter, we review relevant work for privacy-preserving data processing of outsourced time-series data in cloud storages.

We first discuss relevant work that uses advanced encryption techniques to protect the privacy of time-series data but does not provide query privacy. In the second part, we discuss systems that employ Function Secret Sharing as a technique to hide query patterns from a data provider.

3.1 Private Time-Series Data Processing

In the following, we present TimeCrypt [7] and Zeph [8], which are both systems that provide queries on encrypted time-series data.

3.1.1 TimeCrypt

TimeCrypt [7] is a system that provides scalable and real-time analytics over large volumes of encrypted time-series data through a careful design of cryptographic primitives tailored for times series data. This competitive performance is achieved by the following techniques:

- **Different keys for every timestamp:** To allow for fine-grained access control, each segment in the time-series database is encrypted with a different encryption key.
- **Homomorphic Encryption:** Through the use of an additively homomorphic symmetric encryption scheme, TimeCrypt is able to natively support aggregation queries. These aggregates are computed by the server by summing up all ciphertexts in a given time range which a client can later decipher.
- **Key Canceling:** To decrypt the aggregated ciphertext, a client would need to additively combine all keys used in the target range resulting

in local computation being linear in the number of aggregated ciphertexts. Since most time-series queries are over a continuous range in time, this can be reduced to a constant by choosing individual encryption keys such that the inner keys cancel each other out during aggregation.

Although the plaintext values are hidden from the server through semantically secure encryption, TimeCrypt is non-oblivious. That is, it does not hide the access patterns of a client since the server accesses only the time range requested by the client. For example, if a client queries an aggregate of a specific month in TimeCrypt, the server does not learn the result of the query but learns that the client issued an aggregation query for that specific month.

3.1.2 Zeph

Zeph [8] is a system that enables users to set privacy preferences on how their data can be shared and processed, which through the use of cryptographically enforced privacy transformations, guarantees that data consumers are not able to access more data than is granted to them. Zeph is optimized for streaming data, thus a fit for time-series workloads, scaling to thousands of data sources and allowing support for large-scale low-latency data stream analytics. This is achieved through the following approaches:

- **Decoupling Encryption from Privacy:** Zeph decouples the end-to-end encrypted data stream, in the data plane, from the policy enforcement logic, in the privacy plane. This separation ensures that a data producer can encrypt its data without adhering to a specific privacy policy.
- **Homomorphic Secret Sharing:** The decoupling of these two planes is achieved with the help of additively homomorphic secret sharing (HSS) [6, 3]. HSS allows computing a function on secret shared messages by combining the outputs of a separate function applied on the individual secret shares.
- **Transformation Tokens:** A privacy plane controller is capable of authorizing a transformation on the encrypted data by computing a so-called *transformation token* using HSS. These tokens are computed by applying the same transformation to the individual encryption keys of the target values. This computation is performed independently from the ciphertext transformation. The token can then be used to reveal the plaintext.

Similarly to TimeCrypt, Zeph is non-oblivious, allowing an adversary to learn which parts of the data are queried.

3.2 Private Queries with Function Secret Sharing

We continue with Splinter [27] and Dory [11], which are systems that hide access patterns with the help of Function Secret Sharing.

3.2.1 Splinter

Splinter is a practical relational database system that hides user query patterns from *honest-but-curious* database hosts. Splinter supports a subset of the SQL language on top of public datasets including aggregation function `COUNT`, `SUM`, `AVG` and more complex queries such as `MAX` and `TOPK`, which returns up to k individual records matching a predicate.

Splinter assumes that the database is replicated among at least two non-colluding database hosts. To hide the query patterns, Splinter builds on efficient function secret sharing schemes for distributed point and comparison functions. Queries are compiled into FSS shares that each server executes on the whole database to select records of interest. Due to its design with FSS, the system does not hide the type of the query and the column names that are being filtered and selected.

Splinter achieves practical performance with an improved FSS implementation that leverages modern AES-NI [19] instructions, achieving more than 2x higher performance than a naïve implementation.

Though, Splinter supports SQL queries on public relational databases, Splinter does not scale to high volumes of time-series data and is not compatible with encrypted data processing.

3.2.2 Dory

Dory [11] addresses the need for a system with practical server-side keyword search on encrypted documents that does not leak document access patterns. Dory splits search queries between multiple servers, using a DPF [4], to protect against an attacker who controls all but one of the servers. Dory achieves high performance, searching over 1M documents in under 1s. This is achieved with the following techniques:

- **Encrypted Bloom Filter:** To support efficient search, a server builds a table where every row corresponds to a bitmap of words for a document and every column represents a different keyword. The storage footprint of these bitmaps is reduced by using a bloom filter for compression. Furthermore, these bloom filters are encrypted before being inserted into the table to prevent an attacker from immediately reconstructing the entire plaintext search index if all servers have been compromised.

- **Distributed Point Function:** To search for a keyword within a folder of documents, a client splits the request into DPF shares and sends them to separate servers, which then evaluate their share on every column of the table of encrypted bloom filters, thus hiding the keyword access patterns. Evaluations are XORed and sent back to the client.

Dory improves the linear scan by only executing its DPF on subfolders. This is the consequence of the tradeoff in reducing the number of DPF evaluations a server must perform from all documents to a subset only in order to increase performance. However, this approach only hides access patterns for keyword searches within a specific subfolder and thus the subfolder accesses are leaked to the database provider.

Chapter 4

Objectives

In this chapter, we discuss the scenario our system fits in and the goals, our system should achieve. We continue with a description of the threat model we consider in our scenario.

4.1 Scenario

For our system we consider a standard time-series pipeline scenario, as depicted in Fig. 4.1. A time-series pipeline consists of three main actors, data producers, a database server and data consumers. A data producer is an entity, usually with limited computational performance and storage space, such as an industry monitoring sensor or Internet of Things (IoT) device [2], which continuously generates time-series data. The data producer continuously sends its generated data to the remote database server for storage. The database server stores time-series streams and appends incoming records

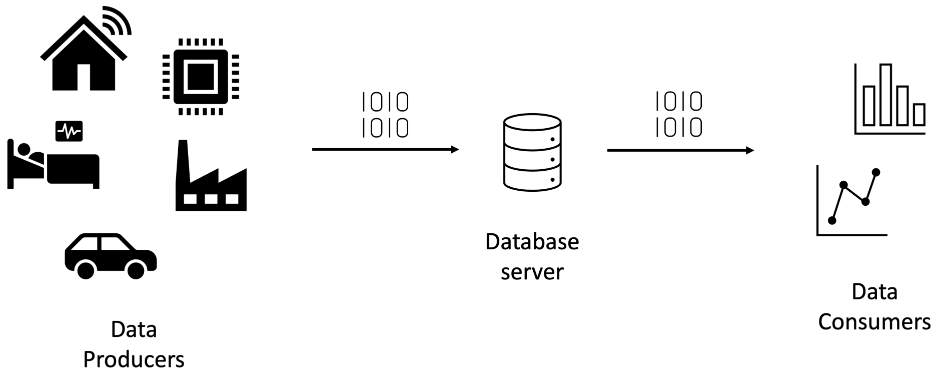

Figure 4.1: Time-series database setting. Devices produce large volumes of data which are sent to storage servers to later be used by various data consumers.

accordingly. A data consumer (e.g. health service) issues statistical aggregation queries to the database server to extract utility. For example, a health service issues queries to visualize the time-series stream in a dashboard application. These queries often take the following form expressed in SQL:

```
SELECT agr(*) FROM streams WHERE $t_1 \leq t < t_2$
AND stream_id $= s'$
```

where `agr` denotes a supported statistical aggregate query such as `SUM` or `AVG`, `streams` the target table of the database, $[t_1, t_2[$ the target time range, `stream_id` the id of individual streams and s' the target stream id. A stream may be used to separate different types of measurements. For example, a health service may differentiate clients' heart rates by assigning them different stream ids.

Statistical queries are a key necessity in time-series workloads. For example, a health service storing clients' heart rates may offer an app that displays a dashboard of average heart rates over certain periods of time. Queries from the app to the health service would include many queries for the client's average heart rate.

4.2 Goals

We aim to build a system within this scenario with the following properties:

- **Hiding Query Access Patterns:** The main goal of our system is to hide query access patterns issued by the data consumers from the database server. Based on query patterns, storage operators may learn detailed information about the underlying data even if the data is protected with encryption or other tools. In time-series workloads query access patterns reveal which stream and which timestamps of the stream a particular data consumer queries.

  In our system, we aim to hide this information from the storage operators. Specifically, given the query structure that we consider (see section 4.1), we aim to hide the queried time range $[t_1, t_2[$ and stream s'. The system, however, does not hide all access patterns. The database server learns what type of statistical query `agr` a client has executed as well as a client's target table `streams`. For example, a health service providing a doctor access to a client's heart rates will neither learn the specific time range $[t_1, t_2[$ queried by the doctor, nor the client id whose data is being accessed. The server will, however, learn that a query has been made by the doctor and the type of the statistical query.

- **Support for Statistical Queries:** Another crucial goal is for our system to support a broad range of statistical queries, common in time-series workloads. In particular, we aim to support queries of the following form:

```
SELECT agr(*) FROM streams WHERE t1 ≤ t < t2
AND stream_id = s'.
```

where agr is an aggregation query such as SUM or VAR, streams the target table of streams in the database, $[t_1, t_2[$ the target time range, stream_id the id of individual streams and s' the target stream id.

- **Practical Scalability:** We aim to build a system that scales to practical database sizes and can handle realistic time-series applications.
- **Compatibility with TimeCrypt:** Our system does not provide confidentiality, i.e., the server can observe data in the clear. However, we aim to be compatible with TimeCrypt [7].

4.3 Threat Model

Our system assumes that there are at least two non-colluding servers operating identical databases. We consider a passive adversary that can statically corrupt all but one server. The adversary can observe all processes and interactions with the compromised server(s) but does not deviate from the protocol.

4.4 Access Pattern Leakage

Our system aims to hide access patterns, but each query leaks a confined amount of information to the adversary. If a client executes a query, the adversary learns the following:

- **A query is being executed:** Once a query has been made, the database servers know that the client is requesting some data from the database. This could be mitigated by having the client constantly send 'dummy' queries (for example generated randomly) at a high rate regardless of whether they want to retrieve data from the database or not. In this case, the server would constantly be executing queries, either dummy or relevant, sent by the client.
- **Query Aggregate:** We do not hide the type of statistical query the client performs. For example, if the client requests the sum over a target interval, the database servers learn that the client is requesting the sum of said interval but do not however learn which interval is targeted. This can be mitigated by querying all statistical digests (see subsection 5.2.1).

- **Number of Digest Elements:** We do not hide the number of digest elements and their type stored in every entry of the database. The database providers thus know which aggregation type each digest element corresponds to.

Chapter 5

Design

In this chapter, we present our system, which enables private queries on time-series data by hiding database record access patterns and still achieving low-latency requirements for small database sizes. We begin with an overview of the system and continue with an in-depth explanation of each component.

5.1 Overview

Our system builds on typical time-series database designs [1, 7, 16], which augment a key-value store with further logic and APIs optimized for time-series workloads. As illustrated in Fig. 5.1, Our system consists of a three components. A client, which compiles requests from data consumers to equivalent FSS shares and sends each of them to a different replicated data-

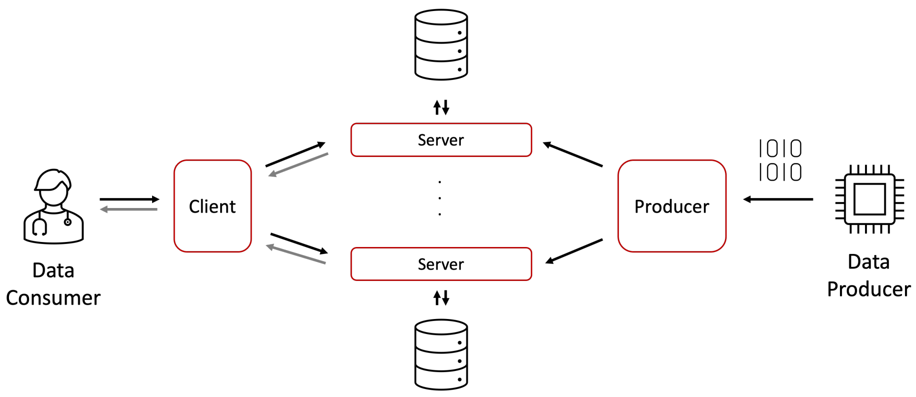

Figure 5.1: Overview of a data consumer's query, which is sent to the client component, which compiles it into query shares that are sent to several replicated servers. Results of the evaluated shares are sent back to the client component, combined and returned to the data consumer. The producer component sends the data producers' generated data to the replicated servers.

base server via the Server API. These server components take the generated shares, and evaluate them on all database records before sending results back to the client, which aggregates all share results into a final query result returned to the data consumer. The producer component sends time-series data streams generated by data producers to all replicated servers.

Due to the properties of FSS, the system hides the client access patterns to database records from the database servers as long as one provider is honest and does not collude with others.

5.1.1 Client, Server and Producer System Components

The client allows a data consumer to execute queries of the following form:

```
SELECT agr(*) FROM streams WHERE $t_1 \leq t < t_2$
AND stream_id $= s'$
```

where agr can be one of five supported query types: COUNT SUM, sum of squares, AVG (average) and VAR (variance), often used in time-series workloads [8, 27]. A data consumer creates queries with a target time range $[t_1, t_2[$, stream id s' (discussed in subsection 5.2.2) and a table streams as parameters and sends them to the system's client via its API.

Given a query type and parameters, the client generates as many query shares as there are replicated database servers and sends each to a different database server via the servers' API. Each server evaluates its share on all pairs of stream ids and timestamps in the database and returns its result. The client then uses modular addition to aggregate the returned results from the query shares into one final result. If the query is one of the three basic types (COUNT, SUM, sum of squares) then the result is forwarded to the data consumer, otherwise, further computation is performed for AVG and VAR request, before being delivered to the data consumer.

5.2 Stream Abstractions

We continue with a detailed discussion of the techniques used for handling and storing time-series data as well as components needed to execute a private query.

5.2.1 Client Data Serialization & Encoding

We introduce a custom serialization and value encoding technique to address the scalability challenges of applying FSS to time-series data. Data producers write to the time-series databases through the producer component, which provides an API to the data producers and processes the generated data stream in predefined time windows before forwarding it to the

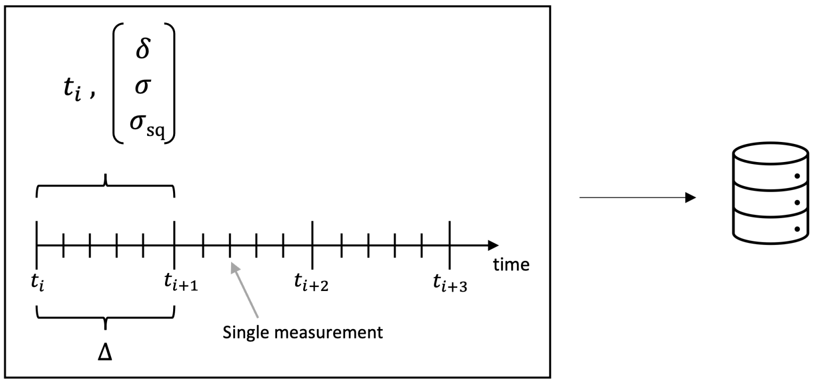
t_i, $\begin{pmatrix}\delta\\ \sigma\\ \sigma_{sq}\end{pmatrix}$
time
t_i
t_{i+1}
t_{i+2}
t_{i+3}
Single measurement
Δ

Figure 5.2: Encoding used to store time-series data, where for each time chunk $\Delta = t_{i+1} - t_i$ a statistical summary of the count δ, sum σ and sum of squares σ_{sq} of the underlying data is stored.

database servers [7]. As illustrated in Fig. 5.2, for each time window $[t_i, t_{i+1}[$ of predefined size $\Delta = t_{i+1} - t_i$, the producer component of our system aggregates the values within the time window and computes a digest consisting of statistical summaries of the underlying data. This digest enables us to perform statistical queries efficiently.

Efficient FSS constructions are limited to additive operations (i.e., no multiplications) on the server side, and, therefore, limit the queries a system can support. To overcome this issue, we encode the aggregated data such that it consists of a vector of three separate statistical summaries: the number of individual underlying data points δ, the sum of data points σ and the sum of squares of the underlying data σ_{sq} of a time window $[t_i, t_{i+1}[$.

In combination with client-side computation, we can support a wider range of aggregate queries. We compute an `AVG` query by requesting δ and σ of a target time range and computing $\frac{\sigma}{\delta}$ on the client side. Furthermore, we can support variance queries by requesting all three digest elements and performing the following computation:

$$VAR_{t_1,t_2} = \frac{\sigma_{sq} - 2 \cdot \frac{\sigma^2}{\delta}}{\delta} + \left(\frac{\sigma}{\delta}\right)^2 \tag{5.1}$$

where VAR_{t_1,t_2} is the underlying data's variance of the time-window $[t_i, t_{i+1}[$.

5.2.2 Server Storage Layout

We design our system to support multiple streams of time-series data as is typical in time-series workloads [17, 26]. Streams are used to allow the system to differentiate measurements taken at the same point in time. This can be used for example to differentiate heart rate data produced by separate

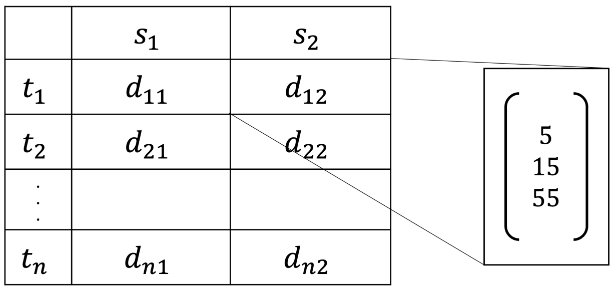

Figure 5.3: Layout of a database server where each digest vector d_{ij} is stored within a row associated with a timestamp t_i and column associated with a stream s_i.

users over the same time period or different types of measurements taken by an industry sensor, such as temperature or relative humidity.

As illustrated in Fig. 5.3 we model our databases store of time-series data as a table where each columns represents a specific stream s_i containing digests of different points in time t_i. Each entry in the table indexed by (t_i, s_i) stores a digest value containing statistical summaries over the time range $[t_i, t_{i+1}[$ of stream s_i.

In our system, we maintain this table in a key-value store. Each digest is stored with the key consisting of the stream id s_i appended to the timestamp t_i, denoted with $s_i||t_i$, and where all three statistical summaries are appended to each other to form the key's value.

5.3 Enabling Private Queries

We now discuss how our system integrates FSS [4] to hide query patterns from the database operators. FSS is a tool that allows to split a function f into function shares $f_1, f_2, \ldots, f_n$ so that multiple parties can evaluate f without revealing which function f has been shared (see subsection 2.2.1). More precisely, an FSS scheme is a pair of algorithms (**Gen**, **Eval**), where given a function description, **Gen** generates n keys which can each be used to independently evaluate a share of the original function using **Eval**. All share results can then be recombined to equate to the original function. We begin by giving an overview of how a query is executed in our system using FSS and continue with an example of a `SUM` query.

5.3.1 Query Overview

A data consumer can create a query of the following form expressed in SQL:

```
SELECT agr(*) FROM streams WHERE $t_1 \leq t < t_2$
AND stream_id = $s'$
```

where `agr` denotes one of the five queries supported by our system (see section 5.2.1), s' denotes the target stream id, $[t_1, t_2[$ denote the queried time range and `streams` the name of the database table containing our data streams.

To hide the time-range and the selected stream id of a query, we construct a selection function for each query that returns 1 when given a stream id s equals to s' and timestamp t within the predetermined time-window $[t_1, t_2[$ and 0 otherwise:

$$h_{t_1,t_2,s'}(s||t) = \begin{cases} 1 & \text{if } t_1 \leq t < t_2 \land s = s' \\ 0 & \text{otherwise} \end{cases}$$

where $||$ denotes string concatenation. The selection function outputs 1 if the pair (s, t) should be in the aggregation for the query and 0 if not. Thus $h_{t_1,t_2,s'}(s||t)$ is an interval function (see section 2.2.1). The server executes this selection function on each timestamp and stream id pair (s_j, t_k) in the storage database and multiplies it with the stored digest at timestamp k in stream j:

$$h_{t_1,t_2,s'}(s_j||t_k) \cdot \gamma_{j,k}$$

where $\gamma_{j,k}$ denotes any one of the three statistical summaries in a digest (see subsection 5.2.1).

Since the server stores the stream id s_j concatenated to the timestamp t_k as its key to a database record, we use string concatenation to pass both the string id and the timestamp to the interval function. This prevents us from needing to evaluate two separate interval functions, one for the stream and one for the timestamp. Furthermore, it guarantees a lexicographic ordering to all stream id and timestamp pairs allowing us to directly compare time ranges within a stream with a single interval function.

Thus the database server evaluates the selection function on each stream id and timestamp pair of the database and multiplies the result with its corresponding digest entry:

$$Q(t_1, t_2, s', \gamma) = \sum_{j=1}^{m} \sum_{k=1}^{l} h_{t_1,t_2,s'}(s_j||t_k) \cdot \gamma_{j,k}$$

where Q denotes our query which we perform over the time-window $[t_1, t_2[$ on stream with id s', which retrieves digest element $\gamma \in \{\delta, \sigma, \sigma_{sq}\}$ and where m is the total number of streams and l the total number of timestamps within our database.

Making Use of Function Secret Sharing

So far, the database server can observe the output of the selection function. To hide the output from the server we apply function secret sharing on the selection function. In particular, we can construct the selection function with a distributed comparison function (DCF) (see subsection 2.2.1):

$$Q(t_1, t_2, s', \gamma) = \sum_{j=1}^{m} \sum_{k=1}^{l} (g_{t_2,s'}(s_j||t_k) - g_{t_1,s'}(s_j||t_k)) \cdot \gamma_{j,k}$$

where $g_{t_\alpha,s'}$ is defined as:

$$g_{t_\alpha,s'}(s_j||t_k) = \begin{cases} 1 & \text{if } t_k < t_\alpha \wedge s_j = s' \\ 0 & \text{otherwise} \end{cases}$$

and $t_\alpha \in \{t_1, t_2\}$.

Furthermore, we assume all computations over function shares are done over $\mathbb{Z}_{2^q}$ where q is the number of bits in the output range.

The client uses the FSS algorithm $\textbf{Gen}(1^\lambda, f) \rightarrow (k_1, \ldots k_n)$, where 1^λ is a security parameter and f a function description to generate key shares to be used with $\textbf{Eval}(i, k_i, x) \rightarrow y_i$ to evaluate the i-th key share on the input x. For FSS schemes used for comparison functions we denote the pair of algorithms as $(\textbf{Gen}^<, \textbf{Eval}^<)$. Thus the client uses $\textbf{Gen}^<(1^\lambda, g_{t_\alpha,s'})$ twice to efficiently generate two sets of keys $\{k_{11}, k_{12}, \ldots k_{1n}\}$ and $\{k_{21}, k_{22} \ldots, k_{2n}\}$, where n is the number of replicated database servers, to later be used to evaluate shares of the two comparison functions. Furthermore the $\textbf{Eval}^<$ algorithms have the following property:

$$g_{t_\alpha,s'}(s||t) = \sum_{i=1}^{n} \textbf{Eval}^<(i, k_{\alpha i}, (s||t))$$

which we use for our query:

$$\begin{aligned} &\sum_{j=1}^{m} \sum_{k=1}^{l} (g_{t_2,s'}(s_j||t_k) - g_{t_1,s'}(s_j||t_k)) \cdot \gamma_{j,k} \\ =&\sum_{j=1}^{m} \sum_{k=1}^{l} (\sum_{i=1}^{n} \textbf{Eval}^<(i, k_{2i}, (s_j||t_k)) - \sum_{i=1}^{n} \textbf{Eval}^<(i, k_{1i}, (s_j||t_k))) \cdot \gamma_{j,k} \end{aligned}$$

which we can further rearrange to push our function share loop to the outside:

$$Q(t_1, t_2, s', \gamma) = \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{l} (\textbf{Eval}^<(i, k_{2i}, (s_j||t_k)) - \textbf{Eval}^<(i, k_{1i}, (s_j||t_k))) \cdot \gamma_{j,k}. \tag{5.2}$$

As illustrated in Fig. 5.4, given n replicated servers and two sets of n key shares, $\{k_{11}, k_{12}, \ldots k_{1n}\}$ and $\{k_{21}, k_{22} \ldots, k_{2n}\}$, each key share pair (k_{1i}, k_{2i})

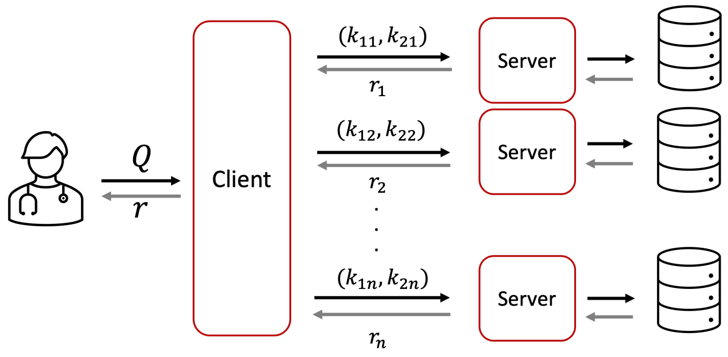

Figure 5.4: Overview of a query where the client component generates n key pairs (k_{1i}, k_{2i}) from the data consumer's request to send to the database servers. All servers evaluate their key pairs on all their pairs of timestamps and stream ids before sending their result r_i back to the client to be aggregated into the final result r.

can be sent to one of the n database servers. Each server can evaluate its pair of key shares independently on all pairs of timestamps and stream ids and return the result back to the client, which aggregates all individual server results. This process is described in further detail in our following example of a `SUM` query.

5.3.2 Example: `SUM` Query

We focus on the details of a sum query first illustrated in Fig. 5.5 and later discuss the other types of queries supported by our system.

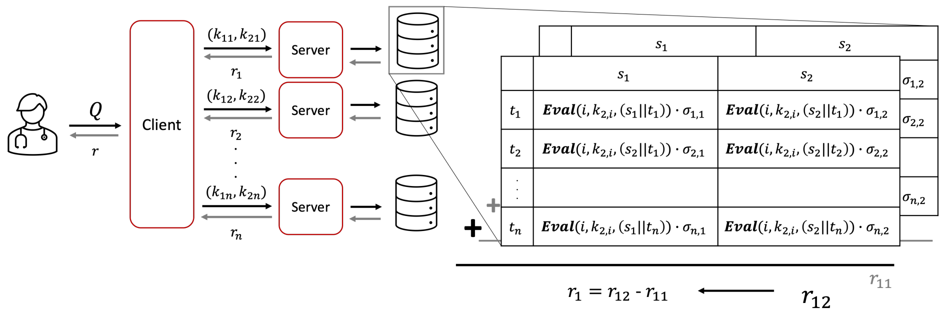
Q
r
Client
(k11, k21)
r1
(k12, k22)
r2
(k1n, k2n)
rn
Server
Server
Server
s1
s2
t1
t2
tn
Eval(i, k2,i, (s1||t1)) · σ1,1
Eval(i, k2,i, (s2||t1)) · σ1,2
Eval(i, k2,i, (s1||t2)) · σ2,1
Eval(i, k2,i, (s2||t2)) · σ2,2
Eval(i, k2,i, (s1||tn)) · σn,1
Eval(i, k2,i, (s2||tn)) · σn,2
σ1,2
σ2,2
σn,2
+
r11
r1 = r12 - r11
r12

Figure 5.5: Process of a `SUM` query, which is split into function shares using FSS by the client component and sent to individual database servers to be evaluated on all pairs of timestamps and stream ids in the database. Results are sent back to the client to be aggregated.

A data consumer creates a ranged `SUM` query of the following form:

```
SELECT SUM(*) FROM streams WHERE $t_1 \leq t < t_2$
AND stream_id $= s'$
```

where t_1 and t_2 denote the queried time-window, s' the target stream id and `streams` the name of the database table which in our case contains all database entries. The query is given to the client component of our system for further processing and denoted as $Q(t_1, t_2, s', \sigma)$.

Client Side Execution

Our client component aims to create a distributed interval function that returns 1 if evaluated on a timestamp and stream id that is within the time-window $[t_1, t_2[$ and stream with id s'. This is achieved by using two comparison functions for the two intervals $[0, t_1[$ and $[0, t_2[$.

The client uses the FSS algorithm $\mathbf{Gen}(1^\lambda, f)$ to generate the two sets of key shares $\{k_{11}, k_{12}, \ldots k_{1n}\}$ and $\{k_{21}, k_{22} \ldots, k_{2n}\}$ for the n replicated database servers. The client then sends a key pair (k_{1i}, k_{2i}) as well as additional meta-data, such as the input domain size of the $\mathbf{Eval}^<$ algorithm, to all n servers.

Server Side Execution

The server component receives the pair of function keys and evaluates them in the following way. Each key k_{1i} and k_{2i} is used to evaluate every timestamp and stream id pair (s_j, t_k) in the database with the **Eval** algorithm, denoted as $\mathbf{Eval}^<(i, k_{\alpha i}, (s_j||t_k))$ where $\alpha \in \{1, 2\}$. Every evaluation with individual keys are separately added together and then the smaller is subtracted from the larger. More precisely the database server i performs the following computation on a database with m streams and l separate timestamps:

$$\begin{aligned} r_i &= \sum_{j=1}^{m} \sum_{k=1}^{l} \mathbf{Eval}^<(i, k_{2i}, (s_j||t_k)) \cdot \sigma_{j,k} - \sum_{j=1}^{m} \sum_{k=1}^{l} \mathbf{Eval}^<(i, k_{1i}, (s_j||t_k)) \cdot \sigma_{j,k} \\ &= \sum_{j=1}^{m} \sum_{k=1}^{l} \mathbf{Eval}^<(i, k_{2i}, (s_j||t_k)) \cdot \sigma_{j,k} - \mathbf{Eval}^<(i, k_{1i}, (s_j||t_k)) \cdot \sigma_{j,k} \end{aligned}$$

where $\sigma_{j,k}$ denotes the sum entry of the jth stream at timestamp k and every addition is modular over $\mathbb{Z}_{2^q}$ where q is the number of bits in the output range.

Intuitively, one can see that a sum $\sigma_{j,k}$ is only included in the final result if its timestamp is in the queried range and in the corresponding stream since only then will $\mathbf{Eval}^<(i, k_{2i}, (s_j||t_k)) - \mathbf{Eval}^<(i, k_{1i}, (s_j||t_k))$ return 1. The result r_i is then returned to the client.

The client then adds up all individual results received from the database servers with modular addition and returns the final result r to the data consumer:

$$r = \sum_{i=1}^{n} r_i.$$

Due to the fact that every sum entry $\sigma_{j,k}$ of the databases was accessed and that FSS schemes computationally hide the original comparison functions, the servers can only know that a `SUM` query was performed but neither have access to the specific time range nor stream requested by the client.

5.3.3 Further Supported Queries

`COUNT` and sum of squares queries work analogously to our `SUM` example except that the respective δ or σ_{sq} summary of the digest is multiplied with the key share evaluation $\mathbf{Eval}^{<}(i, k_{\alpha i}, (s_j||t_k))$.

Our design additionally supports averages and variances of time ranges. An `AVG` query is performed by querying the servers for the count and sum of the target time range and dividing the sum by the count. A `VAR` query is performed by requesting all three digest components and performing the computation from Equation 5.1.

5.4 Security Analysis

In this section, we provide arguments for the correctness and privacy of our system. We argue that if a secure FSS DCF construction is employed, our system both hides certain parameters of queries as well as produces the correct query result.

5.4.1 Correctness

We argue that if the construction of an FSS scheme for a distributed comparison function provided by Boyle et al. [4, 5] is correct and the adversary is passive and follows the protocol, then a query given to our system's client by a data consumer will be returned with a correct result.

If said FSS scheme construction is correct, the following relationship between function and function shares holds (see subsection 5.3.1):

$$g_{t_\alpha,s'}(s||t) = \sum_{i=1}^{n} \mathbf{Eval}^{<}(i, k_{\alpha i}, (s||t))$$

Furthermore, we have seen that a query in our system amounts to the following equation using comparison functions:

$$Q(t_1, t_2, s', \gamma) = \sum_{j=1}^{m} \sum_{k=1}^{l} (g_{t_2,s'}(s_j||t_k) - g_{t_1,s'}(s_j||t_k)) \cdot \gamma_{j,k}$$

which we show is equivalent to Equation 5.2:

$$\begin{aligned}
&\sum_{j=1}^{m} \sum_{k=1}^{l} (g_{t_2,s'}(s_j||t_k) - g_{t_1,s'}(s_j||t_k)) \cdot \gamma_{j,k} \\
=&\sum_{j=1}^{m} \sum_{k=1}^{l} (\sum_{i=1}^{n} \mathbf{Eval}^{<}(i, k_{2i}, (s_j||t_k)) - \sum_{i=1}^{n} \mathbf{Eval}^{<}(i, k_{1i}, (s_j||t_k))) \cdot \gamma_{j,k} \\
=&\sum_{j=1}^{m} \sum_{k=1}^{l} (\sum_{i=1}^{n} \mathbf{Eval}^{<}(i, k_{2i}, (s_j||t_k)) - \mathbf{Eval}^{<}(i, k_{1i}, (s_j||t_k))) \cdot \gamma_{j,k} \\
=&\sum_{j=1}^{m} \sum_{k=1}^{l} \sum_{i=1}^{n} \mathbf{Eval}^{<}(i, k_{2i}, (s_j||t_k)) \cdot \gamma_{j,k} - \mathbf{Eval}^{<}(i, k_{1i}, (s_j||t_k)) \cdot \gamma_{j,k} \\
=&\sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{l} (\mathbf{Eval}^{<}(i, k_{2i}, (s_j||t_k)) - \mathbf{Eval}^{<}(i, k_{1i}, (s_j||t_k))) \cdot \gamma_{j,k}.
\end{aligned}$$

Thus if the FSS scheme construction for a distributed comparison function $\mathbf{Eval}^{<}$ is correct, then our system returns the correct query result.

5.4.2 Privacy

We argue that if the DCF construction is a secure FSS scheme for distributed comparison functions, then the adversary does not learn the output of the query nor certain of its parameters.

If the FSS construction for a distributed comparison function is secure, then the keys $k_{\alpha i}$ generated by $\mathbf{Gen}^{<}(1^{\lambda}, g_{t_1,s'})$ and $\mathbf{Gen}^{<}(1^{\lambda}, g_{t_2,s'})$ computationally hide t_1, t_2 and s'. Since these keys are the only information we share with the replicated servers, this implies that our system computationally hides the parameters t_1, t_2 and s' from our database servers as long as at least two of them do not collude.

Although we hide t_1, t_2 and s' from our database servers, they do learn that a query was executed and the type of aggregation that was requested, be it the count, the sum or the sum of squares. Both of these leaks could be mitigated by constantly having the client component of our system send 'dummy' queries regardless of whether the data consumer wants a query to be executed or not. Furthermore, one could mitigate leaking the aggregation type by always requesting all three components of the digest.

5.5 TimeCrypt Compatibility

Though our system does not hide digest values, it is compatible with TimeCrypt [7] to hide them. We use a similar stream abstraction and their employed homomorphic encryption is compatible with our function secret sharing approach, though, the plaintext multiplications and additions we use must be replaced with their respective operations on ciphertexts.

5.6 System Optimizations

In our design, the server has to perform a linear scan of the full database for each query. Thus, with increased database size, the linear scan is a dominant performance bottleneck. In this section, we present two optimizations to our baseline design to mitigate the performance hit from this linear scan. The two optimizations include batching multiple queries to decrease the number of round trips taken between client and server and the use of multithreading to parallelize the key pair evaluations on the database servers.

5.6.1 Parallelism

Though a linear scan is necessary to evaluate a server's key pair with the use of the $\textbf{Eval}^{<}$ algorithm, each evaluation of a timestamp and stream id pair is performed independently from another. Thus, one can substantially decrease the latency of queries by parallelizing the execution of the $\textbf{Eval}^{<}$ algorithm across streams and timestamps. This optimization leads to a constant speedup proportional to the number of processors available.

5.6.2 Batching

A time-series application often issues multiple statistical queries simultaneously. For example, a health service providing a dashboard within an app displaying a client's average heart rate over different periods of time generates many request for averages over time windows for one view of the dashboard. We can optimize I/O operations at the server and reduce the number of network round-trips by batching queries into a single request.

Thus, when a server performs its linear scan of the database, every time it requests a block of data containing a digest, it can multiply every evaluated key from the batch with its corresponding digest element. This reduces the number of I/O requests to the database by a factor proportional to the number of requests within a batch. Furthermore due to all requests in the batch being sent at once, and all results being returned together by the server, we reduce the number of round trips needed to send all keys from the number of queries in a single batch to one per batch.

Chapter 6

Implementation

We implement our two client and server components in a 2-server setting in ~660 lines of C++ code excluding benchmarking infrastructure and FSS library. We rely on Splinter's [27] FSS implementation [28] to build the client and server, which can perform, `SUM`, `COUNT`, sum of squares, `AVG` and `VAR` of a given continuous time range within a specific stream. Furthermore, we exclude any encryption from our implementation thus only operating on plaintext data.

For our key-value store in our database servers, we use LevelDB [22], storing the digest vector as the value with its corresponding stream ID and timestamp concatenated to form the key. Since we access every record in the databases, we are able to reduce the number of database accesses by storing more than one digest into a single entry of our key-value store. We allow for a predefined block size, which sets the number of digests stored in a single database entry, in this case, the key corresponds to the timestamp and stream of the first entry of the block.

For client-server communication, we use gRPC [21], serializing the batched key pairs and other metadata such as the input domain size of the $\mathbf{Eval}^{<}$ algorithm.

6.1 Batching & Parallelism

We parallelize the evaluation of our distributed interval function $\mathbf{Eval}^{<}$ on each database server across streams. Thus, for every stream, a separate thread is created to evaluate its stream ID and timestamps. This approach leads to a constant speedup of server performance roughly proportional to the number of CPUs available. Furthermore, we take the trivial step of parallelizing queries to our two servers, sending function shares to both simultaneously.

We implement batch queries with various sizes depending on the application, generating several key pairs before sending them to the database servers.

Chapter 7

Evaluation

In this chapter, we evaluate the scalability and feasibility of our system. We first provide a description of the experimental setup and methodology. Then, we discuss the microbenchmarks of the function secret sharing library to understand its impact on the system. We then end with the performance of our overall system.

7.1 Experimental Setup

For our evaluation, we use different hardware for the client and servers. For the client, we use an early 2015 MacBook Pro equipped with a 2.7 GHz Intel Core I5-5257U CPU and 16 GB RAM located in Zürich, Switzerland. For the database servers, we use two AWS `c5.xlarge` instances located in Frankfurt, Germany. Each of these instances runs with 4 Intel Xeon Platinum 8000 series vCPU, a total of 8 GB RAM and up to 10 Gbit/s network bandwidth [20]. We measure an average of 9ms RTT between the client and both servers.

In all benchmarks, we measure latency, which is the time the system takes to return a result given a computation. In the case of the microbenchmarks, we measure the time to evaluate a share of an FSS scheme locally on the MacBook Pro. In the end-to-end benchmarks, we measure the time elapsed between dispatching a certain query to the client component and receiving its result.

We evaluate and plot every benchmark $10\times$ where the batch size is set to 1. For batches of size 10, we measure once and divide the resulting time by 10 to plot the average latency of a query within the batch.

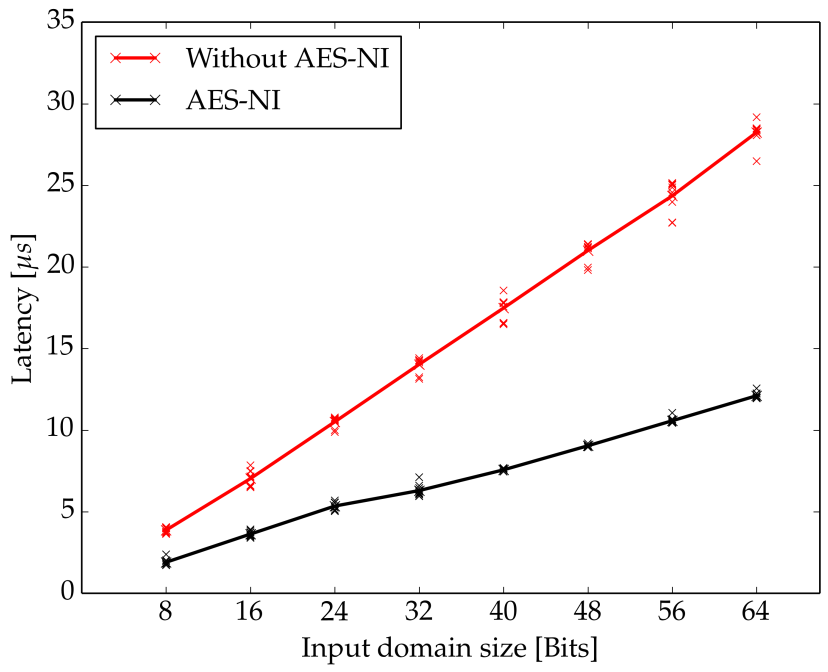

Figure 7.1: Latency of a a single FSS share evaluation with and without the use of the AES-NI hardware instruction. The x-axis represents the number of bits a share must evaluate.

7.2 Microbenchmarks

Our system uses function secret sharing constructions for its distributed interval functions as a core building block to hide query access patterns. The database servers have to perform a linear scan to evaluate the FSS shares on every timestamp and stream id pair and multiply it with the desired digest value. The greatest bottleneck of these operations is executing the core $\mathbf{Eval}^{<}$ algorithm (see subsection 5.3.2), making it crucial for this algorithm as well as $\mathbf{Gen}^{<}$ to be implemented efficiently and understand its performance impact. Thus in this section, we evaluate the latency of our pair of core FSS algorithms $(\mathbf{Gen}^{<}, \mathbf{Eval}^{<})$.

7.2.1 Eval Latency

In Fig. 7.1, we show the computation latency of the $\mathbf{Eval}^{<}$ algorithm in the Splinter FSS library for varying input domain sizes in bits. The cost of each evaluation increases linearly with the domain size. In our system, the input domain size is dependent on the required number of streams and timestamps. That is, the greater the input domain to the $\mathbf{Eval}^{<}$ algorithm, the larger the database can be. In a standard configuration with 48 bits, i.e., 32-bit stream id and 16-bit timestamp, the evaluation cost is 9.05µs with the

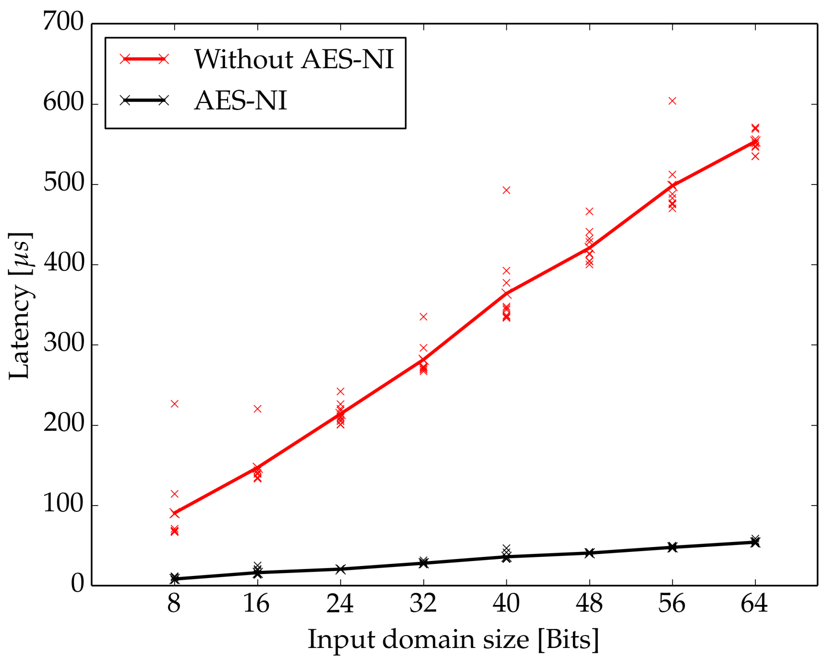

Figure 7.2: Latency of the FSS share generation algorithm **Gen**$^<$ with and without the use of the AES-NI hardware instruction. The x-axis represents the number of bits of the input domain of the distributed comparison function.

AES-NI hardware support and 21µs without.

Taking advantage of the AES-NI hardware instruction we achieve on average 2.2× faster evaluation compared to a software AES implementation, a key contributing factor to the practical scalability of our system discussed in the following section. Thus on average we are able to evaluate an **Eval**$^<$ algorithm with a 48-bit input size over 110′000× per second.

7.2.2 Gen Latency

In Fig. 7.2, we plot the computation latency of Splinter's **Gen**$^<$ algorithm for varying input domain sizes in bits. Cost for both AES hardware and software implementations increase linearly. In the standard configuration (32-bit stream id and 16-bit timestamp), the evaluation cost is 40.6µs with AES-NI hardware support and 420.72µs without.

On average, support for the AES-NI hardware instruction decreases the latency of the **Gen**$^<$ algorithm by 10.2×. Here we see the performance benefit the AES-NI hardware instruction gives an implementation of an FSS DCF, greatly reducing the latency compared to a software implementation.

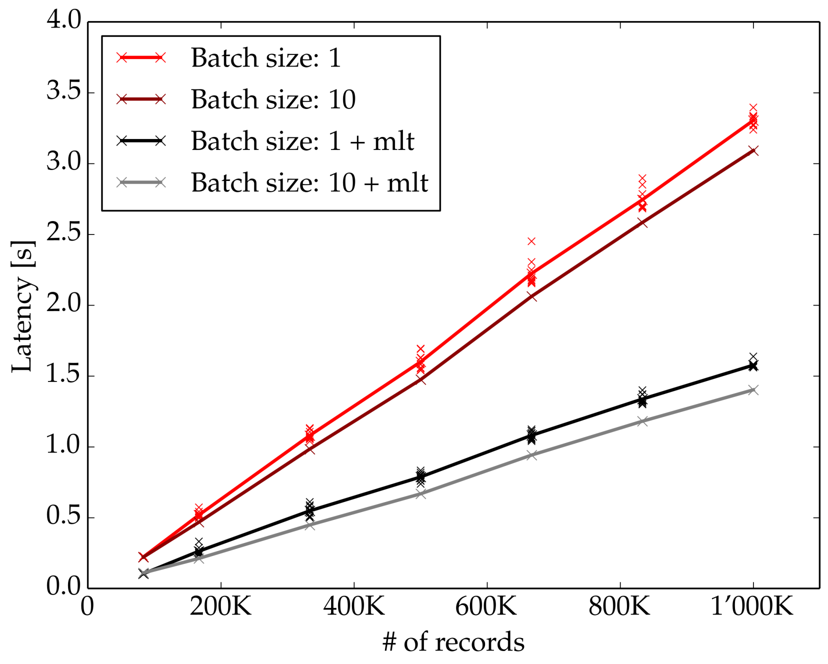

Figure 7.3: Latency for single `COUNT` queries with batch sizes set to 1 and 10 as well as with and without the use of multithreading (mlt).

7.3 End-to-End Benchmarks

In this section, we discuss the end-to-end performance of our full system implementation in a realistic deployment. To understand the optimizations discussed in section 5.6, we consider different system setups. In the baseline configuration, no optimizations are enabled such that the queries are executed on a single thread without batching. The second configuration enables batching of queries where the size of batches is set to 10 but where queries are executed on a single thread. The third configuration enables multithreaded execution of the $\textbf{Eval}^{<}$ algorithm but sends every query individually without batching them. The final configuration consists of both optimizations enabled where queries are batched where the size of batches is set to 10 and function shares are evaluated in parallel when executing the $\textbf{Eval}^{<}$ algorithm.

7.3.1 Single Query

Latency

In Fig. 7.3, we show the impact of the database size on the query latency for a single `COUNT` query for the different setups. Since for every query a linear scan of the database and multiplication with the chosen digest must be performed, regardless of the digest element chosen in the query (count, sum, sum of squares), the actual digest element chosen has little impact on the measured query latency.

For all setups, each database splits its records across 8 streams and sets the server block size to 8. Our standard batch size is set to 10 since as we will see in subsection 7.3.2, further increasing batch sizes yields diminishing returns. We observe a linear increase in latency from 0.1s for our multithreaded implementations with batch sizes 1 and 10, up to 1.4s for our fastest multithreaded implementation with a batch size of 10.

We note that our implementations with multithreading enabled offer the greatest latency improvement, where on average a 2.1× speedup compared to their single-threaded counterpart. Our batching optimization however yields an average 1.107× speedup. This marginal improvement is because many records can be cached in the LevelDB database servers' memory, leading to quick accesses to individual records. This combined with the relatively low latency of 9ms between our client and server, contributes to the observed minimal speedup.

For single queries on databases of small to medium sizes, we observe a practical latency, implying that our system can be used in real-world scenarios where a client's requests are few and far between. Furthermore, we could easily further decrease latency by evaluating requests on servers equipped with a higher CPU count, since we evaluate our system with a server containing 4 CPUs and typical modern servers can easily exceed 32 CPUs [20].

#Bits	Size [Byte]
8	0.8K
16	1.7K
32	3.5K
64	7.1K

Table 7.1: Size of a single key share depending on the input domain size of the $\mathbf{Eval}^{<}$ algorithm.

Bandwidth

We continue by discussing the bandwidth our system uses when executing a query. Table. 7.1 shows the size of a key share given a certain input domain size, which is generated by the client component. For every request, two keys are sent as well as metadata about the key shares such as the number of input bits to the FSS function and the number of total function shares. The metadata for one server totals to 1.06kB. As an example, in a single request in our evaluation from Fig. 7.3 on a database with 1M records, we send 11.28kB to the servers, which includes four key shares for the two servers

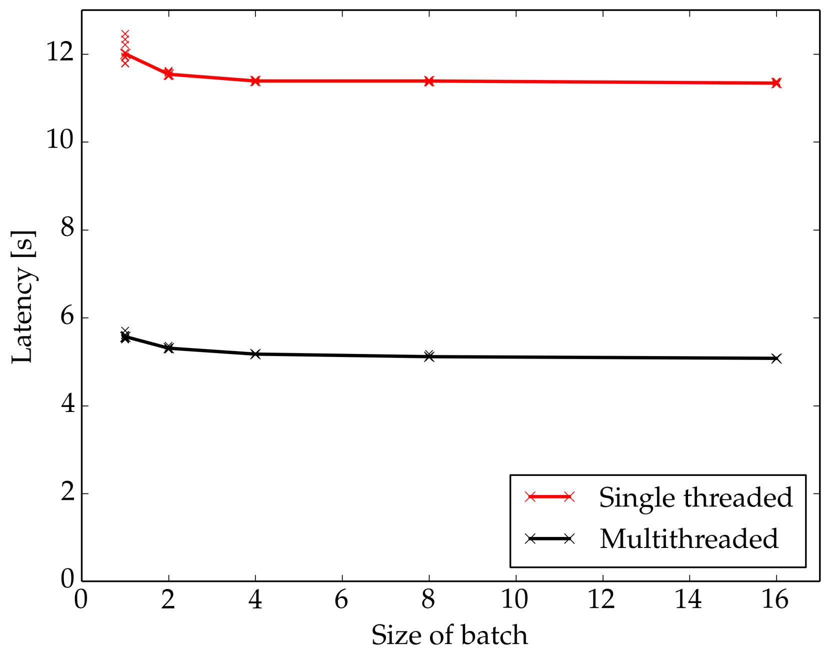

Figure 7.4: Latency of a single `COUNT` query for different batch sizes using the single threaded and multithreaded implementations on a database with 3'200'000 records divided into 4 streams. Size of database blocks are set to 8.

with an input domain size of 21 bits (17-bit timestamp and 4-bit stream id) and two copies of the metadata. The response received by both servers totals 16 bytes. In total, excluding any overhead introduced by gRPC, bandwidth equals 7162 Bps in this specific case.

Due to the small bandwidth needed to send FSS key shares, our system can scale to a high volume of single queries where bandwidth is limited and where database sizes remain in the small to medium range.

7.3.2 Batch Sizes

In Fig. 7.4 we compare latency of `COUNT` queries with different batch sizes on a database of 3.2M records evenly divided among 4 streams. We measure the latency of batches with sizes set to 1, 2, 4, 8 and 16.

Increasing batch sizes to from 1 to 4 offers a decrease in latency from 5.57s to 5.173s, or a 7.7% decrease. We observe a minimal increase in performance with batch sizes greater than 4, where we measure a decrease in latency from 5.173s to 5.077s for a batch size of 16, implying 1.9% decrease. This marginal gain in performance, again, is due to the relatively low latency of 9ms between client and server and the large cache of LevelDB. Since the latency cost of a round trip is minimal, the advantage of batching a large number of requests together, which decreases the number of round trips, is small. Thus one can keep batch sizes relatively low without affecting query latency. Once very large database sizes are reached, however, batching queries will show greater performance gains since only then will the database cache no longer be able to store large portions of the total records and thus will need to perform many more I/O operations.

7.3.3 Health App Dashboard

To understand the real-world performance of our system, we consider a query workload from a health dashboard application. This dashboard displays a summary plot of a client's heart rate at a chosen granularity. The application creates one `AVG` request per data point in the plot, which the client component of our system compiles to individual `COUNT` and `SUM` requests leading to 2× as many requests to the database servers as application queries.

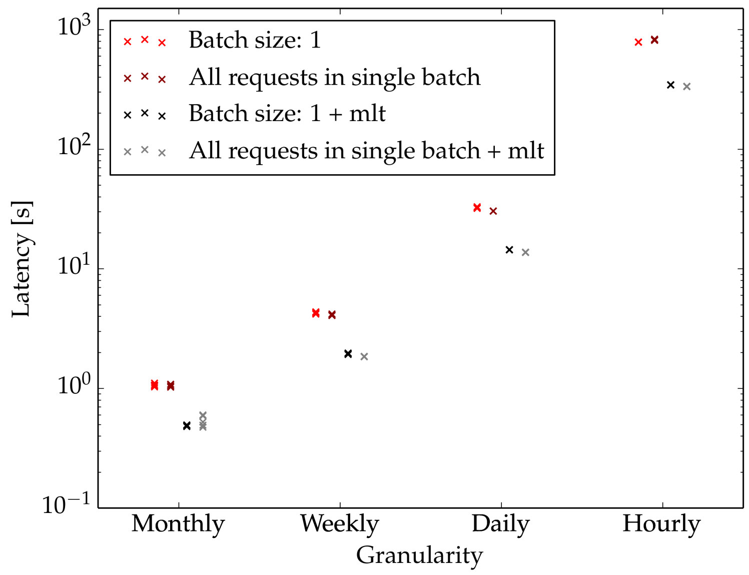

Figure 7.5: Latency in log-scale to request data for a windowed average view from a health service storing one month of data where one measurement is taken every minute for 4 users (172'800 records). The x-axis shows the granularity of the requested data; one hour (1440 requests database requests), one day (60 requests), one week (8 requests), one month (2 requests).

In Fig 7.5 we compare latencies of the application's requests for one user's data at different granularity. We run every benchmark 5× and plot the results. Here, our implementations with batched queries send all their requests in one single batch. Our evaluation's database consists of a month of heart rate data where a measurement was taken at every minute (43’200 records per user) for 4 users, totaling 172’800 records. For a monthly granularity where a user sees an average of all his stored data, the application generates 1 `AVG` query (2 server requests). At weekly granularity, 4 queries are generated (8 requests), daily granularity consists of 30 queries (60 requests) and finally 720 queries (1440 requests) are generated for an average of a month’s data at hourly granularity.

For typical workloads, where a user requests his month’s data at weekly or daily granularity, we observe an acceptable latency of 1.8s and 13.7s respectively for our implementation with both optimizations enabled. For our evaluation at an hourly granularity, we observe a latency of 344s. This high latency is due to each server needing to evaluate 2880 separate function shares, each of which consists of a linear scan of the database. We do note, however, that this type of request is an extreme case, where plotting this amount of data, which in this case amounts to the client requesting all of his data, is not typically seen in this setting. Furthermore, this high latency could still be reasonable to a client who requires the high level of privacy that our system guarantees.

For larger database sizes which, for example, accommodate a larger user base, or applications where both privacy and latency are important, further measures would need to be taken to further decrease latency to a reasonable level.

Chapter 8

Conclusion

In this thesis, we provide the design and implementation of a time-series storage system that hides a client's query access patterns from the storage providers as long as two of them do not collude. Our system enables efficient pattern-hiding time-series queries on data streams by employing function secret sharing techniques. While previous work explored how FSS can be used in other data storage systems [11, 27], our system applies FSS to the setting of time-series data. To scale to large volumes of data common in time-series workloads, our system introduces stream abstractions to allow for efficient query execution with FSS evaluations. Furthermore, our system introduces a module that translates times-series queries into query shares which are distributed to replicated servers. We achieved this by generating distributed comparison functions with the help of FSS, which select the requested digest elements from the database without leaking the function's selection to the servers.

To demonstrate our system's performance we implement a baseline which we extend with two further optimizations and evaluate their performance. The evaluation shows, that for small- to medium-sized time-series databases we achieve acceptable query latency. This, combined with our system's strong privacy guarantee, enables it to be used in certain real-world scenarios, where hiding query access patterns is paramount.

8.1 Future Work

Different aspects of our design are open to future work. We design our system with compatibility with TimeCrypt in mind but have not implemented nor evaluated it. Implementing our system on top of TimeCrypt would strengthen the privacy guarantees of the system by introducing end-to-end encryption to database records. Furthermore, our system could be extended to support further query types as well as support writes to the databases.

This would introduce other challenges such as guaranteeing consistency of the database replicated over several servers, which would need to be addressed.

During this thesis, another line of work applying FSS to time-series data appeared [12]. Although we use a different system design and threat model, it would be worth exploring the performance differences as well as comparing the techniques used.

Bibliography

[1] Michael P Andersen and David E Culler. BTrDB: Optimizing storage system design for timeseries processing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 39–52, 2016.

[2] Davis Blalock, Samuel Madden, and John Guttag. Sprintz: Time series compression for the internet of things. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(3):1–23, 2018.

[3] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, and Michele Orrù. Homomorphic secret sharing: optimizations and applications. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2105–2122, 2017.

[4] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 337–367. Springer, 2015.

[5] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1292–1303, 2016.

[6] Elette Boyle, Niv Gilboa, Yuval Ishai, Huijia Lin, and Stefano Tessaro. Foundations of homomorphic secret sharing. *Cryptology ePrint Archive*, 2017.

[7] Lukas Burkhalter, Anwar Hithnawi, Alexander Viand, Hossein Shafagh, and Sylvia Ratnasamy. TimeCrypt: Encrypted data stream processing at scale with cryptographic access control. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 835–850, 2020.

[8] Lukas Burkhalter, Nicolas Küchler, Alexander Viand, Hossein Shafagh, and Anwar Hithnawi. Zeph: Cryptographic enforcement of end-to-end data privacy. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 387–404, 2021.

[9] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In *EUROCRYPT*, 2020.

[10] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the Scalability Bottleneck of Oblivious Storage. In *ACM SOSP*, 2021.

[11] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. Dory: An encrypted search system with distributed trust. Cryptology ePrint Archive, Report 2020/1280, 2020. https://ia.cr/2020/1280.

[12] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. Waldo: A private time-series database from function secret sharing. *Cryptology ePrint Archive*, 2021.

[13] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and Private Media Consumption with Popcorn. In *USENIX NSDI*, 2016.

[14] InfluxDB. https://get.influxdata.com/rs/972-GDU-533/images/Customer_Case_Study_Loft_Orbital.pdf.

[15] InfluxDB. https://www.influxdata.com/time-series-database/.

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

[17] InfluxDB. https://docs.influxdata.com/influxdb/v2.2/write-data/best-practices/schema-design/.

[18] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proc. VLDB Endow.*, 8(12):1816–1827, aug 2015.

[19] Jeffrey Rott. Intel advanced encryption standard instructions (AES-NI). *Technical Report, Technical Report, Intel*, 2010.

[20] Amazon Web Service. AWS c5 instance type. https://aws.amazon.com/ec2/instance-types/c5/.

[21] Google Open Source. gRPC. https://grpc.io.

[22] Google Open Source. Leveldb. https://github.com/google/leveldb.

[23] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM CCS*, 2013.

[24] TimescaleDB. https://docs.timescale.com/timescaledb/latest/overview/what-is-time-series-data/#characteristics.

[25] TimescaleDB. https://www.timescale.com/.

[26] TimescaleDB. https://docs.timescale.com/timescaledb/latest/overview/data-model-flexibility/#data-model.

[27] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical Private Queries on Public Data. In *USENIX NSDI*, 2017.

[28] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Function secret sharing (FSS) library, 2022. https://github.com/frankw2/libfss.