This is the Pre-Published Version.
The following publication H. Ouyang, Y. Zheng, S. Wang and Z. Hua, "OblivTime: Oblivious and Efficient Interval Skyline Query Processing Over Encrypted Time-Series Data," in IEEE Transactions on Services Computing, vol. 18, no. 3, pp. 1602-1617, May-June 2025 is available at https://doi.org/10.1109/TSC.2025.3553698.

OblivTime: Oblivious and Efficient Interval Skyline Query Processing over Encrypted Time-Series Data

Huajie Ouyang, Yifeng Zheng, Songlei Wang, and Zhongyun Hua

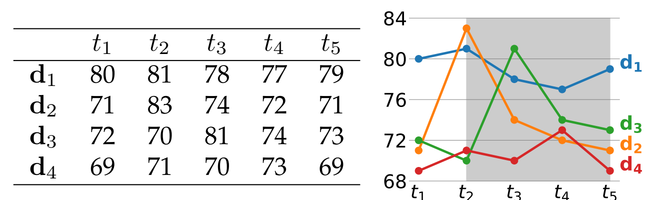
Abstract—Time-series data is prevalent in many applications like smart homes, smart grids, and healthcare. And it is now increasingly common to store and query time-series data in the cloud. Despite the benefits, data privacy concerns in such outsourced services are pressing, making it imperative to embed privacy assurance mechanisms from the outset. Most existing related works have been focused on querying for different types of aggregate statistics. In this paper, we instead focus on the secure support for advanced interval skyline queries, which allow to identify time series that are not dominated by any other time series within a query time interval. This is valuable for time-series data analytics in applications like remote health monitoring (e.g., identifying patients with high heart rates in a certain week). We present OblivTime, a new system framework for oblivious and efficient interval skyline query processing over encrypted time-series data. OblivTime is built from a synergy of time-series data analytics, lightweight cryptography, and GPU parallel computing, achieving stronger security guarantees and lower online query latency over the state-of-the-art prior work. Extensive experiments demonstrate that OblivTime can achieve up to $666\times$ speedup in online query latency over the state-of-the-art prior work.

Index Terms—Time-series analytics, privacy preservation, query processing, cloud computing

1 Introduction

Many practical applications involve continuously generated time-series data, such as remote health monitoring [1], [2], smart homes [3], smart grids [4], and more. To store and query the time-series data, a common trend nowadays is to leverage cloud databases [5]–[7], due to the prominent benefits like cost-effectiveness and scalability. Despite the convenience, such outsourcing also raises critical privacy concerns, as the time-series data may contain sensitive information (e.g., time-series data generated from medical applications or smart homes) and it is not unusual to see the occurrence of cloud data breach incidents in practice [8], [9]. Hence, it is important to have built-in privacy assurance mechanisms for the emerging trend of storing and querying time-series data stored in the cloud.

Among others, interval skyline query [10] is highly useful for time-series data analytics, which identifies time series that are not dominated by any other time series within a query time interval in various applications such as health monitoring and smart grids [10], [11]. Given a query time interval $[t_x : t_y]$ and two time series $\mathbf{p}_1, \mathbf{p}_2$, $\mathbf{p}_1$ is said



	t_1	t_2	t_3	t_4	t_5
$\mathbf{d}_1$	80	81	78	77	79
$\mathbf{d}_2$	71	83	74	72	71
$\mathbf{d}_3$	72	70	81	74	73
$\mathbf{d}_4$	69	71	70	73	69

Fig. 1. An example of time-series data for interval skyline query.

to dominate $\mathbf{p}_2$ if the values of $\mathbf{p}_1$ within the interval are greater than that of $\mathbf{p}_2$ at least one timestamp and no lesser at other timestamps. For demonstration, we describe a concrete example to show the usefulness of interval skyline query processing on time-series data. Consider that a hospital employs medical devices for a remote health monitoring service. These devices continuously collect sensitive biometric data of patients, e.g., heart rate data, and transmit it to the cloud in real time. This enables doctors to remotely monitor patients' health conditions and identify individuals with high heart rates over a certain period of time based on interval skyline queries. For illustration, Fig. 1 shows synthesized time-series data for the heart rates (measured by beats per minute) of four patients in a monitoring period from timestamp t_1 to timestamp t_5. Suppose a doctor wishes to gather heart rate data of patients exhibiting high heart rates within a time interval $[t_2 : t_5]$ for further analytics. It is obvious that the time series $\mathbf{d}_1$, $\mathbf{d}_2$ and $\mathbf{d}_3$ should be returned, as each of them is not dominated by any other time series. Indeed, $\mathbf{d}_1$ has the highest average heart rate, while $\mathbf{d}_2$ and $\mathbf{d}_3$ exhibit spikes at t_2 and t_3 respectively. The time series $\mathbf{d}_4$ is not retrieved as $\mathbf{d}_1$ has higher value than $\mathbf{d}_4$ at every timestamp within the query interval (i.e., $\mathbf{d}_4$ is dominated by $\mathbf{d}_1$).

In such cloud-based time-series data service, it is imper-

- *Huajie Ouyang is with the School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, Guangdong 518055, China (e-mail: huajie.ouyang@outlook.com).*
- *Yifeng Zheng is with the Department of Electrical and Electronic Engineering, The Hong Kong Polytechnic University, Hong Kong, China (e-mail: yifeng.zheng@polyu.edu.hk).*
- *Zhongyun Hua are with the School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, Guangdong 518055, China (e-mail: huazhongyun@hit.edu.cn).*
- *Songlei Wang is with the National Engineering Laboratory for Big Data System Computing Technology, Shenzhen University, Shenzhen 518055, China (e-mail: songlei.wang@szu.edu.cn).*
- *Corresponding author: Yifeng Zheng.*

© 2025 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

ative to preserve the privacy of the time-series database, the query time interval, as well as the query result. Meanwhile, it is also necessary to hide data patterns that may incur indirect leakages [12], [13]. These data patterns include the dominance relationships among the time series in the database, the number of time series dominated by each skyline time series, and the search and access patterns. Note that the search pattern indicates whether a new query time interval has been issued previously, while the access pattern reveals which time series in the database are identified as the query result. In the literature, privacy-preserving skyline query processing has attracted increasing attention in recent years [11]–[17]. However, most existing works [12]–[17] consider the execution of skyline queries on static databases of multi-dimensional data records (with a fixed number of dimensions). Also, the query in these works is in the form of a vector with the same number of dimensions as database records. These works cannot be directly adapted to securely handle interval skyline queries on time-series data, where the database is dynamic in nature and continuously grows, and the query dynamically specifies a time interval. To our best knowledge, only the prior work in [11] has studied interval skyline query processing on time-series data with privacy preservation, which represents the state-of-the-art.

Despite being a valuable seminal work, the state-of-the-art scheme in [11] is not yet fully satisfactory due to the following downsides. Firstly, it directly exposes the query time interval to the cloud, which could be sensitive information in time-series data analytics [18]. Secondly, it relies on an insecure sorting protocol which leaks the order information of the input secret values and thus entails the risk of data reconstruction [19]. Thirdly, it suffers from high-performance overheads in query latency. For example, even when handling a database with 1,000 time series and a query time interval of 100 timestamps, the query latency in [11] is over 10,000 seconds.

In light of the above, in this paper, we propose OblivTime, a new system framework for oblivious and efficient interval skyline query processing over encrypted time-series data. OblivTime delicately bridges insights on time-series data analytics and lightweight cryptography (such as replicated secret sharing [20], function secret sharing (FSS) [21], and secure shuffle [22]). At a high level, time-series data are continuously collected and stored in the cloud in encrypted form in OblivTime. The cloud can then receive encrypted query time intervals and obliviously identify the skyline time series in the ciphertext domain. Throughout the whole querying process, OblivTime not only ensures data confidentiality but also hides the aforementioned data patterns. We start with devising a protocol of oblivious database projection, which allows the cloud to obliviously produce a sub-database of encrypted time-series data within a secret query interval. Then, we devise a protocol to allow oblivious interval skyline identification, which processes the above sub-database and produces encrypted identifiers of the target skyline time series. Lastly, we construct a protocol to allow oblivious retrieval of the encrypted skyline time series given their encrypted identifiers. We also show how to rely on GPU-based protocol instantiations to boost the performance. We highlight our main contributions as below:

1) We present a new system framework OblivTime for oblivious and efficient interval skyline query processing over encrypted time-series data, which achieves stronger security and better query latency performance than the state-of-the-art [11].
2) We devise a suite of secure protocols for oblivious interval skyline query processing, including oblivious database projection, oblivious interval skyline identification, and oblivious interval skyline retrieval.
3) We present formal security analysis and introduce GPU-based practical performance optimization mechanisms. We conduct extensive evaluations and the results show that while providing stronger security guarantees, OblivTime also achieves up to 666× better query latency than the state-of-the-art [11].

2 Related Work

2.1 Privacy-Aware Skyline Query Processing

Bothe *et al*. [14] propose the first scheme for executing skyline queries over encrypted data. Since then, a number of research endeavors have been devoted to privacy-aware skyline query processing. Most existing works [12], [13], [15]–[17], [23] have been focused on privacy-preserving dynamic skyline query processing. They aim to allow the querier to dynamically specify a vector as a query and securely find skyline data records where the dominance relationships are defined with respect to the query vector. In other works [11], [24], [25], other different types of skyline queries are considered. Specifically, the works in [24], [25] focus on user-defined skyline queries, where users can specify subsets of dimensions for querying, assign preferences for maximum/minimum values in each selected dimension, or define range constraints.

The work in [11] considers interval skyline query over time-series data, which is most related to this paper and represents the state-of-the-art. Their design uses symmetric homomorphic encryption and also works under a distributed trust model. Despite the great value, this state-of-the-art work [11] suffers from security and efficiency downsides. Specifically, the private query time interval in [11] is directly exposed to the cloud for easily obtaining encrypted time-series data within the time interval. This also leaks the search pattern. They then propose algorithms to identify skyline time series in the ciphertext domain, where they need to rely on secure sorting of the encrypted time-series data. Unfortunately, the so-called secure sorting protocol in their design has limited security, as it leaks the secret order information of the inputs to one cloud server. Such leakage might entail risks of data reconstruction [19]. Furthermore, their design suffers from high query latency, which limits the practical usability as well. In comparison, OblivTime does not leak the sensitive query time interval, and achieves much better query latency performance through custom designs and practical GPU-based optimization.

2.2 Privacy-Aware Query Processing on Time-Series Data

There has been a line of work devoted to advancing privacy-preserving query processing on time-series data. The works in [26], [27] focus on aggregation by time over a data stream

Algorithm 1 Interval Skyline Query Processing in Plaintext

Input: A time-series database $\mathbf{M}$ in matrix form, and a query interval $[t_1 : t_2]$.

Output: The skyline $\mathcal{R}$ over the query interval $[t_1 : t_2]$.

1: Let N be the number of time series, $[t_a : t_c]$ be the most recent W timestamps recorded in the database, $m = 1$.
2: Initialize a matrix $\mathbf{P} = 0^{N \times I}$, where I is the length of query interval and $I = t_2 - t_1 + 1$.
3: **for** $j = 1$ to W **do**
4:   **if** $t_1 \le t_a + j - 1 \le t_2$ **then**
5:     $\mathbf{P}[:][m] = \mathbf{M}[:][j]$.
6:     $m = m + 1$.
7:   **end if**
8: **end for**
9: **for** $i = 1$ to N **do**
10:   $\mathbf{s}[i] = \sum_{j=1}^{I} \mathbf{P}[i][j]$.
11: **end for**
12: Initialize an empty set $\mathcal{I}$.
13: **while** $\mathbf{P}$ is not empty **do**
14:   The time series $\mathbf{P}[id_*]$ with maximum sum $\mathbf{s}[id_*]$ in $\mathbf{P}$ is selected, and denoted as $\mathbf{p}_*$.
15:   Delete $\mathbf{p}_*$ and the time series dominated by it from $\mathbf{P}$.
16:   Add id_* to $\mathcal{I}$.
17: **end while**
18: $\mathcal{R} = \{\mathbf{M}[id_*] \mid id_* \in \mathcal{I}\}$.
19: **return** $\mathcal{R}$.

(e.g., computing the average heart rate over a week) while supporting cryptographically enforced access control. The work in [18] focuses on supporting additive aggregates (such as sum, count, and variance) based on multiple predicates, and arbitrary aggregates (e.g., max, min, and top-k) over a time interval. The work in [28] introduces a window operator to query over encrypted time series for oblivious windows-based aggregation. Despite being valuable, all these works focus on aggregate queries to obtain aggregate statistics, which are quite different from the interval skyline queries we target in this paper.

3 Preliminaries

3.1 Interval Skyline Query on Time-Series Data

A time series $\mathbf{d}$ is a sequence of values ordered with respect to timestamps [29]. Without loss of generality, we assume the timestamps are consecutive positive integers starting from 1. We denote the value of $\mathbf{d}$ at timestamp t by $\mathbf{d}[t]$, so a time series $\mathbf{d}$ can be written as a sequence of values $\mathbf{d}[1], \mathbf{d}[2], \cdots$. A time interval $[t_1 : t_2]$ where $(t_1 \le t_2)$ specifies a set of timestamps $\{t \mid t_1 \le t \le t_2\}$. The subsequence of the time series $\mathbf{d}$ within the interval $[t_1 : t_2]$ is denoted as $\mathbf{d}[t_1 : t_2] = (\mathbf{d}[t_1], \mathbf{d}[t_1+1], \ldots, \mathbf{d}[t_2])$. Time series $\mathbf{d}_1$ is said to dominate $\mathbf{d}_2$ within the interval $[t_1 : t_2]$ if $\forall t \in [t_1 : t_2], \mathbf{d}_1[t] \ge \mathbf{d}_2[t]$, and $\exists t_0 \in [t_1 : t_2], \mathbf{d}_1[t_0] > \mathbf{d}_2[t_0]$ [10]. Let $\mathcal{D} = \{\mathbf{d}_1, \ldots, \mathbf{d}_N\}$ be a set of N time series and contains data across the most recent W timestamps, i.e., $[t_a : t_c]$ where t_c refers to the most recent timestamp and $t_a = t_c - W + 1$. Given a set of time series $\mathcal{D}$ and a query interval $[t_1 : t_2] \subseteq [t_a : t_c]$, $\mathbf{d}_i \in \mathcal{D}$ is a skyline time series if, within the query interval, there does not exist any other time series in $\mathcal{D}$ that dominates it [10]. Therefore, an interval skyline query with interval $[t_1 : t_2]$ over the time series database $\mathcal{D}$ yields a subset of $\mathcal{D}$, denoted as $\mathcal{R}$, which contains only the skyline time series as defined above [10].

In Algorithm 1, we illustrate an iterative algorithm for interval skyline query processing in the plaintext domain, which will serve as the basis for our secure design. Given a time-series database $\mathcal{D}$ with N time series across the most recent timestamps $[t_a : t_c]$, the input $\mathbf{M}$ is the time-series values in matrix form, where $\mathbf{M}[i][j] = \mathbf{d}_i[t_a + j - 1]$, $i \in [1 : N], j \in [1 : W]$. For notational convenience, we use $\mathbf{M}[i]$ to represent the i-th row (corresponding to the i-th time series), and $\mathbf{M}[:][j]$ to refer to the j-th column (which corresponds to all the time-series values at timestamp $t_a + j - 1$). The algorithm works as follows: Firstly, it projects the matrix $\mathbf{M}$ to matrix $\mathbf{P}$ so that $\mathbf{P}$ contains time series with values only over the query interval $[t_1 : t_2]$. And the sum $\mathbf{s}[i]$ over the values of each row $\mathbf{P}[i]$ in $\mathbf{P}$ is calculated. Then, the algorithm goes through multiple rounds, where each round produces an intermediate skyline times series from the current $\mathbf{P}$ and updates $\mathbf{P}$. Specifically, in each round, the time series $\mathbf{P}[id_*]$ ($id_* \in [1 : N]$) with maximum sum $\mathbf{s}[id_*]$, denoted by $\mathbf{p}_*$, is selected from the current $\mathbf{P}$ to be the intermediate skyline time series. The original time series $\mathbf{d}_*$ in $\mathbf{M}$ corresponding to $\mathbf{p}_*$ is a target skyline time series, and its ID id_* is added to $\mathcal{I}$. Then, $\mathbf{P}$ is updated by removing $\mathbf{p}_*$ and other time series dominated by $\mathbf{p}_*$. After that, the algorithm is ready to go through another round to find the next skyline time series. When all the time series are deleted from $\mathbf{P}$, the algorithm returns $\mathcal{R}$ retrieved with IDs in $\mathcal{I}$ as the interval skyline query result on $\mathcal{D}$ for the query interval $[t_1 : t_2]$.

3.2 Replicated Secret Sharing

Replicated secret sharing (RSS) [20] is a secret sharing technique that supports secure computation among three parties C_1, C_2, C_3. Given a secret value $x \in \mathbb{Z}_{2^l}$, RSS first split x into three shares $\langle x \rangle_1, \langle x \rangle_2, \langle x \rangle_3$ where $\langle x \rangle_1 + \langle x \rangle_2 + \langle x \rangle_3 = x$ and each party gets a pair of the shares, i.e., C_1 gets $(\langle x \rangle_1, \langle x \rangle_2)$, C_2 gets $(\langle x \rangle_2, \langle x \rangle_3)$, and C_3 gets $(\langle x \rangle_3, \langle x \rangle_1)$. For ease of presentation, we use C_{i+1} (or $\langle x \rangle_{i+1}$) to refer to the next party (or secret share) of C_i (or $\langle x \rangle_i$) and C_{i-1} (or $\langle x \rangle_{i-1}$) to refer to the prior, all with wrap around. We denote such secret sharing of x as $[\![x]\!]$, and the secret shares for C_i, i.e., $(\langle x \rangle_i, \langle x \rangle_{i+1})$, as $[\![x]\!]_i$. Given a public value x, the parties can locally generate RSS shares of x. We define that $\langle x \rangle_1 = x$, $\langle x \rangle_2 = 0$, $\langle x \rangle_3 = 0$, so that each party $C_i, i \in \{1, 2, 3\}$ can obtain its RSS shares $[\![x]\!]_i = (\langle x \rangle_i, \langle x \rangle_{i+1})$ without interaction with other parties. We denote such process as $[\![x]\!] = \mathsf{localShare}(x)$.

RSS can securely support basic operations including addition, subtraction, and multiplication. Specifically, for addition and subtraction, each C_i can locally calculate $\langle z \rangle_i = \langle x \rangle_i \pm \langle y \rangle_i$ and $\langle z \rangle_{i+1} = \langle x \rangle_{i+1} \pm \langle y \rangle_{i+1}$ when given the RSS shares $\langle x \rangle_i$, $\langle x \rangle_{i+1}$, $\langle y \rangle_i$, $\langle y \rangle_{i+1}$. This can yield the RSS sharing of $z = x \pm y$ among the three parties. Similarly, one can perform addition and subtraction with a publicly known value by first locally sharing the public value using localShare. The resulting shares can then be used for computation with the two RSS sharings. Multiplication between

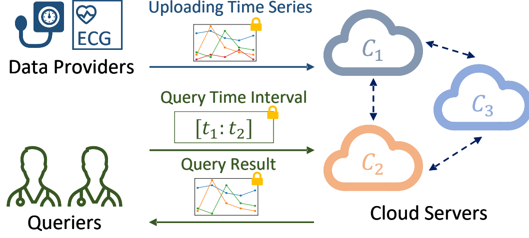


Fig. 2. The architecture of OblivTime.

a publicly known value and an RSS sharing, denoted as $[\![z]\!] = x \times [\![y]\!]$, can be computed by each C_i locally with $\langle z\rangle_i = x \times \langle y\rangle_i$, where $z = x \times y$ and $[\![z]\!]_i = (\langle z\rangle_i, \langle z\rangle_{i+1})$. To compute the multiplication result $[\![z]\!]$ on two RSS sharings $[\![x]\!]$ and $[\![y]\!]$ (i.e., $z = xy$), each C_i first locally calculate $\langle z\rangle_i = \langle x\rangle_i \times \langle y\rangle_i + \langle x\rangle_i \times \langle y\rangle_{i+1} + \langle x\rangle_{i+1} \times \langle y\rangle_i, i \in \{1,2,3\}$. Here the output are 3-out-of-3 additive secret shares. To get z in RSS form, $C_{\{1,2,3\}}$ reshare it where each C_i sends $\langle z'\rangle_i = \langle z\rangle_i + \langle\alpha\rangle_i$ to C_{i-1} where $(\langle\alpha\rangle_1, \langle\alpha\rangle_2, \langle\alpha\rangle_3)$ is a fresh secret sharing of 0 so that $\langle\alpha\rangle_1 + \langle\alpha\rangle_2 + \langle\alpha\rangle_3 = 0$. Then each C_i has its RSS shares $[\![z]\!]_i = (\langle z'\rangle_i, \langle z'\rangle_{i+1})$. Here, the 3-out-of-3 sharing of 0 used in RSS multiplication for masking can be easily generated by C_1, C_2, C_3 locally using a pseudo-random function (PRF) F [20]. In particular, each party C_i generates a key k_i for PRF and sends it to C_{i-1}. Then, the j-th fresh share of 0 for party C_i is $\langle\alpha\rangle_i = F(k_i, j) - F(k_{i+1}, j), i \in \{1,2,3\}$. Such computation can be conducted in advance.

3.3 Function Secret Sharing

Function secret sharing (FSS) [30] can split a function into secret shares. In particular, a 2-party FSS scheme consists of a generator algorithm Gen and an evaluation algorithm Eval. Given a function f, Gen generates two FSS keys k_1, k_2, which are secret shares of the function f. With inputs k_i and x, Eval outputs a share $f_i(x)$, so that $f(x) = f_1(x) + f_2(x)$. We identify two FSS constructions suitable for use in OblivTime, i.e., distributed point functions (DPFs) [31] and distributed interval functions (DIFs) [21], [32]. For a point function $f_{\alpha,\beta}$, we have $f_{\alpha,\beta}(\alpha) = \beta$ and $f_{\alpha,\beta}(\alpha') = 0$ for $\alpha' \neq \alpha$. For an interval function $g_{a,b}$, it evaluates to 1 at input x for $x \in [a : b]$, and to 0 at any other input. Given the security parameter 1^λ, we denote the generator algorithms for DPFs and DIFs as $\mathsf{Gen}_{\mathsf{DPF}}(1^\lambda, \alpha, \beta)$ and $\mathsf{Gen}_{\mathsf{DIF}}(1^\lambda, a, b)$ respectively. And we write $f_i(x) = \mathsf{Eval}_{\mathsf{DPF}}(i, k_i, x)$ and $g_i(x) = \mathsf{Eval}_{\mathsf{DIF}}(i, k_i, x)$ to refer to the evaluation algorithms for DPFs and DIFs, respectively.

4 System Overview

4.1 Architecture

Fig. 2 illustrates the system architecture of OblivTime, which is aimed at supporting oblivious interval skyline query processing over encrypted time-series data stored in the cloud. There are three types of entities in OblivTime: data providers, cloud servers, and queriers. Data providers can be devices (such as ECG sensors, blood pressure monitors, and other medical sensors), which continuously collect time-series data and upload them to the cloud servers. For privacy assurance as well as low overhead, data providers protect the collected time-series data via lightweight cryptography, so the time-series data received by the cloud servers are in encrypted form.

Queriers (e.g., doctors who want to analyze the time-series database of patients' heart rates and find the patients with high heart rate in a certain query time interval.) are interested in issuing interval skyline queries (in the form of query intervals) to the cloud servers for processing over the encrypted time-series database. As the query interval indicates a querier's personal interests and is thus privacy-sensitive, the querier may not be willing to directly send it in cleartext. In OblivTime, a querier instead sends to the cloud servers secure query tokens for processing, so that the cloud servers are oblivious to the underlying secret query intervals. We observe that time-series data has the following key elements: time-series data is append-only and the most recent time-series data is more valuable [10]. So in our system, it is practical to consider that the cloud servers maintain a sliding window that contains only (encrypted) time-series data within the most recent W timestamps for answering interval skyline queries.

It is noted that the cloud servers in OblivTime (denoted as C_1, C_2, C_3 respectively, or as $C_{\{1,2,3\}}$ collectively) are from different trust domains and jointly empower the oblivious interval skyline query service over the collected encrypted time-series data. This means that the cloud servers should be hosted by different and independent cloud service providers in practice. Such a distributed trust model has recently gained increasing popularity for adoption in both academia [18], [33], [34] and industry [35], [36]. OblivTime follows such a trend and newly explores the support for secure time-series data analytics based on oblivious interval skyline query processing.

4.2 Threat Model and Security Guarantees

Similar to prior works on secure skyline query processing under the multi-server model [11]–[13], [17], we consider that the threats in OblivTime primarily come from the cloud serves. Similarly, we assume a static semi-honest and non-colluding adversary model, where each cloud server exactly follows our protocol but may individually try to learn private information from the protocol execution. Also, following the state-of-the-art prior work [11], we assume that the data providers and queriers are trustworthy. We assume that the communication between any two parties is through standard secure TLS channels. Following prior works [11], [18], [37] on privacy-preserving time-series data analytics, we consider that the data providers upload their data at a fixed interval with ordered and public timestamps. With respect to the above threat model and similar to prior works on secure skyline query processing [12], [13], [17], OblivTime aims to offer protection for the following private information against the cloud servers: 1) The contents of the stored time-series data, query intervals, and query results; 2) The number of time series each skyline time series dominates; 3) The search access patterns. Here, the search pattern reveals whether the query intervals under two different secure query tokens are the same, and the access pattern reveals which time series in the database are identified as the interval skyline query result.

4.3 Design Overview

In OblivTime, for low-overhead protection of time-series data, we resort to the lightweight RSS technique to encrypt the time-series values at each timestamp. To upload the value $v_{j,t}$ for the j-th time series at timestamp t, the data provider first encrypts $v_{j,t}$ with RSS and sends $(j, t, [\![v_{j,t}]\!]_i)$ to each cloud server $C_i, i \in \{1,2,3\}$. As aforementioned, the cloud servers maintain a sliding window containing only the encrypted time-series data within the most recent W timestamps to facilitate query responses. Let $[\![\mathbf{M}]\!]$ be the cloud servers' encrypted input matrix containing the encrypted time-series values within the sliding window, where each row in $\mathbf{M}$ corresponds to a time series and each column corresponds to a timestamp. Note that $[\![\mathbf{M}]\!]$ is produced by applying RSS (as introduced above in Section 3.2) to $\mathbf{M}$. Specifically, three shares $\langle\mathbf{M}\rangle_1$, $\langle\mathbf{M}\rangle_2$, and $\langle\mathbf{M}\rangle_3$ are first generated by the data provider such that $\langle\mathbf{M}\rangle_1 + \langle\mathbf{M}\rangle_2 + \langle\mathbf{M}\rangle_3 = \mathbf{M}$. Then, the data provider distributes the shares among three cloud servers $C_{\{1,2,3\}}$ in such a way that C_1 holds the shares $(\langle\mathbf{M}\rangle_1, \langle\mathbf{M}\rangle_2)$, C_2 holds the shares $(\langle\mathbf{M}\rangle_2, \langle\mathbf{M}\rangle_3)$, and C_3 holds the shares $(\langle\mathbf{M}\rangle_3, \langle\mathbf{M}\rangle_1)$.

When the querier wants to issue an interval skyline query with interval $[t_1 : t_2]$, it first generates a secure query token by using the generator algorithm of DIFs. This is because we identify DIFs as a natural fit for efficient secure range predicate evaluation, which can help OblivTime to later securely locate time-series values within the interval. Formally, the querier runs $\mathsf{Gen}_{\mathsf{DIF}}(1^\lambda, t_1, t_2)$ and generates qk_1, qk_2, which are then sent to C_1, C_2 respectively. Upon receiving the secure query token, $C_{\{1,2,3\}}$ conduct oblivious interval skyline query processing as per OblivTime's custom protocol. At a high level, OblivTime's protocol proceeds through three phases, which are introduced as follows. Note that we summarize the purpose of each phase here and present their technical design in the following sections.

Oblivious database projection phase. In this phase, given the encrypted time-series database (in matrix form) $[\![\mathbf{M}]\!]$ and query token (qk_1, qk_2), $C_{\{1,2,3\}}$ obliviously project the encrypted database $[\![\mathbf{M}]\!]$ over the query interval to produce $[\![\mathbf{P}]\!]$, which only contains time-series values within the query interval.

Oblivious interval skyline identification phase. In this phase, given the encrypted and projected database $[\![\mathbf{P}]\!]$, $C_{\{1,2,3\}}$ obliviously fetch the encrypted IDs $[\![\mathcal{I}]\!]$ of all the skyline time series.

Oblivious interval skyline retrieval phase. In this phase, given the encrypted time-series database $[\![\mathbf{M}]\!]$ and encrypted IDs $[\![\mathcal{I}]\!]$ of all skyline time series, $C_{\{1,2,3\}}$ obliviously retrieve each skyline time series from $[\![\mathbf{M}]\!]$ to form the encrypted query result $[\![\mathcal{R}]\!]^3$, which can then be returned to the querier for reconstruction.

5 Oblivious Database Projection Phase

5.1 Overview

In this phase, $C_{\{1,2,3\}}$ securely project the encrypted time-series database $[\![\mathbf{M}]\!]$ to $[\![\mathbf{P}]\!]$ which only contains encrypted time-series data within the query interval, while being oblivious to the secret query interval. Our basic idea is to first leverage the FSS construction DIF for obliviously evaluating each timestamp $t \in [t_a : t_c]$ in the window, producing an encrypted flag for each timestamp. Let $\mathbf{r}$ be the underlying flag vector, and $\mathbf{r}[k]$ ($k \in [1, W]$) denote the flag for timestamp $t_a + k - 1$. The value of $\mathbf{r}[k]$ is 1 if the corresponding timestamp $t_a + k - 1$ in the window falls within the query interval, and is 0 otherwise. Given these encrypted flags, we then consider how to securely obtain the encrypted times series data corresponding to the eligible timestamps whose flags are 1. It is noted that directly opening the flags will leak which timestamp is eligible and thus compromise the privacy of the query interval.

Algorithm 2 Oblivious Database Projection $\mathsf{OblivProject}$

Input: The encrypted time-series database $[\![\mathbf{M}]\!]$ held by $C_{\{1,2,3\}}$, and secure query token qk_1, qk_2 held by C_1, C_2 respectively.

Output: $C_{\{1,2,3\}}$ hold the encrypted and projected database $[\![\mathbf{P}]\!]$.

1: Let N be the number of time series, $[t_a : t_c]$ be the W timestamps in the sliding-window of the time-series database.
2: **for** $t = 1$ to W **do**
3: $\quad C_i, i \in \{1,2\}$: $\langle\mathbf{r}[t]\rangle_i^2 = \mathsf{Eval}_{\mathsf{DIF}}(i, qk_i, t_a + t - 1)$. // $\langle\cdot\rangle_i^2$ denotes 2-out-of-2 additive shares held by C_i
4: **end for**
 // $C_{\{1,2,3\}}$ execute:
5: $C_{\{1,2,3\}}$ convert $[\![\mathbf{r}]\!]^2$ to $[\![\mathbf{r}]\!]$. // $[\![\cdot]\!]^2$ denotes additive sharing among C_1 and C_2.
6: Set $[\![\mathbf{G}]\!]$ to be a matrix where $[\![\mathbf{G}[1:N]]\!] = [\![\mathbf{M}]\!]$, $[\![\mathbf{G}[N+1]]\!] = [\![\mathbf{r}]\!]$.
7: $[\![\mathbf{G}']\!] = \mathsf{OblivColShuffle}([\![\mathbf{G}]\!])$.
8: Open $[\![\mathbf{G}'[N+1]]\!]$ and let $\mathbf{r}' = \mathbf{G}'[N+1]$.
9: Let $I = \sum_{t=1}^{W} \mathbf{r}'[t]$ and $[\![\mathbf{P}]\!]$ be a matrix of size $N \times I$.
10: $m = 1$.
11: **for** $t = 1$ to W **do**
12: $\quad$ **if** $\mathbf{r}'[t] = 1$ **then**
13: $\quad\quad [\![\mathbf{P}[1:N][m]]\!] = [\![\mathbf{G}'[1:N][t]]\!]$.
14: $\quad\quad m = m + 1$.
15: $\quad$ **end if**
16: **end for**
17: **return** $[\![\mathbf{P}]\!]$.

Our key insight is to first have the cloud servers perform an oblivious shuffle on the encrypted flags as well as their associated encrypted time-series values. The shuffle process is oblivious, meaning that the cloud servers cannot learn the underlying time-series values and flags being shuffled, as well as the random permutation being applied for the shuffle. Specifically, given a secret-shared matrix $[\![\mathbf{G}]\!]$ consisting of the time-series values and flags (as will be detailed below in Section 5.2), the oblivious shuffle protocol will allow the cloud servers to produce a secret-shared matrix $[\![\mathbf{G}']\!]$ such that $\mathbf{G}'$ is a column-wise shuffled version of $\mathbf{G}$. And the cloud servers cannot learn $\mathbf{G}$ and $\mathbf{G}'$, and the random permutation π applied to $\mathbf{G}$.

Hence, the *obliviously shuffled* flags can then be opened, allowing the identification of the encrypted time-series values that fall within the query interval while hiding the secret query interval. That is to say, the flags being opened are (obliviously) shuffled ones (i.e., in a permuted space).

Algorithm 3 The OblivColShuffle Protocol for Oblivious Column Shuffling

Input: The encrypted matrix $[\![\mathbf{G}]\!]$.
Output: The encrypted matrix $[\![\mathbf{G}']\!]$ such that $\mathbf{G}'$ is a column-wise shuffled version of $\mathbf{G}$.

1: C_i and C_{i-1} in advance mutually generate random permutation π_i^{col}, random matrix $\mathbf{Z}_i, i \in \{1,2,3\}$.
2: C_3 and C_1 in advance mutually generate random matrix $\mathbf{G}_1'$, C_1 and C_2 mutually generate random matrix $\mathbf{G}_2'$.
3: C_1: $\mathbf{X}_1 = \pi_2^{\mathsf{col}}(\langle\mathbf{G}\rangle_1 + \langle\mathbf{G}\rangle_2 + \mathbf{Z}_2)$; $\mathbf{X}_2 = \pi_1^{\mathsf{col}}(\mathbf{X}_1 + \mathbf{Z}_1)$.
4: C_2: $\mathbf{Y}_1 = \pi_2^{\mathsf{col}}(\langle\mathbf{G}\rangle_3 - \mathbf{Z}_2)$.
5: C_1 sends $\mathbf{X}_2$ to C_2 , and C_2 sends $\mathbf{Y}_1$ to C_3.
// After $\mathbf{X}_2$ and $\mathbf{Y}_1$ are received:
6: C_2: $\mathbf{X}_3 = \pi_3^{\mathsf{col}}(\mathbf{X}_2 + \mathbf{Z}_3)$; $\mathbf{W}_1 = \mathbf{X}_3 - \mathbf{G}_2'$.
7: C_3: $\mathbf{Y}_2 = \pi_1^{\mathsf{col}}(\mathbf{Y}_1 - \mathbf{Z}_1)$; $\mathbf{Y}_3 = \pi_3^{\mathsf{col}}(\mathbf{Y}_2 - \mathbf{Z}_3)$; $\mathbf{W}_2 = \mathbf{Y}_3 - \mathbf{G}_1'$.
8: C_2 sends $\mathbf{W}_1$ to C_3 and C_3 sends $\mathbf{W}_2$ to C_2.
// After $\mathbf{W}_1$ and $\mathbf{W}_2$ are received:
9: C_2 and C_3: $\mathbf{G}_3' = \mathbf{W}_1 + \mathbf{W}_2$.
10: $C_i, i \in \{1,2,3\}$: $[\![\mathbf{G}']\!]_i = (\mathbf{G}_i', \mathbf{G}_{i+1}')$.
11: **return** $[\![\mathbf{G}']\!]$.

Combining the insights above, we devise the detailed construction for the oblivious database projection OblivProject, which is introduced in what follows.

5.2 Detailed Construction

Algorithm 2 gives the detailed construction of OblivProject. Firstly, with $qk_i, i \in \{1,2\}$, C_i invokes the evaluation algorithm of DIF for every timestamp $t \in [t_a : t_c]$ in the window, and produces 2-out-of-2 additive shares $\langle\mathbf{r}[k]\rangle_i^2 = \mathsf{Eval}_{\mathsf{DIF}}(i, qk_i, t)$, where $k \in [1:W]$ and $t = t_a + k - 1$. This produces a 2-out-of-2 additive sharing of the flag vector $\mathbf{r}$ among C_1 and C_2, which we denote by $[\![\mathbf{r}]\!]^2$. Note that we will continue to use such notation $\langle\cdot\rangle_i^2$ and $[\![\cdot]\!]^2$ hereafter to denote 2-out-of-2 additive shares held by C_i and 2-out-of-2 additive sharings among C_1 and C_2, respectively. As the values in time-series data are protected under RSS, the RSS sharing $[\![\mathbf{r}]\!]$ of the flag vector $\mathbf{r}$ is needed for facilitating the subsequent secure computation with the encrypted times series data. This is achieved as follows. Firstly, $C_{\{1,2,3\}}$ generate a 3-out-of-3 additive sharing of the zero vector such that C_1 holds $\mathbf{k}_1$, C_2 holds $\mathbf{k}_2$, and C_3 holds $\mathbf{k}_3$, where $\mathbf{k}_1 + \mathbf{k}_2 + \mathbf{k}_3 = \mathbf{0}$. This can be done using the PRF-based technique introduced in Section 3.2. Then, C_1 computes $\langle\tilde{\mathbf{r}}\rangle_1 = \langle\mathbf{r}\rangle_1 + \mathbf{k}_1$, C_2 computes $\langle\tilde{\mathbf{r}}\rangle_2 = \langle\mathbf{r}\rangle_2 + \mathbf{k}_2$, and C_3 sets $\langle\tilde{\mathbf{r}}\rangle_3 = \mathbf{k}_3$. Next, C_i sends $\langle\tilde{\mathbf{r}}\rangle_i$ to C_{i-1}, so that each C_i has $[\![\mathbf{r}]\!]_i = (\langle\tilde{\mathbf{r}}\rangle_i, \langle\tilde{\mathbf{r}}\rangle_{i+1})$, for $i \in \{1,2,3\}$. Through these steps, the RSS sharing $[\![\mathbf{r}]\!]$ of the flags $\mathbf{r}$ is produced among C_1, C_2, C_3.

As mentioned above, the cloud servers then need to perform an oblivious shuffle on the encrypted flags as well as their associated encrypted time-series values. To this end, $C_{\{1,2,3\}}$ first form an encrypted extended matrix $[\![\mathbf{G}]\!]$ with $N+1$ rows, where the first N rows are filled with the RSS sharing $[\![\mathbf{M}]\!]$ and the last row is filled with the RSS sharing $[\![\mathbf{r}]\!]$. Maintaining the alignment of encrypted time-series values at the same timestamp is essential for preserving the original dominance relationships among different time series. So the (oblivious) shuffle on $[\![\mathbf{G}]\!]$ should be performed in a column-wise manner. As $[\![\mathbf{G}]\!]$ is in RSS form, what we need here is an oblivious shuffle protocol that can work in the RSS sharing domain. We identify that the state-of-the-art protocol in [22] is a good fit for our purpose, which takes as input the RSS sharing of the data to be shuffled and outputs a new RSS sharing of the shuffled data. The protocol in [22] works with encrypted data under binary RSS sharing, and we adapt it to work with data under arithmetic RSS sharing in our context.

$C_{\{1,2,3\}}$ collectively shuffle the encrypted extended matrix $[\![\mathbf{G}]\!]$. Algorithm 3 shows how $C_{\{1,2,3\}}$ collaborate in the oblivious shuffle protocol on $[\![\mathbf{G}]\!]$, which we denote as $[\![\mathbf{G}']\!] = [\![\pi^{\mathsf{col}}(\mathbf{G})]\!] = \mathsf{OblivColShuffle}([\![\mathbf{G}]\!])$. Similar to RSS, OblivColShuffle also decomposes the overall permutation into three shares, so any server with a strict subset of the shares cannot learn the overall permutation applied. Note that in Algorithm 3 the materials π_i^{col}, $\mathbf{Z}_i$, $\mathbf{G}_1'$, and $\mathbf{G}_2'$ are data-independent and they can be easily prepared offline using preshared seed [22]. For the output of OblivColShuffle, i.e., $[\![\mathbf{G}']\!] = \mathsf{OblivColShuffle}([\![\mathbf{G}]\!])$, the matrix $\mathbf{G}'$ contains the same columns as $\mathbf{G}$, but arranged in a random order determined by the permutation $\pi^{\mathsf{col}} = \pi_3^{\mathsf{col}} \circ \pi_1^{\mathsf{col}} \circ \pi_2^{\mathsf{col}}$. The correctness of OblivColShuffle holds as the sum of the shuffle result $\sum_{i=1}^{3} \mathbf{G}_i' = \mathbf{G}_1' + \mathbf{G}_2' + \pi_3^{\mathsf{col}}(\pi_1^{\mathsf{col}}(\pi_2^{\mathsf{col}}(\langle\mathbf{G}\rangle_1 + \langle\mathbf{G}\rangle_2 + \mathbf{Z}_2) + \mathbf{Z}_1) + \mathbf{Z}_3) - \mathbf{G}_2' + \pi_3^{\mathsf{col}}(\pi_1^{\mathsf{col}}(\pi_2^{\mathsf{col}}(\langle\mathbf{G}\rangle_3 - \mathbf{Z}_2) - \mathbf{Z}_1) - \mathbf{Z}_3) - \mathbf{G}_1' = \pi_3^{\mathsf{col}}(\pi_1^{\mathsf{col}}(\pi_2^{\mathsf{col}}(\langle\mathbf{G}\rangle_1 + \langle\mathbf{G}\rangle_2 + \langle\mathbf{G}\rangle_3))) = \pi^{\mathsf{col}}(\mathbf{G})$. Therefore, OblivColShuffle correctly generates the shuffle result. As C_i only has π_i, π_{i+1} and knows nothing about π_{i-1}, so the overall permutation π applied is unknown to any single server. It is noted that as the secret random permutation is applied column-wise, the time series values are not changed and the alignment of encrypted time-series values (along with the corresponding flag) at the same timestamp is preserved after the random permutation. This ensures that the original dominance relationships among different time series after the random permutation are preserved, and so the correctness of skyline query result is not affected.

Once $[\![\mathbf{G}']\!]$ is produced, $C_{\{1,2,3\}}$ safely open the shuffled flag vector, denoted as $\mathbf{r}'$ (which corresponds to the last row in $[\![\mathbf{G}']\!]$, i.e., $[\![\mathbf{r}']\!] = [\![\mathbf{G}'[N+1]]\!]$). Based on the opened flags, $C_{\{1,2,3\}}$ can easily produce the encrypted projected database $[\![\mathbf{P}]\!]$ (i.e., lines 9–16 in Algorithm 2), which contains only the encrypted time-series values within the query interval.

Remark. We emphasize that opening the shuffled flag vector does not reveal the secret query interval $[t_a : t_c]$ to $C_{\{1,2,3\}}$, as the random permutation applied to the flag vector is unknown to any individual server. This prevents them from inferring which timestamps actually falling within the query interval based on the opened flags. What $C_{\{1,2,3\}}$ can learn about the query interval is its length I only (through the count of 1s in $\mathbf{r}'$). In other words, $C_{\{1,2,3\}}$ can only learn the fact that $t_c - t_a = I - 1$ and nothing more. We are not aware of any concrete harm from revealing this type of size information.

6 Oblivious Interval Skyline Identification Phase

6.1 Overview

After obtaining the projected and encrypted database $[\![\mathbf{P}]\!]$, the cloud servers proceed to the oblivious interval skyline

Algorithm 4 Oblivious Interval Skyline Fetching OblivFetch

Input: The projected and encrypted database $[\![\mathbf{P}]\!]$, the encrypted IDs $[\![\mathbf{id}]\!]$, the encrypted sum $[\![\mathbf{s}^{(k)}]\!]$.

Output: The encrypted skyline row $[\![\mathbf{p}_*]\!]$, its encrypted ID $[\![id_*]\!]$, and its encrypted sum $[\![s_*]\!]$.

// $C_{\{1,2,3\}}$ jointly execute:

1: $[\![\mathbf{p}_*]\!] = [\![\mathbf{P}[1]]\!]$; $[\![id_*]\!] = [\![\mathbf{id}[1]]\!]$; $[\![s_*]\!] = [\![\mathbf{s}^{(k)}[1]]\!]$; and N is the number of rows in $[\![\mathbf{P}]\!]$.
2: **for** $i = 2$ to N **do**
3: $\quad [\![\varphi]\!] = \mathsf{secCmp}([\![s_*]\!], [\![\mathbf{s}^{(k)}[i]]\!])$.
4: $\quad [\![\mathbf{p}_*]\!] = [\![\varphi]\!] \times [\![\mathbf{P}[i]]\!] + (1 - [\![\varphi]\!]) \times [\![\mathbf{p}_*]\!]$.
5: $\quad [\![id_*]\!] = [\![\varphi]\!] \times [\![\mathbf{id}[i]]\!] + (1 - [\![\varphi]\!]) \times [\![id_*]\!]$.
6: $\quad [\![s_*]\!] = [\![\varphi]\!] \times [\![\mathbf{s}^{(k)}[i]]\!] + (1 - [\![\varphi]\!]) \times [\![s_*]\!]$.
7: **end for**
8: **return** $[\![\mathbf{p}_*]\!], [\![id_*]\!], [\![s_*]\!]$.

identification phase, which aims to produce the encrypted IDs of the skyline time series. Recall that the ID of each time series in OblivTime is their system-wide index $i \in [1:N]$. With respect to the computation in Algorithm 1, we design two essential oblivious components: (i) oblivious interval skyline fetching OblivFetch and (ii) oblivious interval skyline and dominated time series filtering OblivFilter. The OblivFetch component obliviously fetches an interval skyline time series, and the OblivFilter component obliviously filters the skyline and the time series dominated by the skyline. Based on these two components, we derive the complete protocol for oblivious interval skyline identification.

6.2 Oblivious Interval Skyline Fetching

We now introduce OblivFetch where $C_{\{1,2,3\}}$ jointly fetch the encrypted information corresponding to a new skyline time series. OblivFetch takes as input the projected and encrypted database $[\![\mathbf{P}]\!]$, the encrypted IDs of the corresponding time series of each row $[\![\mathbf{id}]\!]$, and the encrypted sum $[\![\mathbf{s}^{(k)}]\!]$ of each row of $[\![\mathbf{P}]\!]$ in the round k (where $\mathbf{s}^{(k)}[i] = \sum \mathbf{P}[i][:], i \in [1:N]$), and then outputs the encrypted row that corresponds to a skyline time series (such row will be referred to as a skyline row in the following for the sake of simplicity) $[\![\mathbf{p}_*]\!]$, its encrypted ID $[\![id_*]\!]$, and its encrypted sum $[\![s_*]\!]$.

Firstly, $C_{\{1,2,3\}}$ initialize the first row of $[\![\mathbf{P}]\!]$ as the (temporary) skyline row (line 1). After that, $C_{\{1,2,3\}}$ iterate over the remaining rows. $C_{\{1,2,3\}}$ first invoke a secure comparison protocol $\mathsf{secCmp}(\cdot,\cdot)$ (introduced below) on the sum of the current skyline row $[\![s_*]\!]$ and $[\![\mathbf{s}^{(k)}[i]]\!]$ where $i \in [2:N]$ (line 3), and output $[\![\varphi]\!]$. Here, given two secret-shared values $[\![a]\!]$ and $[\![b]\!]$, $\mathsf{secCmp}([\![a]\!], [\![b]\!])$ outputs $[\![\varphi]\!] = [\![1]\!]$ if $a < b$, and outputs $[\![\varphi]\!] = [\![0]\!]$ if $a \geq b$. Subsequently, $C_{\{1,2,3\}}$ leverage $[\![\varphi]\!]$ to obliviously update $[\![\mathbf{p}_*]\!]$, $[\![id_*]\!]$, as well as $[\![s_*]\!]$, so they correspond to the row with the maximum sum encountered thus far (lines 4-6). Algorithm 4 gives the detailed construction of OblivFetch.

We construct $\mathsf{secCmp}(\cdot,\cdot)$ based on FSS-based DIF, detailed in Algorithm 5. The FSS-based DIF evaluation requires cloud servers to operate on identical inputs, which is incompatible with our secret-shared database. To address this, we employ the method from [32], where cloud servers work with additively masked variants of the secret values and customize the generation of FSS-based DIF keys for

Algorithm 5 Secure Comparison secCmp

Input: RSS sharings $[\![a]\!]$ and $[\![b]\!]$.

Output: The RSS sharing of the comparison result $[\![\varphi]\!]$ where $\varphi = 1$ if $a < b$ and $\varphi = 0$ otherwise.

1: C_3 samples a random value r and generates $(k_1^r, k_2^r) = \mathsf{Gen}_{\mathsf{DIF}}(1^\lambda, 2^{l-1} + r, 2^l - 1 + r)$.
2: C_3 splits r into two additive secret shares $\langle r\rangle_1^2$, $\langle r\rangle_2^2$, where $r = \langle r\rangle_1^2 + \langle r\rangle_2^2$, and sends $(k_i^r, \langle r\rangle_i^2)$ to C_i where $i \in \{1,2\}$.
3: $C_{\{1,2,3\}}$ compute $[\![c]\!] = [\![a]\!] - [\![b]\!]$.
4: C_1 sends $\langle c\rangle_1 + \langle r\rangle_1^2$ to C_2, and C_2 sends $\langle c\rangle_3 + \langle r\rangle_2^2$ to C_1, where $(\langle c\rangle_i, \langle c\rangle_{i+1}) = [\![c]\!]_i$ is the RSS shares of c held by $C_i, i \in \{1,2\}$.
5: C_1 and C_2 locally reconstruct $c + r = \langle c\rangle_1 + \langle c\rangle_2 + \langle c\rangle_3 + \langle r\rangle_1^2 + \langle r\rangle_2^2$ with received message and private shares.
6: $C_i, i \in \{1,2\}$ locally evaluates $\langle\varphi\rangle_i^2 = \mathsf{Eval}_{\mathsf{DIF}}(i, k_i^r, c + r)$.
7: $C_{\{1,2,3\}}$ reshare $\langle\varphi\rangle^2$ to obtain $[\![\varphi]\!]$.
8: **return** $[\![\varphi]\!]$.

these masked values. To compare two values a and b, the idea is to mask the subtraction result of the input values with a random offset r and reconstruct the masked result to facilitate FSS key evaluation. Given $a, b \in \mathbb{Z}_{2^l}$, the masked subtraction result $a-b+r$ falls within $[2^{l-1}+r : 2^l-1+r]$ under two's complement representation when $a < b$. Therefore, we let C_3 act as a dealer who generates a random offset and a pair of DIF keys for the corresponding interval function with the offset and then splits the random offset into a pair of secret shares to conform to secret-shared input of secCmp. The offset shares and DIF keys are distributed to C_1 and C_2, who then evaluate the keys on the masked subtraction result to obtain the comparison result, 2-out-of-2-shared $\langle\varphi\rangle^2$. Finally, $C_{\{1,2,3\}}$ apply reshare (introduced in Section 5.2) to $\langle\varphi\rangle^2$ to produce RSS-shared output $[\![\varphi]\!]$. In some cases, the final resharing is not necessary. We denote a variant of secCmp, called $\mathsf{secCmp}^{\mathsf{R}\to 2}$, which takes RSS-shared values as input and outputs the comparison result in 2-out-of-2-shared form. The only difference from the original secCmp is that the final resharing step (line 7) is omitted, directly outputting $\langle\varphi\rangle^2$, with C_1 holding $\langle\varphi\rangle_1^2$ and C_2 holding $\langle\varphi\rangle_2^2$.

6.3 Oblivious Interval Skyline and Dominated Time Series Filtering

After fetching the encrypted information (i.e., $[\![\mathbf{p}_*]\!]$, and $[\![id_*]\!], [\![s_*]\!]$) of a skyline time series in $[\![\mathbf{P}]\!]$, the cloud servers then leverage OblivFilter to obliviously filter the corresponding skyline row and the rows dominated by it from $[\![\mathbf{P}]\!]$. This guarantees that these filtered rows will not be retrieved as skylines in future rounds. Algorithm 7 shows the OblivFilter protocol, which takes as input the encrypted sum $[\![\mathbf{s}]\!]$ in the current round k (denoted as $[\![\mathbf{s}^{(k)}]\!]$), and outputs the updated encrypted sum for the next round $k{+}1$, i.e., $[\![\mathbf{s}^{(k+1)}]\!]$. To simplify the following process, we denote the numbers of rows and columns of $[\![\mathbf{P}]\!]$ as N and I respectively (line 1). We consider how to determine the fetched skyline time series and other time series dominated by it. For each row $[\![\mathbf{P}[i]]\!], i \in [1:N]$ in $[\![\mathbf{P}]\!]$ (line 2), $C_{\{1,2,3\}}$ first invoke a secure equality test protocol secEqual over the encrypted

Algorithm 6 Secure Equality Test secEqual

Input: RSS sharings $[\![a]\!]$ and $[\![b]\!]$.

Output: RSS sharing of the equality test result $[\![\varphi]\!]$ where $\varphi = 1$ if $a = b$ and $\varphi = 0$ otherwise.

1: C_3 samples a random value r and generates $(k_1^r, k_2^r) = \mathsf{Gen}_{\mathsf{DPF}}(1^\lambda, r, 1)$.
2: C_3 splits r into two additive secret shares $\langle r\rangle_1^2$, $\langle r\rangle_2^2$, where $r = \langle r\rangle_1^2 + \langle r\rangle_2^2$, and sends $(k_i^r, \langle r\rangle_i^2)$ to C_i where $i \in \{1, 2\}$.
3: $C_{\{1,2,3\}}$ compute $[\![c]\!] = [\![a]\!] - [\![b]\!]$.
4: C_1 sends $\langle c\rangle_1 + \langle r\rangle_1^2$ to C_2, and C_2 sends $\langle c\rangle_3 + \langle r\rangle_2^2$ to C_1, where $(\langle c\rangle_i, \langle c\rangle_{i+1}) = [\![c]\!]_i$ is the RSS shares of c held by $C_i, i \in \{1, 2\}$.
5: C_1 and C_2 locally reconstruct $c + r = \langle c\rangle_1 + \langle c\rangle_2 + \langle c\rangle_3 + \langle r\rangle_1^2 + \langle r\rangle_2^2$ with received message and private shares.
6: $C_i, i \in \{1, 2\}$ locally evaluates $\langle\varphi\rangle_i^2 = \mathsf{Eval}_{\mathsf{DPF}}(i, k_i^r, c+r)$.
7: $C_{\{1,2,3\}}$ reshare $\langle\varphi\rangle^2$ to obtain $[\![\varphi]\!]$.
8: **return** $[\![\varphi]\!]$.

IDs of $[\![\mathbf{p}_*]\!]$ and $[\![\mathbf{P}[i]]\!]$ to determine whether $[\![\mathbf{P}[i]]\!]$ is the skyline $[\![\mathbf{p}_*]\!]$, where $\alpha = 1$ means that $[\![\mathbf{P}[i]]\!]$ is the skyline (line 3). Here, secEqual takes as input secret-shared values $[\![a]\!]$ and $[\![b]\!]$, and outputs $[\![1]\!]$ if $a = b$, and outputs $[\![0]\!]$ if $a \neq b$. The equality test can be achieved by checking if $a - b$ equals 0 with point function $f_{0,1}$, which yields 1 only when evaluated at 0 and yields 0 otherwise. So, similar to secCmp, secEqual utilize FSS-based DPF to perform oblivious testing of equality between a and b. Detailed steps for secEqual are provided in Algorithm 6. The OblivTime utilizes variants of secEqual with different input and output forms, denoted as $\mathsf{secEqual}^{\mathsf{x}\to\mathsf{y}}$, where x and y can be 2 (2-out-of-2 shared) or R (RSS shared). For instance, $\mathsf{secEqual}^{2\to\mathsf{R}}$ accepts 2-out-of-2 shared values and produces an RSS-shared equality test result. Variants that accept RSS-shared inputs follow the steps outlined in lines 3–4 of Algorithm 6. The process for inputting 2-out-of-2 values $\langle a\rangle^2$ and $\langle b\rangle^2$ differs slightly. In the subtraction step (line 3), C_1 and C_2 compute $\langle c\rangle^2 = \langle a\rangle^2 - \langle b\rangle^2$. During the share exchange step (line 4), C_1 sends $\langle c\rangle_1^2 + \langle r\rangle_1^2$, while C_2 sends $\langle c\rangle_2^2 + \langle r\rangle_2^2$. In the reconstruction step (line 5), $c + r$ is reconstructed as $\langle c\rangle_1^2 + \langle c\rangle_2^2 + \langle r\rangle_1^2 + \langle r\rangle_2^2$. Variants producing RSS-shared results apply reshare to $\langle\varphi\rangle^2$ to obtain $[\![\varphi]\!]$ (line 7), while those returning 2-out-of-2 sharings directly output $\langle\varphi\rangle^2$.

$C_{\{1,2,3\}}$ then determine whether $[\![\mathbf{P}[i]]\!]$ is dominated by $\mathbf{p}_*$. More specifically, the sums of $[\![\mathbf{p}_*]\!]$ and the row $[\![\mathbf{P}[i]]\!]$ are first compared by invoking $\mathsf{secCmp}^{\mathsf{R}\to 2}$ on $[\![\mathbf{s}^{(k)}[i]]\!]$ and $[\![s_*]\!]$ (line 4). Then, for every column $m \in [1 : I]$, $C_{\{1,2,3\}}$ obliviously compare the encrypted time-series values using $\mathsf{secCmp}^{\mathsf{R}\to 2}$ to get comparison result $[\![\gamma_m]\!]^2$ (lines 5–7). Afterwards, $C_{\{1,2,3\}}$ obliviously combine the encrypted results to calculate the exact dominance relationship and output $[\![\epsilon]\!]$ (line 8). Here, $\epsilon = 1$ means that $[\![\mathbf{P}[i]]\!]$ is dominated by $\mathbf{p}_*$ and needs to be filtered. After that, $C_{\{1,2,3\}}$ combine $[\![\alpha]\!]$ and $[\![\epsilon]\!]$ (line 9), and obliviously filter $\mathbf{P}[i]$ based on $[\![\zeta]\!]$ (line 10). Here, ζ can be either 1 or 0, as α and ϵ cannot be 1 simultaneously. $\zeta = 1$ indicates that $\mathbf{P}[i]$ is either the skyline or dominated by the skyline, and should be filtered; $\zeta = 0$ indicates that $\mathbf{P}[i]$ should not be filtered.

To obliviously filter $\mathbf{P}[i]$, inspired by [17], we let $C_{\{1,2,3\}}$

Algorithm 7 Oblivious Interval Skyline and Dominated Time Series Filtering OblivFilter

Input: The projected and encrypted database $[\![\mathbf{P}]\!]$, the encrypted IDs $[\![\mathbf{id}]\!]$, the encrypted sum $[\![\mathbf{s}^{(k)}]\!]$, the encrypted skyline row $[\![\mathbf{p}_*]\!]$, the encrypted ID $[\![id_*]\!]$ of the skyline row, and the encrypted sum $[\![s_*]\!]$ of the skyline row.

Output: The updated encrypted sum $[\![\mathbf{s}^{(k+1)}]\!]$.

// $C_{\{1,2,3\}}$ jointly execute:
1: Let N and I be the number of rows and columns of $[\![\mathbf{P}]\!]$.
2: **for** $i = 1$ to N **do**
3: $\quad[\![\alpha]\!] = \mathsf{secEqual}([\![\mathbf{id}[i]]\!], [\![id_*]\!])$.
4: $\quad[\![\beta]\!]^2 = \mathsf{secCmp}^{\mathsf{R}\to 2}([\![\mathbf{s}^{(k)}[i]]\!], [\![s_*]\!])$.
5: $\quad$**for** $m = 1$ to I **do**
6: $\quad\quad[\![\gamma_m]\!]^2 = \mathsf{secCmp}^{\mathsf{R}\to 2}([\![\mathbf{p}_*[m]]\!], [\![\mathbf{P}[i][m]]\!])$.
7: $\quad$**end for**
8: $\quad[\![\epsilon]\!] = \mathsf{secEqual}^{2\to\mathsf{R}}([\![\beta]\!]^2 + \sum_{m=1}^{I}(1 - [\![\gamma_m]\!]^2), [\![I+1]\!]^2)$.
9: $\quad[\![\zeta]\!] = [\![\alpha]\!] + [\![\epsilon]\!]$.
10: $\quad[\![\mathbf{s}^{(k+1)}[i]]\!] = \mathsf{vMIN} \times [\![\zeta]\!] + [\![\mathbf{s}^{(k)}[i]]\!] \times (1 - [\![\zeta]\!])$.
11: **end for**
12: **return** $[\![\mathbf{s}^{(k+1)}]\!]$.

perform $[\![\mathbf{s}^{(k+1)}[i]]\!] = \mathsf{vMIN} \times [\![\zeta]\!] + [\![\mathbf{s}^{(k)}[i]]\!] \times (1 - [\![\zeta]\!])$, where $\zeta = 1$ means that the i-th row should be filtered and $\mathbf{s}^{(k+1)}[i]$ is obliviously set as a preset minimum value vMIN; $\zeta = 0$ means that the i-th row need not be filtered and $\mathbf{s}^{(k+1)}[i]$ remains unchanged. As OblivFetch fetches the row with the largest sum in $[\![\mathbf{s}^{(k)}]\!]$ to be the skyline row, so modifying the sum of the rows to a preset minimum value vMIN can prevent the filtered rows from being fetched as a skyline in the subsequent rounds.

6.4 Putting It Together

With the above secure components, we now present the complete protocol for oblivious interval skyline identification, which is shown in Algorithm 8. Here, what we need to consider is how to securely determine whether all time series have been filtered because the identification process should end when all time series are marked as filtered. Inspired by [17], we let the cloud servers obliviously evaluate whether the sum of the fetched time series in the current round s_* (i.e., the largest sum in $\mathbf{s}^{(k)}$) is equal to the preset minimum value vMIN. If s_* equals vMIN, it indicates that all rows have been filtered and this skyline identification process should be terminated. To achieve this, $C_{\{1,2,3\}}$ first invoke $\mathsf{secEqual}^{\mathsf{R}\to 2}$ on $[\![s_*]\!]$ and $[\![\mathsf{vMIN}]\!]$ to get the test result $[\![\mathsf{isStop}]\!]^2$ (line 6). To securely open isStop (line 7), C_1 sends $\langle\mathsf{isStop}\rangle_1^2 + \alpha$ and C_2 sends $\langle\mathsf{isStop}\rangle_2^2 - \alpha$ to all servers, where α is a PRF-generated value derived from a pre-shared seed between C_1 and C_2, where isStop = 1 indicates that the largest value in $\mathbf{s}^{(k)}$ equals vMIN, i.e., all time series are deleted and $C_{\{1,2,3\}}$ can terminate OblivIdentify (lines 8–10); isStop = 0 indicates that not all time series have been filtered and $C_{\{1,2,3\}}$ should continue to perform OblivIdentify.

7 Oblivious Interval Skyline Retrieval Phase

So far we have introduced how $C_{\{1,2,3\}}$ obliviously identify the skyline time series. In this section, we will introduce

Algorithm 8 Oblivious Interval Skyline Identification OblivIdentify

Input: The projected and encrypted database $[\![\mathbf{P}]\!]$.
Output: The set of encrypted IDs $[\![\mathcal{I}]\!]$ of interval skyline.

// $C_{\{1,2,3\}}$ jointly execute:
1: Let N and I be the number of rows and columns of $[\![\mathbf{P}]\!]$.
2: $[\![\mathbf{s}^{(0)}[i]]\!] = \sum_{j=1}^{I} [\![\mathbf{P}[i][j]]\!]$, $[\![\mathbf{id}]\!] = [\![i]\!]$, where $i \in [1:N]$.
3: Set $k = 0$, and $[\![\mathcal{I}]\!]$ an empty set.
4: **while** true **do**
5: $[\![\mathbf{p}_*]\!], [\![id_*]\!], [\![s_*]\!] = \mathsf{OblivFetch}([\![\mathbf{P}]\!], [\![\mathbf{id}]\!], [\![\mathbf{s}^{(k)}]\!])$.
6: $[\![\mathsf{isStop}]\!]^2 = \mathsf{secEqual}^{\mathsf{R}\to 2}([\![s_*]\!], [\![\mathsf{vMIN}]\!])$.
7: $C_{\{1,2,3\}}$ securely open isStop.
8: **if** isStop = 1 **then**
9: **break**
10: **end if**
11: $[\![\mathbf{s}^{(k+1)}]\!] = \mathsf{OblivFilter}([\![\mathbf{P}]\!], [\![\mathbf{s}^{(k)}]\!], [\![\mathbf{p}_*]\!], [\![id_*]\!], [\![s_*]\!])$.
12: $[\![\mathcal{I}]\!].\mathsf{append}([\![id_*]\!])$.
13: $k = k + 1$.
14: **end while**
15: **return** $[\![\mathcal{I}]\!]$.

how $C_{\{1,2,3\}}$ obliviously retrieve them (i.e., the query result $[\![\mathcal{R}]\!]^3$) from the original encrypted database $[\![\mathbf{M}]\!]$ based on the encrypted ID vector $[\![\mathcal{I}]\!]$ of all skylines. Algorithm 9 presents the protocol OblivRetrieve. Firstly, $C_{\{1,2,3\}}$ initialize an empty set of 3-out-of-3 additive secret sharing $[\![\mathcal{R}]\!]^3$ to store the encrypted query result (line 1). $C_{\{1,2,3\}}$ then invokes the secure equality test protocol $\mathsf{secEqual}(\cdot,\cdot)$ on $[\![id_*]\!]$ and $[\![i]\!] = \mathsf{localShare}(i), i \in [1:N]$ for each encrypted ID $[\![id_*]\!]$ in $[\![\mathcal{I}]\!]$ (line 4), each of which outputs $[\![\mathbf{e}[i]]\!]$, where $\mathbf{e}[i] = 1$ means that $id_* = i$, and $\mathbf{e}[i] = 0$ means that $id_* \neq i$. $C_{\{1,2,3\}}$ then obliviously retrieve the interval skyline (denoted as $\mathbf{r}_*$) from $[\![\mathbf{M}]\!]$ via the encrypted vector $[\![\mathbf{e}]\!]$.

Our key idea is based on the fact that only one element of $\mathbf{e}$ equals 1, while the rest are set to 0. Therefore, we let $C_{\{1,2,3\}}$ multiply $[\![\mathbf{e}[i]]\!]$ with $[\![\mathbf{M}[i]]\!], i \in [1:N]$, and aggregate all the products to produce the encrypted skyline corresponding to id_*. In addition, we note that the above process involves a single-depth multiplication of RSS shares where the resharing operation at the end of each multiplication ($\times$) operation is no longer necessary. Specifically, $C_{\{1,2,3\}}$ can locally perform addition ($+$) over the 3-out-of-3 additive secret sharing of $\mathbf{e}[i] \times \mathbf{M}[i], i \in [1:N]$ to produce the 3-out-of-3 additive secret sharing of $\mathbf{r}_*$, i.e., C_j holds $\langle \mathbf{r}_* \rangle_j, j \in \{1,2,3\}$ and $\mathbf{r}_* = \langle \mathbf{r}_* \rangle_1 + \langle \mathbf{r}_* \rangle_2 + \langle \mathbf{r}_* \rangle_3$ (line 6). $C_j, j \in \{1,2,3\}$ then appends $\langle \mathbf{r}_* \rangle_j$ to the respective result set $\langle \mathcal{R} \rangle_j$, resulting in the query result under 3-out-of-3 additive secret sharing, denoted as $[\![\mathcal{R}]\!]^3$ (line 7). Finally, $C_{\{1,2,3\}}$ return $[\![\mathcal{R}]\!]^3$ to the querier for decryption.

8 Security Analysis

We follow the standard simulation-based paradigm [38] to prove the security of OblivTime. We assume a static semi-honest probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ who can corrupt at most one of $C_{\{1,2,3\}}$ and the corrupted server still strictly follows the protocol. The overall processing in OblivTime can be divided into three main components: Init, Append, and Query, which respectively handle system initialization, time-series database appending, and

Algorithm 9 Oblivious Interval Skyline Retrieval OblivRetrieve

Input: The projected and encrypted database $[\![\mathbf{M}]\!]$, and the set of encrypted IDs $[\![\mathcal{I}]\!]$ of interval skyline.
Output: The encrypted query result $[\![\mathcal{R}]\!]^3$.

1: Let $[\![\mathcal{R}]\!]^3$ be an empty set of 3-out-of-3 additive secret sharing, N be the number of rows of $[\![\mathbf{M}]\!]$.
2: **for** $[\![id_*]\!]$ in $[\![\mathcal{I}]\!]$ **do**
3: **for** $i = 1$ to N **do**
4: $[\![\mathbf{e}[i]]\!] = \mathsf{secEqual}([\![id_*]\!], [\![i]\!])$.
5: **end for**
// $C_j, j \in \{1,2,3\}$ locally performs the following:
6: $\langle \mathbf{r}_* \rangle_j = \sum_{i=1}^{N} (\langle \mathbf{e}[i] \rangle_j \times \langle \mathbf{M}[i] \rangle_j + \langle \mathbf{e}[i] \rangle_j \times \langle \mathbf{M}[i] \rangle_{j+1} + \langle \mathbf{e}[i] \rangle_{j+1} \times \langle \mathbf{M}[i] \rangle_j)$.
7: $\langle \mathcal{R} \rangle_j.\mathsf{append}(\langle \mathbf{r}_* \rangle_j)$.
8: **end for**
9: **return** $[\![\mathcal{R}]\!]^3$.

interval skyline query operations. We start by defining the ideal functionality $\mathcal{F}$ for OblivTime:

- $\mathsf{Init}(1^\lambda, \mathsf{schema})$: $\mathcal{F}$ initializes the time-series database $\mathcal{D}$ with the security parameter 1^λ and database scheme $\mathsf{schema} = (N, 2^l, W, \mathsf{vMIN})$ where N is the number of time series in the database, $\mathbb{Z}_{2^l}$ is the size of the data domain, W is the size limit of the database sliding window, and vMIN is the predefined minimum sum value. $\mathcal{F}$ also initializes an empty sliding window.
- $\mathsf{Append}((1,t,v_{1,t}),(2,t,v_{2,t}),\ldots,(N,t,v_{N,t}))$: $\mathcal{F}$ sets $\mathsf{v}_t = (v_{1,t}, v_{2,t}, \ldots, v_{N,t})$, updates $\mathcal{D}[:][t] = \mathbf{v}_t$, and adds new timestamp t to the sliding window of D. $\mathcal{F}$ then checks the size of the sliding window. If it exceeds the limit W, $\mathcal{F}$ will discard the oldest timestamp and its time-series values from $\mathcal{D}$.
- $\mathsf{Query}(t_1, t_2)$: $\mathcal{F}$ queries the interval skyline $\mathcal{R}$ on database $\mathcal{D}$ over interval $[t_1 : t_2]$, and outputs $\mathcal{R}$ only to the querier.

We allow $\mathcal{F}$ to leak $\mathsf{Leak}(\mathcal{F}) = (\mathsf{schema}, \mathsf{struct}_{\mathcal{Q}})$, where the query structure $\mathsf{struct}_{\mathcal{Q}}$ is defined as: (1) $(\mathsf{Init}, 1^\lambda)$ for Init, (2) (Append, t) for Append, (3) (Query, I, M) for Query where I is the length of query interval, and M is the number of time series of interval skyline result.

***Definition* 1.** *Let Π be a protocol for interval skyline query processing on an encrypted time-series database. $\mathcal{F}$ as described above models the interval skyline query functionality provided by an incorruptible, trusted party. $\mathcal{A}$ is a semi-honest adversary that statically corrupts at most one cloud server in the real world. We denote the view of $\mathcal{A}$ in the real world as $\mathsf{View}_{\Pi}^{\mathsf{Real}}$ and the view generated by a PPT simulator $\mathcal{S}$ in the ideal world as $\mathsf{View}_{\mathcal{S},\mathsf{Leak}(\mathcal{F})}^{\mathsf{Ideal}}$. We say that Π is secure in the semi-honest and non-colluding setting if $\forall \mathcal{A}, \exists \mathcal{S}$ s.t. $\mathsf{View}_{\Pi}^{\mathsf{Real}} \approx \mathsf{View}_{\mathcal{S},\mathsf{Leak}(\mathcal{F})}^{\mathsf{Ideal}}$. That is, for the view that any $\mathcal{A}$ observes in the real world, there is a simulator $\mathcal{S}$ in the ideal world that can simulate it such that the two views are computationally indistinguishable.*

***Theorem* 1.** *In the semi-honest and non-colluding adversary setting, OblivTime securely realizes the ideal functionality $\mathcal{F}$ according to Definition 1.*

Proof. We prove the security of OblivTime by showing the existence of the simulator $\mathcal{S}$ in the ideal world. Since in OblivTime the cloud servers C_1 and C_2 have symmetric

roles while C_3 has a different role, there are two cases to be considered: (1) $\mathcal{A}$ corrupts C_1 or C_2, and (2) $\mathcal{A}$ corrupts C_3. Below we show the existence of the simulator for each of the two cases respectively.

The case of C_1 or C_2 being corrupted. It is sufficient to demonstrate the existence of the simulator for the case that $\mathcal{A}$ corrupts C_1, as the symmetric roles of C_1 and C_2 imply that the simulator must also exist when $\mathcal{A}$ corrupts C_2. Note that the existence of the simulator for the overall processing can be proved by showing the existence of simulators for the three components Init, Append, and Query, respectively.

- *Simulator for* Init. C_1 only receives public parameters in Init, so the simulator can trivially generate the view by following the protocol. Therefore, such a simulator exists.
- *Simulator for* Append. In Append, C_1 receives replicated shares of time-series values and the corresponding timestamp while sending nothing. The simulator for Append can be easily constructed by invoking the RSS simulator [20] to generate replicated shares, and leveraging the fact that the timestamp is known to the simulator. So, the simulator for Append exists.
- *Simulator for* Query. Recall that OblivTime is composed of three phases: oblivious database projection, oblivious interval skyline identification, and oblivious interval skyline retrieval. As OblivTime executes through these phases sequentially with secret-shared inputs and outputs. We can conclude that Query is secure if the simulator for each phase exists. Therefore, we analyze the existence of the simulators in turn.
  - *Simulator for* OblivProject. It is noted that OblivProject consists of four meta operations: evaluation of FSS key, obtaining RSS sharing from 2-out-of-2 additive sharing (or resharing for short), OblivColShuffle, and opening $[\![\mathbf{G}'[N+1]]\!]$ to get $\mathbf{r}'$. Since these operations are invoked sequentially, we analyze the existence of their simulators in turn. The evaluation of FSS keys can be trivially simulated by invoking the simulator of FSS [32]. During resharing, C_1 receives a secret share from C_2 masked with a random value generated by a pseudo-random generator using a seed unknown to C_1. So the security of the pseudo-random generator guarantees that the masked share is indistinguishable from uniformly random message to C_1. Thus, the simulator for resharing exists. Then, in OblivColShuffle, all the messages received are masked with pseudo-random lists and pseudo-random permutation generated with pre-shared seed unknown to C_1. So, the simulator for OblivColShuffle exists. There also exists a simulator for opening $\mathbf{r}'$, as $\mathbf{r}'$ is produced from $\mathbf{r}$ using random permutation, $\mathbf{r}'$ shares the same elements with $\mathbf{r}$ but arranged in random order. So, the simulator can simulate $\mathbf{r}'$ by setting I elements of random position in a zero list to 1, and the security of pseudo-random permutation used in OblivColShuffle guarantees that it is computationally indistinguishable to $\mathbf{r}'$. Therefore, the simulator for OblivProject exists.
  - *Simulator for* OblivIdentify. We identify that OblivIdentify consists of OblivFetch, OblivFilter, secEqual$^{\mathsf{R}\to 2}$, localShare, and secure reconstruction of isStop, we thus analyze the existence of the simulator for each of the components in turn.
    * *Simulator for* OblivFetch. OblivFetch consists of secCmp and multiplication between RSS-shared values. We only need to show that the simulator for secCmp exists given a simulator for RSS multiplication has been shown in [20]. secCmp can be further decomposed into four steps: subtracting the two RSS shares to get $[\![c]\!] = [\![a]\!] - [\![b]\!]$, opening masked $c+r$, evaluating the DIF key at $c+r$, and resharing the 2-out-of-2 additive evaluation result. For the first step, the subtraction can be trivially simulated with an RSS simulator. Then, in the second step, C_1 receives $\langle c_3 \rangle + \langle r \rangle_2^2$ from C_2. As the message is masked with $\langle r \rangle_2^2$ that is uniformly random in the view of C_1, the simulator for the second step also exists. After that, the simulator for evaluating DIF keys also exists by invoking the FSS simulator [32]. Finally, the simulator for resharing exists as shown above. Therefore, the simulator for secCmp exists, which further implies the existence of the simulator for OblivFetch.
    * *Simulator for* OblivFilter. The OblivFilter is composed of secEqual, secCmp$^{\mathsf{R}\to 2}$, secEqual$^{2\to\mathsf{R}}$, and multiplication between RSS-shared values. Similar to secCmp, the components secEqual, secCmp$^{\mathsf{R}\to 2}$, and secEqual$^{2\to\mathsf{R}}$ adhere to the preprocessing FSS model [32]. So the existence of simulators for these components can be proved under a similar framework. Additionally, a simulator for the multiplication of RSS shares has already been established above. Therefore, the simulator for OblivFilter exists.
    * *Simulator for other components.* We continue to discuss the simulators for secEqual$^{\mathsf{R}\to 2}$, localShare, and reconstruction of isStop. The simulator for secEqual$^{\mathsf{R}\to 2}$ exists as it also follows the preprocessing FSS model [32]. The simulator for localShare also trivially exists as no interaction between cloud servers is involved. The simulator for reconstructing isStop can be constructed as follows: given the number of skyline time series M, for round k, the simulator can determine that isStop $= 0$ if $k \le M$ and isStop $= 1$ if $k > M$. So, the simulator for the reconstruction process exists. We conclude that the simulator for OblivIdentify exists.
  - *Simulator for* OblivRetrieve. We note that secEqual is the only component in OblivRetrieve involving interaction between cloud servers. The existence of the simulator for secEqual has been established above, thereby confirming the existence of the simulator for OblivRetrieve.

The case of C_3 being corrupted. It is evident that simulators exist for both Init and Append, as C_3 and C_1 have symmetric roles in these components, and the proof provided earlier for C_1 is applicable here as well. We only focus on the simulator for Query. Following a similar approach, we decompose Query into three phases. By leveraging the existence of simulators for each of these phases, we establish the existence of the simulator for the Query operation. The existence of simulators for these phases are proven as follows:

- *Simulator for* OblivProject. Since the roles of C_3 and C_1 are symmetric in all the components, namely resharing, OblivColShuffle, and opening $[\![\mathbf{G}'[N+1]]\!]$ to get $\mathbf{r}'$, the simulators for these components also exist for C_3. Conse-

quently, the simulator for OblivProject exists.

- *Simulator for* OblivIdentify. OblivIdentify is composed of OblivFetch, OblivFilter, secEqual, and localShare, and secure reconstruction of isStop. We demonstrate the existence of simulators for these components as follows:
  - *Simulator for* OblivFetch. OblivFetch consists of secCmp and multiplication between RSS-shared values. Since the simulation of multiplication between RSS-shared values is straightforward using an RSS simulator, we will focus on secCmp. Within secCmp, C_3 is involved in generating FSS keys, obtaining offset shares, and participating in resharing. The generation of FSS keys and offset sharing can be trivially simulated as C_3 does not receive anything. During resharing, the roles of C_1 and C_3 are symmetric; thus, the simulator for C_3 also exists for the same reasons as for C_1. Consequently, we can conclude that a simulator for OblivFetch exists.
  - *Simulator for* OblivFilter. We identify that OblivFilter consists of secEqual, $\text{secCmp}^{\text{R}\rightarrow 2}$, $\text{secEqual}^{2\rightarrow\text{R}}$, and RSS multiplication. The existence of simulators for RSS multiplication has been shown above. As for secEqual, $\text{secCmp}^{\text{R}\rightarrow 2}$, and $\text{secEqual}^{2\rightarrow\text{R}}$, they all follow the similar preprocessing FSS model as secCmp, therefore the simulators for these components can be similarly constructed. So, the simulators for all components of OblivFilter exist, and consequently, the simulator for OblivFilter also exists.
  - *Simulator for other components.* We now address $\text{secEqual}^{\text{R}\rightarrow 2}$, localShare, and secure opening of isStop. The existence of a simulator for $\text{secEqual}^{\text{R}\rightarrow 2}$ can be proved by following the same framework as secCmp, and the simulator for localShare trivially exists as it requires no interaction between cloud servers. Therefore, we solely focus on the secure reconstruction of isStop. Although the share $\langle \text{isStop} \rangle_i^2$ held by $C_i, i \in \{1, 2\}$ is obtained using the FSS keys and offset shares generated by C_3, we note that the messages received by C_3 (i.e., $\langle \text{isStop} \rangle_1^2 + \alpha$ and $\langle \text{isStop} \rangle_2^2 - \alpha$) are masked with a pseudo-random value α, which is generated using a pre-shared seed unknown to C_3. So, from C_3's perspective, these messages appear uniformly random. Additionally, the simulator can deduce isStop for round k leveraging the number of skyline time series M. Consequently, C_3 cannot distinguish messages sent by C_1, C_2 in the real world from the 2-out-of-2 shares of isStop generated by the simulator. So, a simulator for the secure reconstruction of isStop exists. As the simulators exist for all components of OblivIdentify, the simulator for OblivIdentify also exists.
- *Simulator for* OblivRetrieve. As secEqual is the only component in OblivRetrieve that requires communication between cloud servers, the existence of a simulator for secEqual implies the existence of a simulator for OblivRetrieve. As we have shown the existence of a simulator for secEqual, we conclude that a simulator for OblivRetrieve also exists.

□

Remark. In the literature, various attacks have been proposed against encrypted databases. Most of them mainly exploit the access pattern leakage [39], [40] and the search pattern leakage [41]. In addition, some recent attacks have also exploited the volume pattern leakage [42]–[46]. We emphasize that OblivTime does not reveal search access patterns (see the leakage profile of OblivTime as described above). Therefore, OblivTime is immune to the attacks exploiting the access pattern and/or search pattern. While the current design of OblivTime does not particularly hide the volume pattern, we point out that the existing attacks exploiting this leakage [42]–[46] are not applicable to OblivTime. To start with, it is noted that the attacks that solely exploit the volume pattern leakage have been demonstrated only on datasets with very limited data domain size (e.g., the data domain of dataset in [43] has a size of 91). In contrast, our system works with large data domains (e.g., with a size of 2^{64} as shown in our experiments.) There are some recent attacks [45], [46] exploiting the volume pattern leakage while incorporating active database injection to make the attack more practical. These attacks rely on strong assumptions that are not applicable to our context, as we assume a passive adversary in our threat model. We note that orthogonal padding techniques [47], [48] can be integrated in OblivTime to effectively mitigate the volume pattern leakage, at the cost of higher computational and communication overhead.

9 Practical Optimization

We observe that the overall design of OblivTime lends itself well to parallelization, allowing us to leverage GPUs for concurrent processing and thereby achieve a performance boost. In particular, the GPU architecture is finely tuned for efficiently carrying out numerous simple computations on blocks of values. This means that element-wise operations of vectors can be accelerated on GPU [49]. As the main operations in OblivTime involve the evaluation of FSS keys on encrypted vectors, it is parallelizable because each FSS evaluation is independent, allowing us to allocate them to independent GPU cores for parallel processing. Next, we elaborate on how OblivTime can be optimized to harness the GPU architecture and achieve a performance boost.

- GPU-based instantiation of OblivProject: We note that the cost of OblivProject is dominated by the evaluation of query token (qk_1, qk_2) on the public timestamps, i.e., line 3 of Algorithm 2. Hence, we implement an FSS kernel that enables FSS functionalities on GPU cores and utilize it to compute the evaluation result in parallel. At the beginning, cloud server $C_i, i \in \{1, 2\}$ invokes function cudaMemCpy to copy the FSS key qk_i and sliding window $[t_a : t_c]$ from the CPU to the GPU. After that, C_i allocates each evaluation of FSS key qk_i at each timestamp $t \in [t_a : t_c]$ to independent GPU cores. The evaluation of the FSS key outputs the secret shares $\langle \mathbf{r}[1] \rangle_i^2, \cdots, \langle \mathbf{r}[W] \rangle_i^2$. Finally, the GPU returns the secret shares back to the CPU for the subsequent computations.
- GPU-based instantiation of OblivFetch: We note that OblivFetch involves obliviously fetching the maximum value from the encrypted sum $[\![\mathbf{s}^{(k)}]\!]$. Therefore, we incorporate tournament sort [50] which allows parallel computing and reduces the rounds of communication needed to cut down query latency. Specifically, in tournament sort, the "winners" of each round can be compared in parallel. For example, to find the

maximum value of vector $[\![\mathbf{v}[1:4]]\!]$, $C_{\{1,2,3\}}$ will need to find $\max([\![\mathbf{v}[1]]\!],[\![\mathbf{v}[2]]\!])$ and $\max([\![\mathbf{v}[3]]\!],[\![\mathbf{v}[4]]\!])$ in the first round where $\max([\![\mathbf{v}[1]]\!],[\![\mathbf{v}[2]]\!])$ and $\max([\![\mathbf{v}[3]]\!],[\![\mathbf{v}[4]]\!])$ can be packed and compared simultaneously leveraging GPU.

- GPU-based instantiation of OblivFilter: We observe that the primary cost of OblivFilter is the comparison between the ID $[\![\mathbf{id}[i]]\!]$ and time-series values $[\![\mathbf{P}[i]]\!]$ of each row with those of the fetched skyline row. To boost the performance of OblivFilter, we leverage parallelism both within and across time series. Since the filtering processes for time series are independent, we exploit parallelism across different time series by processing them on independent GPU cores. Additionally, within each time series, the comparisons of $I+1$ corresponding values (lines 4–7 in Algorithm 7) can also be computed in parallel.
- GPU-based instantiation of OblivRetrieve: We note that OblivRetrieve retrieves the complete time series from the original database, and there is also parallelism across and within skyline time series that can be utilized to enhance the efficiency of OblivRetrieve. Different skyline time series can be retrieved independently and in parallel trivially, while the parallelism within each skyline time series can be optimized in the number of DPF keys consumed and the message sent. To obliviously flag the skyline time series with ID $[\![id_*]\!]$ (lines 3–5 in Algorithm 9), instead of invoking secEqual independently on GPU cores N times, we reduce the cost to a pair of DPF keys and sending 2 messages by leveraging the fact that the comparisons are between the same $[\![id_*]\!]$ and every known $i \in [1:N]$. Holding $(k_1^r, \langle r\rangle_1^2, \langle id_*\rangle_1, \langle id_*\rangle_2)$ and $(k_2^r, \langle r\rangle_2^2, \langle id_*\rangle_2, \langle id_*\rangle_3)$ respectively, C_1 and C_2 exchange the masked secret share $\langle id_*\rangle_1 + \langle r\rangle_1^2$ and $\langle id_*\rangle_3 + \langle r\rangle_2^2$ with each other to reconstruct $id_* + r$. $C_j, j \in \{1,2\}$ then evaluates k_j^r in parallel on $id_* + r - i$ for $i \in [1:N]$ to get $[\![\mathbf{e}]\!]^2$. Finally, $C_{\{1,2,3\}}$ obtain $[\![\mathbf{e}]\!]$ in RSS through method described in Section 5.2.

10 Experiments

10.1 Setup

We implement the protocols in OblivTime using CUDA (for generating and evaluating FSS keys) and Python (for all other parts of the protocol). We instantiate the PRFs using AES. The experiments are conducted on a machine equipped with Intel Xeon 8352Y and having 32 physical cores, 64 GB memory, and 3 Nvidia RTX A6000 graphics cards. We simulate all three cloud servers using one process. To model the cost of transferring data between cloud servers in different trust domains, we follow [33] in 3PC setting and set the network bandwidth to be 1.25 GB/s with an average latency of 0.2 ms. We encode the test datasets with 64-bit fix-point representation.

Similar to previous works [10], [11], we use two real-world datasets and one synthesized dataset. Specifically, we follow [11] and adopt greenhouse gas observing network dataset[1] (or GHG dataset for short) and electricity load diagrams 2011–2014 dataset[2] (or LD dataset for short) as the real-world datasets. We also follow the method in [10] and generate a synthesized dataset where the mean μ_i of the i-th time series is sampled from a standard normal distribution $\mathcal{N}(0,1)$ and the values of the i-th time series are sampled from $\mathcal{N}(\mu_i, 0.1)$. Note for the LD dataset, we filter out ineffective timestamps where time-series values are missing (an ineffective timestamp is the one at which two consecutive time series have zero values). This results in $11,172$ effective timestamps. We summarize the size information of the tested datasets in Table 1. The results reported in our experiments are an average over 10 interval skyline queries unless otherwise noted. For each query, we first shuffle the complete dataset and extract the first N time series and W timestamps from the shuffled dataset to be the test data in the database and set the query interval to $[1:I]$.

1. https://doi.org/10.24432/C5JK5M
2. https://doi.org/10.24432/C58C86

TABLE 1
The Size Information of the Tested Datasets

Dataset name	N	W
GHG	2921	327
LD	370	11172
Synthesized	210000	500

TABLE 2
Query Latency and Communication Costs on Real-world Datasets with Varying Numbers of Time Series

	N	Latency (s)		Comm. (MB)		M
		Overall	Per M	Overall	Per M	
GHG	500	8.1	0.2	162	3	48.9
GHG	1000	17.5	0.3	388	7	58.7
GHG	1500	34.0	0.4	749	10	76.2
GHG	2000	55.7	0.6	1210	13	92.8
GHG	2500	82.1	0.7	1788	16	110.1
LD	50	1.0	0.3	30	9	3.4
LD	100	1.5	0.4	61	17	3.6
LD	150	2.1	0.6	93	24	3.8
LD	200	2.2	0.8	109	42	2.6
LD	250	3.1	1.0	142	47	3.0
LD	300	4.0	1.2	176	53	3.3
LD	350	5.0	1.4	214	58	3.7

10.2 Evaluation on Performance

We first test the performance of OblivTime on two real-world datasets: GHG and LD. We set $N=1000$, $I=100$, $W=300$ for GHG and $N=100$, $I=1000$, $W=11000$ for LD unless otherwise noted. We measure the running time and total amount of data transmitted among $C_{\{1,2,3\}}$ online to produce the query result.

Impact of time series number on performance. Table 2 shows the query latency and communication cost on GHG and LD for different N (time series number), where we also provide the average number of skyline time series M. It is observed that the per-skyline query latency and communication cost grow linearly with N, especially for datasets with greater M. Fig. 3 illustrates the breakdowns in query latency and communication cost on the real-world datasets with different N. The majority of the query latency is due to OblivIdentify. Regarding communication overhead, it primarily stems from OblivIdentify on GHG while both OblivProject and OblivIdentify contribute significantly on LD. This discrepancy is attributed to the higher number of

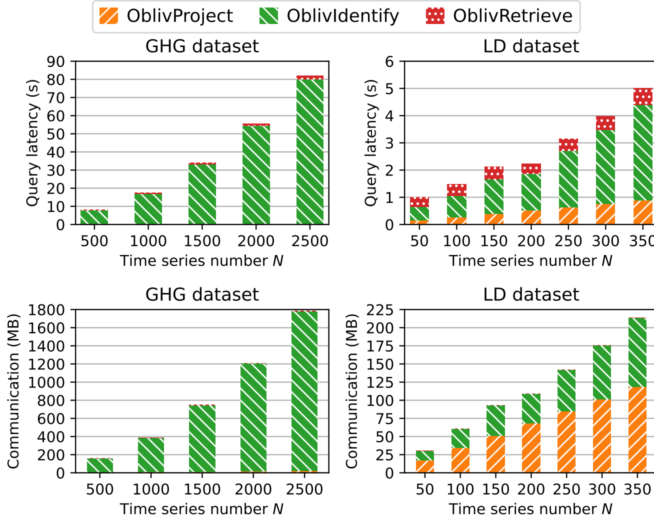


Fig. 3. Breakdown of query latency and communication costs on real-world datasets with different numbers of time series.

TABLE 3
Query Latency and Communication Costs on Real-world Datasets with Varying Query Time Interval Lengths

	I	Latency (s)		Comm. (MB)		M
		Overall	Per M	Overall	Per M	
GHG	50	6.1	0.2	123	4	33.7
	100	17.6	0.3	388	7	58.7
	150	39.4	0.4	855	10	89.1
	200	68.0	0.6	1466	13	116.1
	250	99.7	0.7	2165	16	138.2
	300	160.9	0.9	3450	19	184.6
LD	1000	1.5	0.4	61	17	3.6
	3000	5.0	1.0	143	28	5.2
	5000	9.5	1.5	251	39	6.4
	7000	11.8	1.5	397	51	7.8
	9000	15.2	1.9	501	64	7.8
	11000	25.2	2.7	698	76	9.2

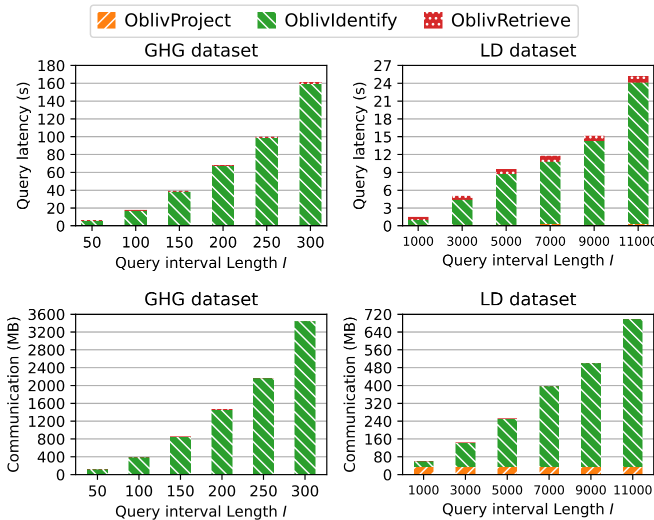


Fig. 4. Breakdown of query latency and communication costs on real-world datasets with varying query time interval lengths.

skyline time series in GHG, which necessitates more rounds of execution for OblivIdentify.

Impact of query interval length on performance. We then show the query latency and communication cost on GHG and LD regarding the query time interval length I. Table 3 displays these results, which indicate a linear growth in per-skyline query latency and communication cost as I increases. Furthermore, Fig. 4 illustrates the breakdowns in query latency and communication cost on real-world datasets with varying I. The impact of increasing I on the query latency and communication cost of OblivProject is minimal. This is because OblivProject operates on the entire database for obliviousness to select the time-series values within the query interval, rendering it unaffected by changes in I.

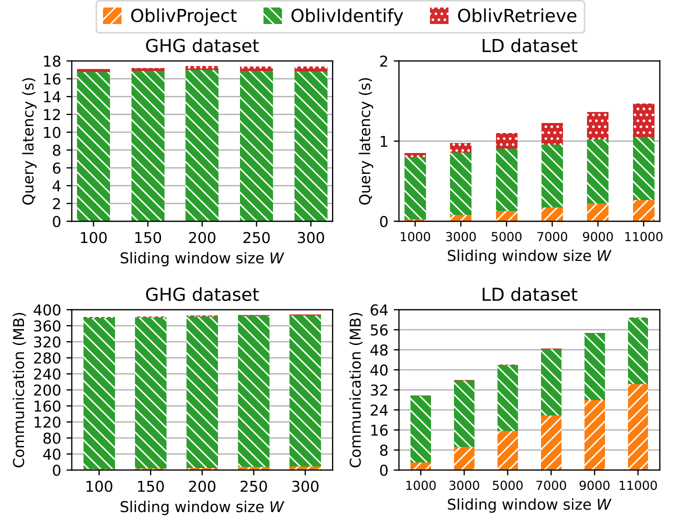


Fig. 5. Breakdown of query latency and communication costs on real-world datasets with varying sliding window sizes.

Impact of sliding window size on performance. In Fig. 5, we show the query latency and communication cost on GHG and LD respectively, with the sliding window size W varying. As shown, increasing W has a relatively small impact on the query latency and communication cost. This is because OblivTime invokes OblivProject only once to select the time-series values within the query interval.

Comparison to state-of-the-art prior work. Given that OblivTime achieves much stronger security guarantees than the state-of-the-art prior work [11], it is indeed unfair to directly compare the performance of OblivTime and [11]. However, for completeness and to demonstrate OblivTime's good efficiency while achieving stronger security, we give performance comparisons here. Note that the protocol in [11] is also tested on the datasets GHG and LD. We compare the query latency and communication cost of OblivTime and their protocol under the aforementioned default setting. According to [11], their results on the query latency and communication cost are about 1×10^4 seconds and 2000 MB respectively for $N = 1000, I = 100$ on GHG; and are about 1000 seconds and 1000 MB with $N = 100, I = 1000$ on LD[3]. In contrast, in OblivTime the query latency is about 17.6 seconds and the communication cost is about 388 MB on GHG, and the query latency is about 1.5 seconds and the communication cost is about 61 MB on LD. Therefore, OblivTime is about 568× better in query latency and 5× better in online communication on GHG, and

3. Results are deduced from the figures in [11] since exact values are not available within text in [11].

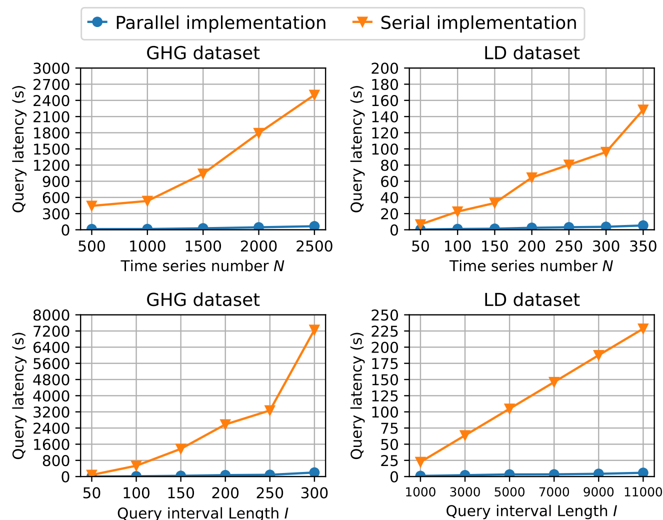


Fig. 6. Query latency of different implementations with varying N or I on real-world datasets.

TABLE 4
Query Latency and Communication Cost on the Synthesized Dataset with Different Numbers of Time Series

	N	Latency (s)		Comm. (MB)		M
		Overall	Per M	Overall	Per M	
Synthesized	$1{\times}10^4$	4.2	1.2	206	59	3.5
	$5{\times}10^4$	13.0	7.2	800	444	1.8
	$9{\times}10^4$	21.1	14.1	1366	911	1.5
	$1.3{\times}10^5$	38.5	18.3	2185	1040	2.1
	$1.7{\times}10^5$	60.0	23.1	3087	1187	2.6
	$2.1{\times}10^5$	87.5	28.2	4098	1322	3.1

about 666× better in query latency and 16× better in online communication on LD.

10.3 Evaluation on GPU-based Practical Optimization

We now showcase the performance advantage rendered by our practical optimization leveraging GPU-based protocol instantiations, as compared with a serial implementation. Fig. 6 shows the comparison in the query latency, as we vary the number of times series N and query interval length I respectively. It is evident that the GPU-based parallel implementation of OblivTime dramatically reduces the query latency. On GHG, we observe about 38× speedup for $N = 2500$ and about 36× speedup for $I = 300$. On LD, we observe about 28× speedup for $N = 350$ and about 39× speedup for $I = 11000$.

10.4 Evaluation on Scalability

To demonstrate the scalability of OblivTime, we also examine the performance on a large-scale synthesized dataset. The results on the query latency and online communication cost with varying numbers of time series are given in Table 4. We set $I = 40$, and $W = 300$ when querying on the synthesized dataset. It is observed that $C_{\{1,2,3\}}$ can finish an interval skyline query using about 87.5 seconds and 4098 MB communication on a database with 2.1×10^5 time series.

11 Conclusion

In this paper, we design, implement, and evaluate OblivTime, a system framework for cloud-based privacy-preserving interval skyline query processing over encrypted time-series data, with stronger security guarantee and lower query latency over prior art. OblivTime employs advanced lightweight cryptographic techniques, such as replicated secret sharing, function secret sharing, and secure shuffling. It also leverages the data parallelism of high-dimensional time-series data to minimize query latency. Extensive experiments show that OblivTime reduces the query latency by 568× and 666× on two real world datasets.

Acknowledgments

This work was supported in part by the Guangdong Basic and Applied Basic Research Foundation under Grant No. 2024A1515012299. Part of this work was done while Yifeng Zheng was with Harbin Institute of Technology, Shenzhen.

References

[1] J. Lin, E. Keogh, A. Fu, and H. Van Herle, "Approximations to magic: finding unusual medical time series," in *Proc. of IEEE CBMS*, 2005, pp. 329–334.
[2] InfluxData, "HealthQ," https://www.influxdata.com/customer/healthq/, 2022, [Online; Accessed 30-Oct-2023].
[3] S. Aminikhanghahi, T. Wang, and D. J. Cook, "Real-Time Change Point Detection with Application to Smart Home Time Series Data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 5, pp. 1010–1023, 2019.
[4] F. Eichinger, P. Efros, S. Karnouskos, and K. Böhm, "A time-series compression technique and its application to the smart grid," *The VLDB Journal*, vol. 24, no. 2, pp. 193–218, 2015.
[5] Amazon Web Services, "Amazon Timestream," https://aws.amazon.com/timestream/, 2023, [Online; Accessed 30-Oct-2023].
[6] InfluxData, "InfluxDB Cloud," https://www.influxdata.com/products/influxdb-cloud/, 2023, [Online; Accessed 30-Oct-2023].
[7] Timescale, "Timescale," https://www.timescale.com/, 2023, [Online; Accessed 30-Oct-2023].
[8] Toyota Motor Corporation, "Apology and Notice Concerning Newly Discovered Potential Data Leakage of Customer Information Due to Cloud Settings," https://global.toyota/en/newsroom/corporate/39241625.html, 2023, [Online; Accessed 30-Oct-2023].
[9] J. Greig, "1,000 GB of local government data exposed by Massachusetts software company," https://www.zdnet.com/article/1000-gb-of-local-government-data-exposed-by-massachusetts-software-company/, 2021, [Online; Accessed 30-Oct-2023].
[10] B. Jiang and J. Pei, "Online Interval Skyline Queries on Time Series," in *Proc. of IEEE ICDE*, 2009, pp. 1036–1047.
[11] S. Zhang, S. Ray, R. Lu, Y. Zheng, Y. Guan, and J. Shao, "Towards Efficient and Privacy-Preserving Interval Skyline Queries Over Time Series Data," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 2, pp. 1348–1363, 2023.
[12] J. Liu, J. Yang, L. Xiong, and J. Pei, "Secure Skyline Queries on Cloud Platform," in *Proc. of IEEE ICDE*, 2017, pp. 633–644.
[13] X. Ding, Z. Wang, P. Zhou, K. R. Choo, and H. Jin, "Efficient and Privacy-Preserving Multi-Party Skyline Queries Over Encrypted Data," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 4589–4604, 2021.
[14] S. Bothe, A. Cuzzocrea, P. Karras, and A. Vlachou, "Skyline Query Processing over Encrypted Data: An Attribute-Order-Preserving-Free Approach," in *Proc. of ACM PSBD*, 2014, pp. 37–43.
[15] W. Wang, H. Li, Y. Peng, S. S. Bhowmick, P. Chen, X. Chen, and J. Cui, "SCALE: An Efficient Framework for Secure Dynamic Skyline Query Processing in the Cloud," in *Proc. of DASFAA*, 2020, pp. 288–305.
[16] S. Zeighami, G. Ghinita, and C. Shahabi, "Secure Dynamic Skyline Queries Using Result Materialization," in *Proc. of IEEE ICDE*, 2021, pp. 157–168.
[17] Y. Zheng, W. Wang, S. Wang, X. Jia, H. Huang, and C. Wang, "SecSkyline: Fast Privacy-Preserving Skyline Queries Over Encrypted Cloud Databases," *IEEE Transactions on Knowledge and Data Engineering*, 2022.

[18] E. Dauterman, M. Rathee, R. A. Popa, and I. Stoica, "Waldo: A private time-series database from function secret sharing," in *Proc. of IEEE S&P*, 2022.

[19] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart, "Leakage-Abuse Attacks against Order-Revealing Encryption," in *Proc. of IEEE SP*, 2017, pp. 655–672.

[20] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara, "High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority," in *Proc. of ACM CCS*, 2016, pp. 805–817.

[21] E. Boyle, N. Chandran, N. Gilboa, D. Gupta, Y. Ishai, N. Kumar, and M. Rathee, "Function Secret Sharing for Mixed-Mode and Fixed-Point Secure Computation," in *Proc. of EUROCRYPT*, 2021, pp. 871–900.

[22] T. Araki, J. Furukawa, K. Ohara, B. Pinkas, H. Rosemarin, and H. Tsuchida, "Secure Graph Analysis at Scale," in *Proc. of ACM CCS*, 2021, pp. 610–629.

[23] S. Zhang, S. Ray, R. Lu, Y. Zheng, Y. Guan, and J. Shao, "Achieving Efficient and Privacy-Preserving Dynamic Skyline Query in Online Medical Diagnosis," *IEEE Internet of Things Journal*, vol. 9, no. 12, pp. 9973–9986, 2022.

[24] X. Liu, K. R. Choo, R. H. Deng, Y. Yang, and Y. Zhang, "PUSC: Privacy-Preserving User-Centric Skyline Computation Over Multiple Encrypted Domains," in *Proc. of IEEE TrustCom/BigDataSE*, 2018, pp. 958–963.

[25] S. Zhang, S. Ray, R. Lu, Y. Zheng, Y. Guan, and J. Shao, "Towards Efficient and Privacy-Preserving User-Defined Skyline Query Over Single Cloud," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 2, pp. 1319–1334, 2023.

[26] L. Burkhalter, A. Hithnawi, A. Viand, H. Shafagh, and S. Ratnasamy, "TimeCrypt: Encrypted data stream processing at scale with cryptographic access control," in *Proc. of NSDI*, 2020.

[27] L. Burkhalter, N. Küchler, A. Viand, H. Shafagh, and A. Hithnawi, "Zeph: Cryptographic enforcement of end-to-end data privacy," in *Proc. of OSDI*, 2021.

[28] M. Faisal, J. Zhang, J. Liagouris, V. Kalavri, and M. Varia, "TVA: A multi-party computation system for secure and expressive time series analytics," in *Proc. of USENIX Security*, 2023.

[29] J. D. Hamilton, *Time Series Analysis*. Princeton University Press, 2020.

[30] E. Boyle, N. Gilboa, and Y. Ishai, "Function Secret Sharing," in *Proc. of EUROCRYPT*, 2015, pp. 337–367.

[31] ——, "Function Secret Sharing: Improvements and Extensions," in *Proc. of ACM CCS*, ser. CCS '16, 2016, pp. 1292–1303.

[32] ——, "Secure Computation with Preprocessing via Function Secret Sharing," in *Proc. of TCC*, 2019, pp. 341–371.

[33] S. Tan, B. Knott, Y. Tian, and D. J. Wu, "CryptGPU: Fast privacy-preserving machine learning on the GPU," in *Proc. of IEEE S&P*, 2021.

[34] S. Badrinarayanan, S. Das, G. Garimella, S. Raghuraman, and P. Rindal, "Secret-Shared Joins with Multiplicity from Aggregation Trees," in *Proc. of ACM CCS*, 2022, pp. 209–222.

[35] Mozilla Security Blog, "Next steps in privacy-preserving telemetry with prio," https://blog.mozilla.org/security/2019/06/06/next-steps-in-privacy-preserving-telemetry-with-prio/, 2019, [Online; Accessed 7-Jul-2023].

[36] Brave Privacy Team, "Star: Brave's new system for privacy-preserving data collection," https://brave.com/privacy-updates/19-star/, 2022, [Online; Accessed 7-Jul-2023].

[37] C. Cai, Y. Zang, C. Wang, X. Jia, and Q. Wang, "Vizard: A Metadata-hiding Data Analytic System with End-to-End Policy Controls," in *Proc. of ACM CCS*, 2022, pp. 441–454.

[38] Y. Lindell, "How to Simulate It – A Tutorial on the Simulation Proof Technique," in *Tutorials on the Foundations of Cryptography*. Springer International Publishing, 2017, pp. 277–346.

[39] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation," in *Proc. of NDSS*, 2012.

[40] M.-S. Lacharité, B. Minaud, and K. G. Paterson, "Improved Reconstruction Attacks on Encrypted Data Using Range Query Leakage," in *Proc. of IEEE S&P*, 2018.

[41] S. Oya and F. Kerschbaum, "Hiding the Access Pattern is Not Enough: Exploiting Search Pattern Leakage in Searchable Encryption," in *Proc. of USENIX Security*, 2021.

[42] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill, "Generic Attacks on Secure Outsourced Databases," in *Proc. of ACM CCS*, 2016.

[43] P. Grubbs, M.-S. Lacharite, B. Minaud, and K. G. Paterson, "Pump up the Volume: Practical Database Reconstruction from Volume Leakage on Range Queries," in *Proc. of ACM CCS*, 2018.

[44] Z. Gui, O. Johnson, and B. Warinschi, "Encrypted Databases: New Volume Attacks against Range Queries," in *Proc. of ACM CCS*, 2019.

[45] R. Poddar, S. Wang, J. Lu, and R. A. Popa, "Practical Volume-Based Attacks on Encrypted Databases," in *Proc. of EuroS&P*, 2020.

[46] X. Zhang, W. Wang, P. Xu, L. T. Yang, and K. Liang, "High Recovery with Fewer Injections: Practical Binary Volumetric Injection Attacks against Dynamic Searchable Encryption," in *Proc. of USENIX Security*, 2023.

[47] R. Bost and P. Fouque, "Thwarting leakage abuse attacks against searchable encryption - A formal approach and applications to database padding," *IACR Cryptol. ePrint Arch.*, p. 1060, 2017.

[48] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre, "{SEAL}: Attack Mitigation for Encrypted Databases via Adjustable Leakage," in *Proc. of USENIX Security*, 2020.

[49] S. Pandey, Z. Wang, S. Zhong, C. Tian, B. Zheng, X. Li, L. Li, A. Hoisie, C. Ding, D. Li *et al.*, "Trust: Triangle counting reloaded on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 11, pp. 2646–2660, 2021.

[50] R. Cole, "Parallel Merge Sort," *SIAM Journal on Computing*, vol. 17, no. 4, pp. 770–785, 1988.

Huajie Ouyang received the BE degree in computer science and technology from Harbin Institute of Technology, Shenzhen, China, in 2022. He is currently working toward the ME degree in the School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China. His research interests include cloud computing security and secure multi-party computation.

Yifeng Zheng is an Assistant Professor with the Department of Electrical and Electronic Engineering, The Hong Kong Polytechnic University, Hong Kong. He received his PhD degree in computer science from the City University of Hong Kong, Hong Kong in 2019. From 2019 to 2020, he worked as a Postdotoral Fellow with the Commonwealth Scientific and Industrial Research Organization, Australia. His research works have been published in prestigious journals (such as TIFS, TDSC, and TSC) and conferences (such as USENIX Security and CCS). His current research interests are focused on machine learning security and privacy, secure networked systems, and secure outsourced services in cloud computing.

Songlei Wang is an Assistant Professor (under Hundred Talents Program) in the National Engineering Laboratory for Big Data System Computing Technology, Shenzhen University, China. He received his Ph.D. in Computer Science and Technology from the Harbin Institute of Technology, Shenzhen, in 2024. His research works have been published in prestigious conferences and journals, such as USENIX Security, TIFS, and TKDE. His research interests include cloud security and secure machine learning.

Zhongyun Hua received the B.S. degree from Chongqing University, Chongqing, China, in 2011, and the M.S. and Ph.D. degrees from University of Macau, Macau, China, in 2013 and 2016, respectively, all in software engineering. He is currently a professor with the School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, Shenzhen, China. His research interests include chaotic system and information security.