Concealer: SGX-based Secure, Volume Hiding, and Verifiable Processing of Spatial Time-Series Datasets

Peeyush Gupta, Sharad Mehrotra, Shantanu Sharma, Nalini Venkatasubramanian, and Guoxi Wang
University of California, Irvine, USA.
sharad@ics.uci.edu, shantanu.sharma@uci.edu

ABSTRACT

This paper proposes a system, entitled Concealer that allows sharing time-varying spatial data (*e.g.*, as produced by sensors) in encrypted form to an untrusted third-party service provider to provide location-based applications (involving aggregation queries over selected regions over time windows) to users. Concealer exploits carefully selected encryption techniques to use indexes supported by database systems and combines ways to add fake tuples in order to realize an efficient system that protects against leakage based on output-size. Thus, the design of Concealer overcomes two limitations of existing symmetric searchable encryption (SSE) techniques: (*i*) it avoids the need of specialized data structures that limit usability/practicality of SSE in large scale deployments, and (*ii*) it avoids information leakages based on the output-size, which may leak data distributions. Experimental results validate the efficiency of the proposed algorithms over a spatial time-series dataset (collected from a smart space) and TPC-H datasets, each of 136 Million rows, the size of which prior approaches have not scaled to.

1 INTRODUCTION

We consider the problem wherein trusted data producers ($\mathcal{DP}$) share users' spatial time-series data in the encrypted form with untrusted service providers ($\mathcal{SP}$) to empower $\mathcal{SP}$ to build value-added applications for users. Examples include a cellular company sharing data about the cell tower a user's mobile phone is connected to, or an organization/university providing WiFi connectivity data about the access point a user's device is connected to, for applications such as building dynamic occupancy maps [1]. We classify applications supported by $\mathcal{SP}$ using user's data into two classes, as follows:

(1) **Aggregate Applications** that aggregate data of multiple users to build novel applications. Examples include occupancy of different regions, heat maps, and count of distinct visitors to a given region over a period of time. Such applications are already supported by several service providers, *e.g.*, Google Maps supports information about busy-status and wait times at stores such as restaurants and shopping malls.

(2) **Individualized Applications** that allow users to ask queries based on their own past movements, *e.g.*, locations a person spent the most time during a given time interval, finding the number of people came in contact with, and/or other aggregate operations on user's data. Such applications can be useful for several contexts including exposure tracing in the context of infectious diseases [37].

Implementing applications at the (untrusted) $\mathcal{SP}$ requires: (*i*) $\mathcal{DP}$ to appropriately encrypt data prior to sharing with $\mathcal{SP}$, (*ii*) $\mathcal{SP}$ to be able to execute queries on behalf of the user over the encrypted data, and (*iii*) the user to be able to decrypt the encrypted answers returned by *SP*. Realizing such a data-sharing architecture leads to the following three requirements, (of which the first two are relatively straightforward, while the third requires a careful design of a new cryptographic technique that this paper focuses on):

R1: Query formulation by the user. Given that data is encrypted by $\mathcal{DP}$ and is hosted at $\mathcal{SP}$, the user needs to formulate the query to enable $\mathcal{SP}$ to execute it over encrypted data. The users can formulate an appropriate encrypted query, if they know the key used for encryption by $\mathcal{DP}$. However, $\mathcal{DP}$ cannot share the key with users to prevent them from decrypting the entire dataset. A trivial way to overcome this problem is to involve $\mathcal{DP}$ in processing queries. Particularly, a user can submit queries to $\mathcal{DP}$ that converts the query into appropriate trapdoors to be executed on encrypted data at $\mathcal{SP}$, fetches the partial results from $\mathcal{SP}$, and processes the fetched rows, before producing the final answer to users. Such an architecture incurs significant overhead at $\mathcal{DP}$ and requires $\mathcal{DP}$ to mediate each user query, (pushing $\mathcal{DP}$ to act as a surrogate $\mathcal{SP}$). Thus, ***the first requirement is how users can formulate appropriate encrypted queries without involving $\mathcal{DP}$ in query processing.***

We can overcome this requirement *trivially* by using secure hardware (*e.g.*, Intel Software Guard eXtensions[1] (SGX) [9]) at $\mathcal{SP}$ that works as a trusted agent of $\mathcal{DP}$. SGX receives queries encrypted using the public key of SGX (which we assume to known to all) from users, decrypts the query, converts the query into appropriate secure trapdoors, and provides the answer.

R2: Preventing $\mathcal{SP}$ from impersonating a user. Since we do not wish to involve $\mathcal{DP}$ during query processing, all users ask queries directly to $\mathcal{SP}$. Such query representations should not empower $\mathcal{SP}$ to mimic/masquerade as a legitimate user to gain access to the cleartext data from the answers to the query. Thus, ***the second requirement is how will the system prevent $\mathcal{SP}$ to mimic as a user and execute a query.***

We overcome this requirement *trivially* by building a list of *registered users*, who are allowed to execute queries on the encrypted data (after a negotiation between users and $\mathcal{DP}$) at $\mathcal{DP}$ and provides it in encrypted form to $\mathcal{SP}$. The registry contains credential information (*e.g.*, public/private key and authentication information of users) about the users who are interested

This material is based on research sponsored by DARPA under Agreement No. FA8750-16-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. This work is supported by NSF Grants 1527536, 1545071, 2032525, 1952247,1528995, 2008993.

[1] Recent Intel CPUs introduced SGX that allows us to create a small trusted execution environment, called *enclave* that is isolated and protected from the rest of the system. SGX protects computations from the operating system (controlled by the third-party) and from numerous applications/system-level attacks. Unfortunately, existing implementations of SGX are prone to side-channel attacks that exploit one of the microarchitectural components of CPUs, *e.g.*, cache-lines, branch execution, page-table access [20, 39, 40], and power attacks. Nevertheless, systems T-SGX [34] and Sanctum [10] have evolved to overcome such attacks, and it is believed that future versions of SGX will be resilient to those attacks.

Techniques	Frequent and fast insertion	Fast query execution	DBMS supported index	Preventing attacks		
				Data distribution	Output-size	Access-patterns
DET (Always Encrypt [3])	1	1	Yes	No	No	No
NDET (Arx [32] or Always Encrypt [3])	2	2 or 3	Yes	Yes	No	No
Indexable-SSE (PB [22]- or IB [23]-Tree)	4	2	No*	No	No	No
Indexable-SSE with ORAM (Blind Seer [30])	4	4	No*	No	No	Yes
Non-indexable-SSE [11, 35]	2	4	No	No	No	No
SGX system (Opaque [42])	2	3	No	No	No	No
MPC or SS (Jana [4])	4	4	No	Yes	No	Yes
Concealer	1	1	Yes	Yes	Yes	Yes (partial)

Table 1: Comparing different techniques vs Concealer. Note: 1: Very fast, 2: Fast, 3: Slow. 4: Very slow. ∗: Indexable SSEs build their own indexes and their index traversal techniques are not in-built in existing commercial DBMS.

in $\mathcal{SP}$ applications. Thus, before generating any trapdoor by SGX, it first authenticates the user and provides the final answers encrypted using the public key of the user.

R3: Selecting the appropriate encryption technique. Spatial time-series data brings in new challenges (as compared to other datasets) in terms of a large-amount of the dataset and dynamically arriving data. Also, spatial time-series data show new opportunities in terms of limited types of queries (*i.e.*, not involving complex operations such as join and nested queries). Particularly, the data encryption and storage must sustain the data generation rate, *i.e.*, the encryption mechanism must support dynamic insertion without the high overhead. Further, cryptographic query execution time should scale to millions of records. Finally, the system must support strong security properties such that the ciphertext representation and query execution do not reveal information about the data to $\mathcal{SP}$. Note that ciphertext representation leaks data distribution only when deterministic encryption (DET) is used. Query execution leaks information about data due to search- and access-patterns leakages, and volume/output-size leakage. In §1.1, we discuss these leakages and argue that none of the existing cryptographic query processing techniques satisfy all the above requirements. Thus, ***the third requirement is how to design a system that has efficient data encryption and query execution techniques, and not prone to such leakages.***

Concealer. We design, develop, and implement a secure spatial time-series database, entitled Concealer. This paper focuses on how Concealer addresses the above-mentioned third requirement, and below, we briefly discuss the proposed solution to the requirement R3. Concealer is carefully designed to support a high rate of data arrival, and large data sets, but it only supports a limited nature of spatial time-series queries required by the domain of interest. To a degree, Concealer can be considered more of a vertical technology compared to general-purpose horizontal solutions, which as will be discussed in §1.1, lack the ability to support application/data that motivates Concealer.

Concealer, for fast data encryption and minimum cryptographic overheads on each tuple, uses a variant of deterministic encryption that produces secure ciphertext (that does not reveal data distribution) and is fast enough to encrypt tuples (≈37,185 tuples/min). Further, Concealer exploits the index supported by MySQL. Note that we do not use any specialized index (*e.g.*, PB-tree [22] and IB-Tree [23]) and do not require to build the entire index for each insertion at the trusted side. Since Concealer users an index supported by DBMS, it supports efficient query execution. For a point query on 136M rows, Concealer needs at most 0.9s. Thus, our implementation of DET and the use of indexes supported by DBMS satisfy the requirements of fast data insertion and fast query execution.

To address the security challenge during query execution, Concealer (*i*) prevents output-size by fixing the unit of data retrieval of the form of bins, formed over the tuples of a given time period; care is taken to ensure that each bin must be of identical size (by implementing a variant of bin-packing algorithm [8]), and (*ii*) hides *partial* access-patterns, due to retrieving a fixed bin having different tuples corresponding to different sensor readings (with different location/time/other values) for any query corresponding to the element of the bin. That means the adversary observes: which fixed tuples are fetched for a set of queries including the real query posed by the user. However, the adversary cannot find which of the fetched tuples satisfy the user query. Since our focus is on practical system implementation, we relax the complete access-pattern hiding requirements. The exact security offered by Concealer will be discussed in §7.

Since we fetch a bin of several tuples, to filter the useless tuples that do not meet the query predicates, SGX at $\mathcal{SP}$ filters them, (while also hides complete access-patterns inside SGX by performing oblivious operations). Further, to verify the integrity of the data before producing the answer, Concealer provides a *non-mandatory* hash-chain-based verification.

Evaluation. We evaluate Concealer on a real WiFi dataset collected from at UCI. To evaluate its scalability, we executed the algorithms on 136M rows, the size that previous existing cryptographic techniques cannot support. We also compare Concealer against SGX-based Opaque [42]. To the best of our knowledge, there is no system that supports identical security properties (hiding output-size and hiding partial access-patterns, while supporting indexes for efficient processing). Our algorithms can be used to deal with non-time-series datasets also; thus, to evaluate algorithms' practicality, we evaluate aggregation queries on 136M rows LineItem table of TPC-H benchmark.

1.1 Comparison & Advantages of Concealer

We discuss common leakages from cryptographic solutions, argue that they do not satisfy the requirements of fast data insertion, fast query execution, and/or security against information leakages (see Table 1 for a comparison).

Leakages. Cryptographic techniques show following leakages:

(1) *Data distribution leakage from the storage [7]*: allows an adversary to learn the frequency-count of each value by just observing ciphertext. DET reveals such information.

(2) *Search- and access-patterns leakages [7, 16]*: occur during query execution. Search-pattern leakages allow an adversary to learn if and when a query is executed, while access-patterns leakage allows learning which tuples are retrieved (by observing the (physical) address/location of encrypted tuples) to answer a query. Practical techniques (*e.g.*, order-preserving encryption (OPE) [2], DET, symmetric searchable encryption (SSE) [22, 23], and secure hardware-based techniques [3, 33, 42]) reveal access-patterns. In contrast, non-efficient techniques (*e.g.*, secret-sharing [4, 6] or oblivious RAM (ORAM) based techniques) hide access-patterns.

(3) *Volume/output-size leakage [7]*: allows an adversary (having some background knowledge) can deduce the data by simply observing the size of outputs (or the number of qualifying tuples). [7, 16, 26] showed that output-size may also leak data distribution. *Access-patterns revealing techniques implicitly disclose the output-size.* Moreover, the seminal work [19] showed that the output-size revealed even due to access-pattern hiding techniques enables the attacker to reconstruct the dataset. A possible solution is adding fake tuples with the real data, thereby each value has

an identical number of tuples and using indexable SSEs. However, [26] showed that it will be even more expensive than simply scanning the entire database in SGX (or download the data at $\mathcal{DP}$ to execute the query locally). Existing output-size preventing solutions, *e.g.*, Kamara et al. [18] or Patel et al. [31], suffer from one major problem: [18] fetches $\alpha \times max$, $\alpha > 2$, rows, while [31] fetches $2 \times max$ rows with additional secure storage of some rows (which is the function of DB size), where max is the maximum number of rows a value can have. Thus, both [18] and [31] fetch more than the desired rows, *i.e.*, max. Moreover, both [18] and [31] cannot deal with dynamic data.

Existing techniques in terms of data insertion, query execution, and leakages. Existing encrypted search techniques differ in their support for dynamic data, efficient query execution, and offered security properties. For instance, DET supports very efficient insertion and query processing, while its ciphertext data leaks data distribution.

Non-indexable techniques/systems (*e.g.*, SSE [11, 35], secret-sharing (SS) [4, 6], secure hardware-based systems [42]) allow fast data insertions by just encrypting the data, but have inefficient query response time, due to unavailability of an index, and hence, reading the entire data. SS hides search- and access-patterns, while others reveal. Moreover, all such techniques are prone to output-size leakage.

In contrast, indexable techniques/systems (*e.g.*, indexable SSEs (such as PB-Tree [22], IB-Tree [23]) and secure hardware-based index [25]) have faster query execution, but show slow data insertion rate, due to building the entire index at the trusted side for each data insertion; *e.g.*, [30] showed that creating a secure index over 100M rows took more than 1 hour. Moreover, these indexable techniques use ***specialized indexes*** that require specialized encryption and tree traversal protocols that are not supported in the existing standard database systems. This, in turn, limits their usability in dealing with large-scale time-series datasets. All such indexable solutions reveal output-size. While indexable solutions mixed with ORAM (*e.g.*, [30]) hide search- and access-patterns, they are not efficient for query processing (due to several rounds of interaction between the data owner and the server to answer a query). In summary, spatial time-series data adds complexity since (*i*) it can be very large, and (*ii*) arrives dynamically (possibly a high velocity). Existing techniques, as discussed above, are not suitable to support secure data processing over such data.

Advantages of Concealer. (*i*) *Frequent data insert.* We deal with frequent bulk data insertions (which is a requirement of spatial time-series datasets). (*ii*) *Deal with large-size data.* We handle large-sized data with several attributes and large-sized domain efficiently, as our experimental results will show in §8. (*iii*) *Output-size prevention.* While Concealer satisfies the standard security notion (supported by existing SSEs), *i.e.*, indistinguishability under chosen keyword attacks (IND-CKA) [11], it also prevents output-size attacks, unlike IND-CKA. (*iv*) *Oblivious processing in SGX.* As we use the current SGX architecture, suffering from side-channel attacks (*e.g.*, cache-line, branch shadow, and page-fault attack [20, 39, 40]) that enable the adversary to deduce information based on access-patterns in SGX. Thus, we incorporate techniques to deal with these attacks.

1.2 Scoping the Problem

There are other aspects, for them either solutions exist or this paper does not deal with them, as: (*i*) *Key management.* We do not focus on building/improving key infrastructure for public/private keys, as well as, key generation and sharing between

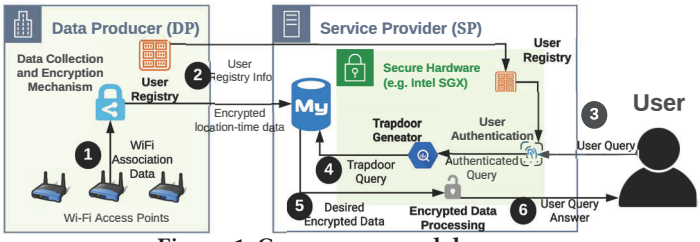

Figure 1: Concealer model.

SGX and $\mathcal{DP}$. Further, changing the keys of encrypted data and re-encrypting the data is out of the scope of this paper, though one may use the recent approach [17] to do so. Also, we do not focus on SGX remote attestation. (*ii*) *Man-in-the-middle (MiM) or replay attacks.* There could be a possibility of MiM and replay attacks on SGX during attestation and query execution. We do not deal with both issues, and techniques [13] can be used to avoid such attacks. (*iii*) *Inference from the number of rows.* Since we send data in epochs, different numbers of tuples in different epochs (*e.g.*, epochs for day vs night time) may reveal information about the user. This can be prevented by sending the same number of rows in each epoch (equals to the maximum rows in any epoch). The current implementation of Concealer does not deal with this issue. (*iv*) *Inference from occupancy count.* Occupancy information mixed with background knowledge reveals the presence/absence of a person at a location (*e.g.*, offices). We do not deal with these inferences, and differential privacy techniques mixed with SGX [41] can be used to deal with such issues.

2 CONCEALER OVERVIEW

This section provides an overview of entities involved in Concealer and its architecture with an overview of algorithms.

2.1 Entities and Assumptions

Concealer consists of the following three entities:

- **Data provider $\mathcal{DP}$:** is a trusted entity that collects user's spatial time-series data as part of its regular operation (*e.g.*, providing cellular service to users). $\mathcal{DP}$ shares such data in encrypted form with service providers $\mathcal{SP}$. $\mathcal{DP}$, also, maintains a *registry*, one per $\mathcal{SP}$, that contains a list of identification information of users, who have registered to use the application provided by the corresponding $\mathcal{SP}$ (*i.e.*, can run queries at that $\mathcal{SP}$). As will be clear, this registry helps to restrict the users to request individualized applications about other users.
- **Service provider $\mathcal{SP}$:** is an untrusted entity that develops location-based applications (as mentioned in §1) over encrypted data. To do so, $\mathcal{SP}$ hosts secure hardware, SGX, that works as a trusted agent of $\mathcal{DP}$.[2] SGX and $\mathcal{DP}$ share a secret key s_k (used for encryption/ decryption of data), and the key s_k is unknown to all other entities.

  An untrusted $\mathcal{SP}$ may try to learn user's data passively by either observing the data retrieved by SGX or exploiting side-channel attacks on SGX during query execution. $\mathcal{SP}$ may, further, learn user's data by actively injecting the fake data into the database and then observing the corresponding ciphertext and query access-patterns. We assume that $\mathcal{SP}$ knows background information, *e.g.*, metadata, the schema of the relation, the number of tuples, and the domain of attributes. However, the adversarial $\mathcal{SP}$ cannot alter anything within the secure hardware and cannot decrypt the data, due to the unavailability of the encryption key. Such assumptions are similar to those considered in the past

[2] The assumption of secure hardware at untrusted third-party machines is consistent with emerging system architectures; *e.g.*, Intel machines are equipped with SGX (see https://newsroom.intel.com/newsroom/wp-content/uploads/sites/11/2017/09/8th-gen-intel-core-product-brief.pdf).

	$\mathcal{L}$	$\mathcal{T}$	$\mathcal{O}$
r_1	l_1	t_1	o_1
r_2	l_1	t_2	o_2
r_3	l_2	t_3	o_2
r_4	l_1	t_4	o_1
r_5	l_2	t_5	o_3
r_6	l_3	t_6	o_2

(a) A relation R in cleartext at $\mathcal{DP}$.

	l_1, l_3	l_2
$t_4\ t_5, t_6$	$cid_1^{\{1,1\}} = \{r_4, r_6\}$	$cid_2^{\{1,2\}} = \{r_5\}$
t_1, t_2, t_3	$cid_1^{\{2,1\}} = \{r_1, r_2\}$	$cid_3^{\{2,2\}} = \{r_3\}$

$cell\_id[] = \{cid_1, cid_2, cid_1, cid_3\}$ $\quad c\_tuple[] = \{4, 1, 1\}$

(b) The grid created at $\mathcal{DP}$ for rows of Table 2a.

	$\mathcal{L}$	$\mathcal{O}$	Tuple	Index($\mathcal{L}, \mathcal{T}$)
r'_1	$\mathcal{E}_k(l_1 \| \| t_1)$	$\mathcal{E}_k(o_1 \| \| t_1)$	$\mathcal{E}_k(l_1 \| \| t_1 \| \| o_1)$	$\mathcal{E}_k(cid_1 \| \| 1)$
r'_7	$\mathcal{E}_{nd}(fake)$	$\mathcal{E}_{nd}(fake)$	$\mathcal{E}_{nd}(fake)$	$\mathcal{E}_k(f \| \| 1)$
r'_2	$\mathcal{E}_k(l_1 \| \| t_2)$	$\mathcal{E}_k(o_2 \| \| t_2)$	$\mathcal{E}_k(l_1 \| \| t_2 \| \| o_2)$	$\mathcal{E}_k(cid_1 \| \| 2)$
r'_3	$\mathcal{E}_k(l_2 \| \| t_3)$	$\mathcal{E}_k(o_2 \| \| t_3)$	$\mathcal{E}_k(l_2 \| \| t_3 \| \| o_2)$	$\mathcal{E}_k(cid_3 \| \| 1)$
r_8	$\mathcal{E}_{nd}(fake)$	$\mathcal{E}_{nd}(fake)$	$\mathcal{E}_{nd}(fake)$	$\mathcal{E}_k(f \| \| 2)$
r'_4	$\mathcal{E}_k(l_1 \| \| t_4)$	$\mathcal{E}_k(o_1 \| \| t_4)$	$\mathcal{E}_k(l_1 \| \| t_4 \| \| o_1)$	$\mathcal{E}_k(cid_1 \| \| 3)$
r'_5	$\mathcal{E}_k(l_2 \| \| t_5)$	$\mathcal{E}_k(o_3 \| \| t_5)$	$\mathcal{E}_k(l_2 \| \| t_5 \| \| o_3)$	$\mathcal{E}_k(cid_2 \| \| 1)$
r'_6	$\mathcal{E}_k(l_3 \| \| t_6)$	$\mathcal{E}_k(o_2 \| \| t_6)$	$\mathcal{E}_k(l_3 \| \| t_6 \| \| o_2)$	$\mathcal{E}_k(cid_1 \| \| 4)$

$Ecell\_id[2,2] = \mathcal{E}_{nd}(\{cid_1, cid_2, cid_1, cid_3\})$

$Ec\_tuple[3] = \mathcal{E}_{nd}(\{4, 1, 1\})$

(c) Encrypted data with encrypted counters at $\mathcal{SP}$.

Table 2: Input time-series relation and output of data encryption algorithm.

work related to SGX-based computation [33, 42], work on attacks based on background knowledge in [19, 27].

- **User or data consumer $\mathcal{U}$:** that uses the services of $\mathcal{DP}$ (such as cellular or WiFi connectivity) and queries to $\mathcal{SP}$. We assume that $\mathcal{U}$ have their public and private keys, which are used to authenticate $\mathcal{U}$ at $\mathcal{SP}$ (via SGX against registry). As mentioned in §1, $\mathcal{U}$ can request both aggregate and individualized queries. While $\mathcal{U}$ is trusted with the data that corresponds to themselves, they are not trusted with data belonging to other users. Finally, we assume that while $\mathcal{U}$ can execute the aggregation queries, they do not collude with $\mathcal{SP}$, *i.e.*, they do not share cleartext results of any query with $\mathcal{SP}$.

2.2 Architecture

Concealer (Figure 1) consists of the following phases:

Phase 0: *Preliminary step: Announcement of $\mathcal{SP}$ by $\mathcal{DP}$.* As a new $\mathcal{SP}$ is added into the system, $\mathcal{DP}$ announces about the $\mathcal{SP}$ to all their users. Only interested users inform to $\mathcal{DP}$ if they want to use $\mathcal{SP}$'s application. Information of such users, their device-ids, and authentication information is stored by $\mathcal{DP}$ in the registry.

Phase 1: *Data upload by $\mathcal{DP}$.* $\mathcal{DP}$ collects spatial time-series data of the form $\langle l_i, t_i, o_i \rangle$ (❶), where l_i is the location, t_i is the time, and o_i is the observed value at l_i and t_i. For instance, in the case of WiFi data, the location may correspond to the region covered by a specific WiFi access-point, and the observation corresponds to a particular device-id connected to that access-point at a given time. $\mathcal{DP}$ encrypts the data using the mechanism given in §3 and provides the encrypted data to $\mathcal{SP}$ (❷) along with encrypted registry and verifiable tags (to verify the data integrity at $\mathcal{SP}$ by SGX).

Concealer considers the data as a relation R with three attributes: $\mathcal{T}$ (time), $\mathcal{L}$ (location), and $\mathcal{O}$ (observation). Table 2a shows an example of the cleartext spatial time-series data, (which will be used in this paper to explain Concealer). In Table 2a, we have added a row-id r_i ($1 \leq i \leq 6$) to refer to individual rows of Table 2a. Table 2c shows an example of the encrypted spatial time-series data as the output of Concealer.

Phase 2: *Query generation at $\mathcal{U}$.* A query $Q = \langle qa, att \rangle$, where qa is an aggregation (count, maximum, minimum, top-k, and average) or selection operation for a given condition, and att is a set of attributes with predicates on which query will be executed, is submitted to $\mathcal{SP}$ (❸). qa is always encrypted to prevent $\mathcal{SP}$ to know the query values.

Phase 3: *Query processing at $\mathcal{SP}$.* $\mathcal{SP}$ holds encrypted spatial time-series dataset and the user query (submitted to the secure hardware SGX). SGX, first, authenticate the user, and then, translates the query into a set of appropriate secured query trapdoors to fetch the tuples from the databases (❹). Note that since the individualized application is executed for the user itself, trapdoors are only generated if the authentication process succeeds to find that the user is wishing to know his past behavior.

The trapdoors are generated by using the methods of §4 for point queries or of §5 for range queries. On receiving encrypted tuples from the database (❺), the secure hardware, first, *optionally* checks their integrity using verifiable tags, and if find they have not tampered, decrypts them (if necessary), obliviously processes them, and produces the final answer to users (❻).

Phase 4: *Answer decryption at $\mathcal{U}$.* On receiving the answer, $\mathcal{U}$ decrypts them.

2.3 Algorithm Overview

Before going into details of Concealer's data encryption and query execution algorithms, we, first, provide their overview.

Data encryption method at $\mathcal{DP}$: partitions the time into slots, called *epochs*, and for each epoch, it executes the encryption method that consists of the following three stages:

Stage 1: *Setup.* Assume that we want to deal with two attributes (A and B), (*e.g.*, location and time). This stage: (*i*) creates a grid of size, say $x \times y$, (*ii*) sub-partitions the time into y subintervals, *e.g.*, for an epoch of 9-10am, creates y subintervals as: 9:00-9:10, 9:11-9:20, and so on, and (*iii*) using a hash function, say $\mathbb{H}$, allocates x values of A attributes over x columns, allocates y values (or y subintervals) of B attribute to y rows, and allocates some ***cell-ids*** $< x \times y$ (each with their ***counters*** initialized to zero) over the grid cells. (Such grid-creation steps can be used for more than two columns trivially and extended for non-time-series dataset.)

Stage 2: *Encryption:* In this stage, each sensor reading is encrypted and a *verifiable tag* is produced for integrity verification, as: (*i*) a tuple t_i is allocated to a grid cell corresponding to its desired column (*e.g.*, location and time) values using a *hash function*, the counter value of the cell-id is increased by one and attached with the tuples, and the tuples is encrypted to produce secure ciphertext with the encrypted counter value as a new attribute value, (*ii*) a hash-chain is created over the encrypted tuple values of the same cell-id for integrity verification (and verify false data injection or data deletion by $\mathcal{SP}$, and (*iii*) encrypted fake tuples are added (to prevent output-size leakage at $\mathcal{SP}$).

Stage 3: *Sharing:* This stage sends encrypted real and fake tuples with encrypted verifiable tags and encrypted cell-id, counter information to $\mathcal{SP}$.

Example. Table 2a shows six cleartext rows of an epoch. A 2×2 grid with three cell-ids cid_1, cid_2, and cid_3 is shown in Table 2b. Six cleartext rows are distributed over different cells of the grid. Table 2c shows the output of the encryption algorithm with fake tuples to prevent the output-size attack at $\mathcal{SP}$ and an index column created over the cell-ids. Encrypted Table 2c with counters and cell-ids (written below Table 2b) in encrypted form is given to $\mathcal{SP}$. ***Details of the encryption method and example will be given in §3.***■

Data insertion into DBMS at $\mathcal{SP}$: On receiving encrypted data from $\mathcal{DP}$, $\mathcal{SP}$ inserts the data into DBMS that creates/modifies the *index based on the counters* associated with each tuple.

Query execution at $\mathcal{SP}$: As a pre-processing phase, the enclave at $\mathcal{SP}$, first, authenticates the user, as mentioned in Phase 3 of §2.3, and then, executes the query, as follows:

Point queries. Consider a query on a location l and time t. For answering this, the enclave at $\mathcal{SP}$ executes the following steps:

(i) first execute the hash function $\mathbb{H}$ on query predicate l and t to know the cell-id, say cid_z, that was allocated by $\mathcal{DP}$ to l and t, (ii) using the information of cell-id and counter information, which was sent by $\mathcal{DP}$, create *static bins of a fixed size* (to prevent output-size leakage), (iii) among the created bins, find a bin, say B_i that has the cell-id cid_z that was obtained in the first step above, and (iv) fetch data from DBMS corresponding to the bin B_i, and (v) verify the integrity of data (if needed), *obliviously* process the data against query predicate in the enclave, and decrypt only the desired data.

Range queries. A range query, of course, can be executed by using the above point query method by converting the range query into several point queries, but will incur the overhead. To avoid the overhead of several point queries, we create static bins of fixed size over the fixed-sized groups of subintervals and fetch such bins to answer the query by following point queries' step (v). Following ***§3,§4,§5 will describe these algorithms in details***, and then §8 will compare these algorithms on different datasets and against different systems.

Example. Underlying DBMS at $\mathcal{SP}$ creates an index over Index column of Table 2c. SGX creates two bins over cell-ids's as $B_1 : \langle cid_1 \rangle$, $B_2 : \langle cid_2, f||1, f||2 \rangle$. Note that both bins corresponds to four rows—B_1 will fetch r_1, r_2, r_4, r_6, and B_2 will fetch $r_3, r_5, f||1, f||2$ rows. Thus, the output size will be the same. Now, consider a query $Q = \langle \text{count}, (l_2, t_5) \rangle$ over Table 2c. Here, SGX will know that it needs to fetch rows corresponding to the bin having cid_2, by generating four trapdoors: $\mathcal{E}_k(cid_2||1)$, $\mathcal{E}_k(cid_3||1)$, $\mathcal{E}_k(f||1)$, and $\mathcal{E}_k(f||2)$. Finally, based on the retrieve rows, SGX produces the final answer.■

3 DATA ENCRYPTION AT DATA PROVIDER

Concealer stores data in discredited time slots, called *epochs* or *rounds*. Epoch duration is selected based upon the latency requirements of $\mathcal{SP}$. Executing queries over multiple epochs could lead to inference attacks, and for dealing with it, we will present a method in §6. This section describes Algorithm 1, which is executed at $\mathcal{DP}$, for encrypting time-series data (assuming with three attributes location $\mathcal{L}$, time $\mathcal{T}$, and object O) belonging to one epoch. (In our experiments §8, we will consider different datasets with multiple columns.) Algorithm 1 uses deterministic encryption (DET) to support fast query execution. Since DET produces the same ciphertext for more than one occurrence of the same location and object, to ensure ciphertext indistinguishability, we concatenate each occurrence of the location and observation values with the corresponding timestamp.

In Concealer, queries retrieve a subset of tuples based on predicates specified over attributes, such as $\mathcal{L}$, O, or both. Queries, further, are always associated with ranges over time (see Table 4). Thus, to support the efficient execution of such queries, Concealer creates a cell-based index over query attributes (*e.g.*, $\mathcal{L}$ and/or O) along with time. For simplicity, Algorithm 1 illustrates how a cell-based index is created for location and time attributes, Index($\mathcal{L}, \mathcal{T}$). Similar indexes can also be created for other attributes, such as Index($O, \mathcal{T}$) and Index($\mathcal{L}, O, \mathcal{T}$). Details of Algorithm 1 is given below:

Key generation (Lines 2). Since using a single key over multiple epochs results in the identical ciphertext of a value, Concealer produces a key for encryption for each epoch, as: $k \leftarrow s_k||eid$, where s_k is the secret key shared between SGX and $\mathcal{DP}$, eid is the epoch-id (which is the starting timestamp of the epoch), and $||$ denotes concatenation. Thus, encrypting a value v using k in two different epochs will produce different ciphertexts. (Only the first eid and epoch duration is provided to SGX to generate other eids to decrypt the data during query execution.)

Algorithm 1: Data encryption algorithm.

```
Inputs: R: a relation. ℍ: a hash function. ℰ(): an encryption function. s_k: secret key.
Outputs: E(R): the encrypted relation.
1  Variables: ∀c_t ← 0, where 1 ≤ t ≤ r. x ← #ℍ(Domain(ℒ)),
     y ← #ℍ(Domain(𝒯)), cell_id[x, y] ← 0, c_tuple[u] ← 0.
2  Function key_gen(s_k) begin
3      k ← (s_k||eid)
4  Function encrypt_data(R) begin
5      for j ∈ (0, n − 1) do
6          Eo_j ← ℰ_k(o_j||t_j), El_j ← ℰ_k(l_j||t_j), Er_j ← ℰ_k(v_j||l_j||t_j)
7          Function Cell-Formation(j^th tuple) begin
8              p ← ℍ(l_j), q ← ℍ(t_j), cid_z^{p,q} ← cell_id[p, q]
9              c_t ← c_tuple[cid_z^{p,q}] ← c_tuple[cid_z^{p,q}] + 1
10             Ec_j ← E_k(cid_z^{p,q} ||c_t)
11         return E(R) ← ⟨Eo_j, El_j, Er_j, Ec_j⟩
12 Function add_fake_tuples() begin
13     for j ∈ (0, n − 1) do
14         Generate fake Eo_j, El_j, and Er_j, and Ec_j ← ℰ_k(f||j)
15         Append the j^th fake tuple to the relation E(R)
16 Function HashChain(c_tuple[u], ⟨Eo, El, Er⟩) begin
17     for j ∈ c_tuple[ ], ∀p tuples with same cell-id do
18         h_l^j ← H(El_p)||(H(El_{p−1})||(...||(H(El_2)||H(El_1))) ...)))
19         h_o^j ← H(Eo_p)||(H(Eo_{p−1})||(...||(H(Eo_2)||H(Eo_1))) ...)))
20         h_r^j ← H(Er_p)||(H(Er_{p−1})||(...||(H(Er_2)||H(Er_1))) ...)))
21         Ehl^j ← E(h_l^j), Eho^j ← E(o_l^j), Ehr^j ← E(h_r^j)
22 Function Transmit(E(R), cell_id[x, y], c_tuple[u]) begin
23     Ecell_id[x, y] ← ℰ_nd(cell_id[x, y]), Ec_tuple[u] ← ℰ_nd(c_tuple[u])
24     Permute all the tuples of the encrypted relation E(R)
25     Send E(R), Ecell_id[x, y], Ec_tuple[u], Ehl^j, Eho^j, Ehr^j
```

Tuple encryption (Lines 4-11). As the tuple arrives, it got appropriately encrypted (Line 6) using DET. Note that by encryption over the concatenated time with location and object values, results in a unique value in the entire relation. Now, in order to allocate the cell value to be used as the index, we proceed as follows: Let $|\mathcal{L}|$ be the number of locations and $|\mathcal{T}|$ be the duration of the epoch. Concealer maps the set of location $|\mathcal{L}|$ into a range of values from 1 to $x \leq |\mathcal{L}|$ using a simple hash function. It, furthermore, partitions $|\mathcal{T}|$ into $y > 1$ subintervals of duration, $|\mathcal{T}|/y$, which are then mapped using a hash function into $y > 1$ values. Thus, all tuples of the epoch are distributed randomly over the grid of $x \times y$ (see Example 3 below). Then, u cell-ids ($u < x \times y$) are allocated to grid cells. To refer to the cell-id of a cell, we use the notation $cid_z^{\{p,q\}}$ that shows that the cell $\{p, q\}$ is assigned a cell-id cid_z. In Step 3 of query execution §4.2, it will be clear that we will fetch tuples to answer any query based on cell-ids, instead of directly using query predicates.

Here, we keep two vectors: (i) *cell_id* of length $x \times y$ to keep the cell-id allocated to each cell of the grid, and (ii) *c_tuple* of length u to store the number of tuples that have been allocated the same cell-id. During processing a j^{th} tuple, we increment the current counter value of the number of tuples that have the same cell-id by one using *c_tuple*, and encrypts it. This value will be allocated to Index($\mathcal{L}, \mathcal{T}$)$\rangle$ attribute of the j^{th} (Lines 9-10).

Allocating fake tuples (Lines 12 -15). Since $\mathcal{SP}$ will read the data from DBMS into the enclave, different numbers of rows according to different queries may reveal information about the encrypted data. Thus, to fetch an equal number of rows for any query, $\mathcal{DP}$ needs to share some fake rows. There are two methods for adding the fake rows:

(i) *Equal number of real and fake rows*: This is the simplest method for adding the fake rows. Here, $\mathcal{DP}$ adds ciphertext secure fake tuples. The reason of adding the same number of real and fake rows is dependent on the property of the bin-packing algorithm,

which we will explain in §4.2 (Theorem 4.1).[3] In Index attribute, a j^{th} fake tuple contains an encrypted identifier with the tuple-id j, denoted by $\mathcal{E}_k(f||j)$, where f is an identifier (known to only $\mathcal{DP}$) to distinguish real and fake tuples.

(*ii*) *By simulating the bin-creation method*: To reduce the number of fake rows to be sent, we use this method in which $\mathcal{DP}$ simulates the bin-packing algorithm (as will be explained in §4.2) and finds the total number of fake rows required in all bins such that their sizes must be identical. Then, $\mathcal{DP}$ share such ciphertext secure fake tuples with their Index values, as in the previous method. As will be clear soon by Theorem 4.1 in §4.1, in the worst case, both the fake tuple generation methods send the same number of fake tuples, *i.e.*, an equal number of real and fake tuples.

Hash-chain creations (an optional step) Line 16-21. $\mathcal{DP}$ creates hash chains over encrypted tuples allocated the same cell-id, as follows: let p be the numbers of tuples with the same cell-ids and p encrypted location ciphertext as: $E(l_1), E(l_2), \ldots, E(l_p)$. Now, $\mathcal{DP}$ executes a hash function as follows:

$$h_{l1} \leftarrow H(E(l_1))$$
$$h_{l2} \leftarrow H(E(l_2)||h_{l1})$$
$$\ldots$$
$$h_l \leftarrow H(E(l_p)||h_{l(p-1)})$$

In the same way, hash digests for other columns are computed, and the final hash digest (*i.e.*, h_l) is encrypted that works as a verifiable tag at $\mathcal{SP}$.

Sending data (Line 22-25). Finally, $\mathcal{DP}$ permutes all encrypted tuples of the epoch to mix fake and real tuples in the relation and sends them with the two encrypted vectors *Ecell_id*[] and *Ec_tuple*[] and encrypted hash digests.[4]

Example 3. Now, we explain with the help of an example how encryption algorithm works. Consider six rows of Table 2a as the rows of an epoch, and we wish to encrypt those tuples with index on attributes $\mathcal{L}$ and $\mathcal{T}$. Assume that Algorithm 1 creates a 2×2 grid (see Table 2b) with three cell-ids: cid_1, cid_2, and cid_3. Table 2b shows two vectors *cell_id*[] and *c_tuple*[] corresponding to $\mathcal{L}$ and $\mathcal{T}$ attributes. Values in *c_tuple*[] show that the number of tuples has been allocated the same cell-id. For instance, $c\_tuple[1] = 4$ shows that four tuples are allocated the same cell-id (*i.e.*, cid_1). In Table 2b, for explanation purposes, we show which rows of Table 2a correspond to which cell; however, this information is not stored, only information of vectors *cell_id*[] and *c_tuple*[] is stored.

The complete output of Algorithm 1 is shown in Table 2c for cleartext data shown in Table 2a, where Index($\mathcal{L}, \mathcal{T}$) is the column on which DBMS creates an index. In Table 2c, $\mathcal{E}$ refers to DET, $\mathcal{E}_{nd}$ refers to a non-deterministic encryption function, and k be the key used to encrypt the data of the epoch. In addition, we create three hash chains, one hash chain per cell-id. Also, note that this example needs only 2 fake tuples to prevent output-size leakage at $\mathcal{SP}$.■

4 POINT QUERY EXECUTION

This section develops a bin-packing-based (BPB) method for executing point queries. Later, §5 will develop a method for range queries. The objectives of BPB method are twofold: first, create identical-size bins to prevent leakages due to output-size, (*i.e.*, when reading some parts of the data from disk to the enclave), and second, show that the addition of *at most n* fake tuples is enough in the worst case to prevent output-size leakage, where n is the number of real tuples. BPB method partitions the values of *c_tuple*[] into almost equal-sized bins, using which a query can be executed. Note that bins are created only once, prior to the first query execution. This section, first, presents the bin-creation method, and then, BPB query execution method.

4.1 Bin Creation

Bins are created inside the enclave using a bin-packing algorithm, after decrypting vector *Ec_tuple*[].

Bin-packing algorithms. A bin-packing algorithm places the given inputs having different sizes to bins of size at least as big as the size of the largest input, without partitioning an input, while tries to use the minimum number of bins. First-Fit Decreasing (FFD) and Best-Fit Decreasing (BFD) [8] are the most notable bin-packing algorithms and *ensure that all the bins (except only one bin) are at least half-full.*

In our context, u cell-ids ($cid_1, cid_2, \ldots, cid_u$) are inputs to a bin-packing algorithm, and the number of tuples having the same cell-id is considered as a weight of the input. Let *max* be the maximum number of tuples having the same cell-id cid_i. Thus, we create bins of size at least $|b| = max$ and execute FFD or BFD over u different cell-ids, resulting in $|Bin|$ bins as an output of the bin-packing algorithm.

The minimum number of bins. Let n be the number of real tuples sent by $\mathcal{DP}$, *i.e.*, $n = \sum_{i=1}^{i=u} c\_tuple[i]$. Let $|b|$ be the size of each bin. Thus, it is required to divide n inputs into at least $\lceil n/|b| \rceil$ bins.

Theorem 4.1. (Upper bounds on the number of bins and fake tuples) *The above bin-packing method using a bin size $|b|$ achieves the following upper bounds: the number of bins and the number of fake tuples sent by $\mathcal{DP}$ are at most $\frac{2n}{|b|}$ and at most $n + \frac{|b|}{2}$, respectively, where $n \gg |b|$ is the number of real tuples sent by $\mathcal{DP}$.*

In the full version (https://arxiv.org/abs/2102.05238), we will provide the proof of this theorem.

Equi-sized bins. The bins produced by FFD/BFD may have different numbers of tuples. Thus, we pad each bin with fake tuples, thereby all bins have $|b|$ tuples. Let $tuple_{b_i} < |b|$ be the number of tuples assigned to an i^{th} bin (denoted by b_i). Here, the ids (*i.e.*, the value of Index column) of fake tuples allocated to the bin b_i will be $|b| - tuple_{b_i}$, and all these fake tuple ids cannot be used for padding in any other bin. Thus, for padding, we create disjoint sets of fake tuple ids (see the example below to understand the reason).

Example 4.1. Assume five cell-ids $cid_1, cid_2, \ldots, cid_5$ having the following number of tuples $c\_tuple[5] = \{79, 2, 73, 7, 7\}$. Here, cid_1 has the maximum number of tuples; hence, the bin-size is at least 79. After executing FFD bin-packing algorithm, we obtain three bins, each of size 79: b_1: $\langle cid_1 \rangle$, b_2: $\langle cid_3, cid_2 \rangle$, and b_3: $\langle cid_5, cid_4 \rangle$. Here, bins b_2 and b_3 needs 4 and 65 fake tuples, respectively. One can think of sending only 65 fake tuples to access bins b_2 and b_3 to have size 79. However, in the absence of access-pattern hiding techniques, the adversary will observe that any 4 tuples out of 65 fake tuples are accessed in both bins. It will reveal that these four tuples are surely fake, and thus, the adversary may deduce that the bin size of b_2 is 75. Thus, $\mathcal{DP}$ needs to send 69 fake tuples in this example.■

4.2 Bin-Packing-based Query Execution

We develop bin-packing-based (BPB) method (see pseudocode in Algorithm 2) based on the created bins (over location and time

[3] For n real tuples, we add a little bit more than n fake tuples in the worst case (Theorem 4.1).
[4] The size of both vectors is significantly smaller (see experimental section §8.1).

Algorithm 2: Bin-packing-based query execution method.

Inputs: $\langle qa, (\mathcal{L} = l, \mathcal{T} = t)\rangle$: a query qa involving predicates over $\mathcal{L}$ and $\mathcal{T}$ attributes. $cell\_id[x, y]$, $c\_tuple[u]$, $\mathbb{H}$: A hash function, $\mathcal{E}_k()$: An encryption function using a key k.
$|Bin|$: the number of bins. $b[i][j]$: i^{th} bin having j cell-ids, where $j > 0$.
Outputs: A set of ciphertext queries.

1 **Function** *Query_Execution*$(\langle qa, (\mathcal{L} = l, \mathcal{T} = t)\rangle)$ **begin**
2 **Function** *Find_cell*(l, t) **begin**
3 $p \leftarrow \mathbb{H}(l)$, $q \leftarrow \mathbb{H}(t)$, $cid_z \leftarrow cell\_id[p, q]$
4 **return** cid_z; **break**
5 **Function** *Find_bin*$(cid_z, |Bin|, b[*][*])$ **begin**
6 **for** $desired \in (0, |Bin| - 1)$ **do**
7 **if** $cid_z \in b[desired][*]$ **then**
8 **return** $b[desired][*]$; **break**
9 **Function** *Formulate_queries*$(b[desired][*])$ **begin**
10 **for** $\forall j \in (b[desired][j])$ **do**
11 $cell\_id \leftarrow b[desired][j]$, $counter \leftarrow c\_tuple[cell\_id]$
12 $\forall counter$, generate ciphertexts $E_k(cell\_id||counter)$

attributes). A similar method can be extended for other attributes. BPB method contains the following four steps:

Step 0: Bin-creation. By following FFD or BFD as described above, this step creates bins over cell-ids ($c\_tuple[]$), if bins do not exist.

Step 1: Cell identification (Lines 2-4). The objective of this step is to find a cell of the grid corresponding to the requested location and time. A query $Q_e = \langle qa, (\mathcal{L} = l, \mathcal{T} = t)\rangle$ is submitted to the enclave that, on the query predicates l and t, applies the hash function $\mathbb{H}$, which was also used by $\mathcal{DP}$ (in *Cell-Formation* function, Line 8 of Algorithm 1). Thus, the enclave knows the cell, say $\{p, q\}$, corresponds to l and t. Based on the cell $\{p, q\}$ and using the vector $cell\_id[]$, it knows the cell-id, say cid_z, allocated to the cell $\{p, q\}$.

Step 2: Bin identification (Lines 5-8). Based on the output of Step 1, *i.e.*, the cell-id cid_z, this step finds a bin b_i that contains cid_z. Bin b_i may contain several other cell-ids along with identities of the first and the last fake tuples required for b_i.

Step 3: Query formulation (Lines 9-12). After knowing all cell-ids that are required to be fetched for bin b_i, the enclave formulates appropriate ciphertexts that are used as queries. Let the set of cell-ids in b_i be $C_1, C_2, \ldots, C_\alpha$, containing $\#_1, \#_2, \ldots, \#_\alpha$ records, respectively. Let the fake tuple range for b_i be f_l and f_h (let $\#_f = f_h - f_l$ be the number of fake tuples that have to be retrieved for b_i). The enclave generates $\#_i$ number of queries, as: $\mathcal{E}_k(C_p||j)$, where $1 \leq j \leq \#_i$ for each cell C_p corresponding to b_i and k is the key obtained by concatenating s_k and epoch-id (as mentioned in Line 2 of Algorithm 1). Also, it generates $\#_f$ fake queries, one for each of the fake tuples associated with b_i.

Advantage of cell-ids. Now, observe that a bin may contain several locations and time values (or any desired attribute value). Fetching data using cell-id does not need to maintain fine-grain information about the number of tuples per location per time.

Step 4: Integrity verification and final answers filtering. We may optionally verify the integrity of the retrieved tuples. To do so, the enclave, first, creates a hash chain over the real encrypted tuples having the same cell-id, by following the same steps as $\mathcal{DP}$ (Lines 16 of Algorithm 1). Then, compares the final hash digest with the decrypted verifiable tag, provided by $\mathcal{DP}$.

Now, to answer the query, the enclave, first, filters those tuples that do not qualify the query predicate, since all tuples of a bin may not correspond to the answer. Thus, decrypting each tuple to check against the query $Q_e = \langle qa, (\mathcal{L} = l, \mathcal{T} = t)\rangle$ may increase the computation cost. To do so, after implementing the above-mentioned Step 3, the enclave generates appropriate filter values ($\mathcal{E}_k(l_i||t_i)$ or $\mathcal{E}_k(o_i||t_i)$, which are identical to the created by $\mathcal{DP}$ using Algorithm 1); while, at the same time, DBMS executes queries on the encrypted data. On receiving encrypted tuples

```
max(int x,int y){
bool getX = ogreator(x, y),
return omove(getX, x, y)
}
```

(a) Oblivious maximum.

```
mov rcx, x
mov rdx, y
cmp rcx, rdx
setg al
retn
```

(b) Oblivious compare: `ogreator`.

```
mov rcx, cond
mov rdx, x
mov rax, y
test rcx, rcx
cmovz rax, rdx
retn
```

(c) Oblivious move: `omove`.

Figure 2: Register-oblivious operators [28].

from DBMS, the enclave performs string-matching operations using filters and decrypts only the desired tuples, if necessary.

Example 4.2. Consider the cells created in Example 3.1, *i.e.*, $cell\_id[] = cid_1, cid_2, cid_1, cid_4$ and $c\_tuple[] = \{4, 1, 1\}$. Now, assume that there are two bins, namely b_1: $\langle cid_1\rangle$ and b_2: $\langle cid_2, cid_3\rangle$. Consider a query $Q = \langle$count, $(l_2, t_5)\rangle$, *i.e.*, find the number of people at location l_2 at time t_5 on the data shown in Table 2c. Here, after implementing Step 1 and Step 2 of BPB method, the enclave knows that cell-id cid_2 satisfies the query, and hence, the tuples corresponding to bin b_2 are required to be fetched. Thus, in Step 3, the enclave generates the following four queries: $\mathcal{E}_k(cid_2||1)$, $\mathcal{E}_k(cid_3||1)$, $\mathcal{E}_k(f||1)$, and $\mathcal{E}_k(f||2)$. Finally, in Step 4, the filtering via string matching is executed over the retrieved four tuples against $\mathcal{E}_k(l_2||t_5)$. Since all the four retrieved tuples have a filter on location and time values, here is no need to decrypt the tuple that does not match the filter $\mathcal{E}_k(l_2||t_5)$.■

4.3 Oblivious Trapdoor Creation & Filtering

Steps for generating trapdoor (Step 3) and answer filtering (Step 4), in §4.2, were not oblivious due to side-channel attacks (*i.e.*, access-patterns revealed via cache-lines and branching operations) on the enclave. Thus, we describe how can the enclave produce queries and process final answers obliviously for preventing side-channel attacks.

Step 3. Let $\#_{Cmax}$ be the maximum cells required to form a bin. Let $\#_{max}$ be the maximum tuples with a cell-id. Let $\#_i$ be the number of tuples with a cell-id C_i. For a bin b_i, the enclave generates $\#_{Cmax} \times \#_{max}$ numbers of queries: $\mathcal{E}_k(C_i||j, v)$, where $1 \leq j \leq \#_{max}$, $C_i \leq \#_{Cmax}$, and $v = 1$ if $j \leq \#_i$ and C_i is required for b_i; otherwise, $v = 0$. Note that *this step produces the same number of queries for each cell and each bin.*

Let $\#_{fmax}$ be the maximum fake tuples required for a bin. Let $\#_{fb_i}$ be the maximum fake tuples required for b_i. The enclave generates $\#_{fmax}$ number of fake queries: $\mathcal{E}_k(f||j, v)$, where $1 \leq j \leq \#_{fmax}$ and $v = 1$ if $j \leq \#_{fb_i}$; otherwise, 0. This step *produces the same number of fake queries for any bin.* Finally, the enclave sorts all real and fake queries based on value v using a data-independent sorting algorithm (*e.g.*, bitonic sort [5]), such that all queries with $v = 1$ precede other queries, and sends only queries with $v = 1$ to the DBMS.

Step 4. The enclave reads all retrieved tuples and appends $v = 1$ to each tuple if they satisfy the query/filter; otherwise, $v = 0$. Particularly, an i^{th} tuple is checked against each filter, and once it matches one of the filters, $v = 1$ remains unchanged; while the value of $v = 1$ is overwritten for remaining filters checking on the i^{th} tuple. It hides that which filter has matched against a tuple. Then, based on v-value, it sorts all tuples using a data-independent algorithm.[5] (After this all tuples with $v = 1$ are decrypted and checked against the query by following the same procedure, if needed.)

Branch-oblivious computation. Note that after either generating an equal number of queries for any bin or filtering the retrieved tuples using a data-oblivious sort, the entire computation is still

[5] If all tuples can reside in the enclave, then bitonic sort is enough. Otherwise, to obliviously sort the tuples, we use column sort [21] instead of the standard external merge sort.

T_4	$cid_1^{\{1,1\}} = 40$	$cid_6^{\{1,2\}} = 30$	$cid_7^{\{1,2\}} = 2$	$cid_{11}^{\{1,4\}} = 9$
T_3	$cid_2^{\{2,1\}} = 50$	$cid_7^{\{2,2\}} = 50$	$cid_6^{\{2,3\}} = 21$	$cid_9^{\{2,4\}} = 60$
T_2	$cid_3^{\{3,1\}} = 60$	$cid_{11}^{\{3,2\}} = 40$	$cid_4^{\{3,3\}} = 45$	$cid_8^{\{3,4\}} = 48$
T_1	$cid_3^{\{4,1\}} = 40$	$cid_{10}^{\{4,2\}} = 50$	$cid_{10}^{\{4,3\}} = 10$	$cid_5^{\{4,4\}} = 50$
	l_1	l_2	l_3	l_4

Table 3: A 4×4 grid.

vulnerable to an adversary that can observe conditional branches, *i.e.*, an if-else statement used in the comparison. To overcome such an attack, we use the idea proposed by [28] that suggested that any computation on registers cannot be observed by an adversary since register contents are not accessible to any code outside of the enclave; thus, register-to-register computation is oblivious. For this, [28] provided two operators: `omove` and `ogreater`, see Figure 2. For any comparison in the enclave, we use these two operators; readers may find additional details in [28].

5 RANGE QUERY EXECUTION

This section develops an algorithm for executing range queries, by modifying BPB method, given in §4.2. For simplicity, we consider a range condition on time attribute. For illustration purposes, this section uses a 4×4 grid (see Table 3, which was created by $\mathcal{DP}$ using Algorithm 1, §3) corresponding to location and time attributes of a relation. In this grid, 11 cell-ids are used, and a number in a cell shows the number of tuples allocated to the cell. The notation T_i shows an i^{th} sub-time interval (after creating a grid using Algorithm 1 §3).

5.1 Enhanced Bin-Packing-Based (eBPB) Method

eBPB method requires $\mathcal{DP}$ to send the number of tuples in each cell of the grid with the vector *cell_id*[]. Thus, it avoids sending the vector *c_tuple*[]. For example, for the grid shown in Table 3, *cell_id*$[4,4] = \{(1,40),(6,30),(7,2),(11,9),(2,50),(7,50),(6,21),(9,60),(3,60),(11,40),(4,45),(8,48),(3,40),(10,50),(10,10),(5,50)\}$. This information helps us in creating bins more efficiently for a range query, as follows:

Step 1: Preliminary step. The enclave decrypts the encrypted vector *Ecell_id*[].

Step 2: Finding top-ℓ cell-ids. Find top-ℓ cells having the maximum number of tuples in one of the locations, where ℓ is the number of cells required to answer the range query. Say, location l_i has top-ℓ cells that have the maximum number of tuples, denoted by *bsize* tuples.

Step 3: Create bins. Execute this step either if ℓ cells required for the current query are more than the cells required for any previously executed query or it is the first query. Fix the bin size to *bsize* and execute FFD that takes $cid_z^{\{p,q\}}$ as inputs and the number of tuples having $cid_z^{\{p,q\}}$ as the weight of the input. If the bin does not have *bsize* number of tuples, add fake tuples to the bin. It results in $|Bin|$ number of bins and then, use all such bins for answering any range covered by ℓ cells.

Step 4: Query formulation and final answers filtering. Find the desired bin satisfying the range query and formulate appropriate queries, as we formed in Step 3 of BPB method §4.2. The DBMS executes all queries and provides the desired tuples to the enclave. The enclave executes the final processing of the query, likewise Step 4 of BPB method §4.2. *Note that for oblivious query formulation and result filtering, we use the same method as described in §4.3.*

Example 5.1.1. Consider a query to count the number of tuples at the location l_1 during a given time interval that is covered by T_2 to T_4. This query spans over three cells; see Table 3. Here, the maximum number of tuples in any three cells at locations l_1, l_2, l_3, and l_4 are $60+50+40 = 150$, $50+50+40 = 140$, $45+21+5 = 71$, and $60+50+48 = 158$, respectively. Thus, the bin of size 158 can satisfy any query that spans over any three cells (arranged in a column) of the grid. ■

Example 5.1.2: *attack on eBPB.* Consider the following queries on data shown in Table 3: (Q_1) retrieve the number of tuples having location l_1 during a given time interval that is covered by T_1 and T_2, and (Q_2) retrieve the number of tuples having location l_1 during a given time interval that is covered by T_2 and T_3. Answering Q_1 and Q_2 may reveal the number of tuples having T_1, T_2, and T_3, as: in answering Q_2 we do not retrieve 40 tuples (corresponding to $\{4,1\}$ cell) that were sent in answering Q_1 and retrieve 50 new tuples (corresponding to $\{2,1\}$ cell). It, also, reveals that 60 tuples (corresponding to $\{3,1\}$ cell) belong to T_2. Note that all such information was not revealed, prior to query execution, due to the ciphertext indistinguishable dataset.■

5.2 Highly Secured Range Query: winSecRange Method

We, briefly, explain a method to prevent the above-mentioned attacks on a range query. Particularly, we fix the length of a range, say $\lambda > 1$, and discretize n domain values, say $v_1, v_2, \ldots, v_n$, into $\lceil \frac{n}{\lambda} \rceil$ intervals (denoted by $\mathcal{I}_i$, $1 \leq i \leq \lceil \frac{n}{\lambda} \rceil$), as: $\mathcal{I}_1 = \{v_1, v_2, \ldots, v_\lambda\}$, …, $\mathcal{I}_{\lceil \frac{n}{\lambda} \rceil} = \{v_{n-\lambda}, \ldots, v_{n-1}, v_n\}$. Here, the bin size equals to the maximum number of tuples belonging to an interval, and bins are created for each interval *only once*. For example, consider 12 domain values: $v_1, v_2, \ldots v_{12}$, and $\lambda = 3$. Thus, we obtain intervals: $\mathcal{I}_1 = \{v_1, v_2, v_3\}$, $\mathcal{I}_2 = \{v_4, v_5, v_6\}$, $\mathcal{I}_3 = \{v_7, v_8, v_9\}$, and $\mathcal{I}_4 = \{v_{10}, v_{11}, v_{12}\}$. Here, four bins are created, each of size equals to the maximum number of tuples in any of the intervals. Now, we can answer a range query of length β by using one of the following methods:

(*i*) $\beta \leq \lambda$ and $\beta \in \mathcal{I}_i$: Here, the entire range exists in $\mathcal{I}_i$; hence, we retrieve only a single entire bin satisfying the range. E.g., if a range is $[v_1, v_2]$, then we need to retrieve the bin corresponding to $\mathcal{I}_1$. (*ii*) $\beta \leq \lambda$ and $\beta \in \{\mathcal{I}_i, \mathcal{I}_j\}$: It may be possible that $\beta \leq \lambda$ but the range β lies in $\mathcal{I}_i$ and $\mathcal{I}_j$, $i \neq j$. Thus, we need to retrieve two bins that cover $\mathcal{I}_i$ and $\mathcal{I}_j$, and hence, we also prevent the attack due to sliding the time window (see Example 5.1.2). E.g., if a range is $[v_2, v_4]$, then we need to retrieve the bins corresponding to $\mathcal{I}_1$ and $\mathcal{I}_2$. (*iii*) $\beta = z \times \lambda$: Here, a range may belong to at most $z+2$ intervals. Thus, we may fetch at most $z+1$ bins satisfying the query. E.g., if a range is $[v_3, v_8]$, then this range is satisfied by intervals $\mathcal{I}_1$, $\mathcal{I}_2$, and $\mathcal{I}_3$; thus, we fetch bins corresponding to $\mathcal{I}_1$, $\mathcal{I}_2$, and $\mathcal{I}_3$.

6 SUPPORTING DYNAMIC INSERTION

Dynamic insertion in Concealer is supported by batching updates into rounds/epochs, similar to [12]. Tuples inserted in an i^{th} period are said to belong to the round rd_i or epoch eid_i. The insertion algorithm is straightforward. Concealer applies Algorithm 1 on tuples of epochs prior to sending them to $\mathcal{SP}$. Note that since Algorithm 1 for distinct rounds is executed independently, the tuples corresponding to the given attribute value (*e.g.*, location-id) may be associated with different bins in different rounds. Retrieving tuples of a given attribute value across different rounds needs to be done carefully, since it might result in leakage, as shown next.

Example 6.1. Consider that the bin size is three, and we have the following four bins for each round of data insertion, where a bin b_i stores tuples of a location l_j:

Queries	Execution (filtering, decryption, and final processing) by the secure hardware
Location and time attributes	
Q1: # observations at l_i during time t_1 to t_x	SM using the filters $El_1 \leftarrow E_k(l_i \| t_1), El_2 \leftarrow \mathcal{E}_k(l_i \| t_2), \ldots, El_k \leftarrow \mathcal{E}_k(l_i \| t_x)$. No decryption needed.
Q2: Locations that have top-k observations during t_1 to t_x	SM using the filters $El_m \leftarrow \mathcal{E}_k(l_i \| t_j)$ where $i \in Domain(\mathcal{L})$ and $j \in \{t_1, t_x\}$, and then, decrypt $\mathcal{E}_k(l \| t \| o)$
Q3: Locations that have at least 10 observations during t_1 to t_x	of qualified tuples only for final processing.
Observation and time attributes	
Q4: Which locations have an observation o_i during t_1 to t_x	SM using $Eo_j \leftarrow \mathcal{E}_k(o_i \| t_j)$, $j \in \{t_1, t_x\}$, and then, decrypt $\mathcal{E}_k(l \| o \| t)$ of qualified tuples to know locations.
Observation, location, and time attributes	
Q5: # an observation o_i has happened at l_i during t_1 to t_x	SM using $Eo_j \leftarrow \mathcal{E}_k(o_i \| t_j \| l_i)$, where $j \in \{t_1, t_x\}$. No decryption needed.

Table 4: Sample queries. *Notation*: SM: String matching.

rd_1 : $b_1 : \langle l_1, l_2, l_3 \rangle$ $b_2 : \langle l_4, l_4, l_4 \rangle$ $b_3 : \langle l_5, l_5, l_5 \rangle$ $b_4 : \langle l_6, l_6, l_6 \rangle$
rd_2 : $b'_1 : \langle l_1, l_1, l_1 \rangle$ $b'_2 : \langle l_2, l_2, l_2 \rangle$ $b'_3 : \langle l_3, l_3, l_3 \rangle$ $b'_4 : \langle l_4, l_5, l_6 \rangle$

Now, answering a query for l_1 fetches bins b_1 and b'_1; a query for l_2 fetches b_1 and b'_2; and a query for l_3 fetches b_1 and b'_3. Here, b_1 is retrieved with three new bins (b'_1, b'_2, b'_3); it reveals that b_1 has three distinct locations. Similarly, b'_4 will be retrieved with three older bins (b_2, b_3, and b_4). Thus, the query execution on older and newer data reveals additional information.■

To prevent such attacks, we need to appropriately modify our query execution methods. In our technique, we will assume that bins across all rounds are of a fixed size, $|b|$,[6] and the number of tuples for a given attribute value (*i.e.*, location) fits within a bin (*i.e.*, $\leq |b|$). Our idea is inspired by Path-ORAM [36], while we overcome the limitation of Path-ORAM that achieves indistinguishability for query execution by keeping a meta-index structure at the trust entity. Note that Path-ORAM builds a binary tree index on the records. To retrieve a single record, Path-ORAM fetches $O(\log n)$ records and rewrites them under a different encryption. Since Path-ORAM uses an external data structure, it cannot be used for our purpose as argued in §1. Below, we provide our modified query execution strategy.

Executing queries. Let rd_i, rd_j, rd_k, and rd_l be four consecutive rounds of data insertion. Let q be a query that spans over rd_j, rd_k, and rd_l rounds; however, only rounds rd_j and rd_l have bins that satisfy query q. For answering q, the modified query execution method takes the following three steps: (*i*) The enclave fetches the desired bin from rd_j and rd_l rounds by following methods given in §4.2 and §5, with randomly selected $\log |Bin| - 1$ additional bins from each rd_j and rd_l round, where $|Bin|$ are created for each round using Algorithm 2. (*ii*) The enclave fetches $\log |Bin|$ bins from round rd_k, to hide the fact that rd_k does not satisfy the query. (*iii*) For round rd_x, $x \in \{j, k, l\}$, the enclave, first, permutes the retrieved data of rd_x and encrypts with a new key.[7] The newly encrypted data replaces the older data of rd_x.

Aside. We rewrite tuples of fetched bins, when asking a query for another value belonging to the previously fetched bin (*e.g.*, query for l_2 in Example 6.1). The adversary cannot link bins of different rounds of data insertion based on attribute values in bins.

7 SECURITY PROPERTIES

This section presents the desired security requirements, discusses which requirements are satisfied by Concealer, and information leakages from the algorithms. To develop applications on top of spatial time-series dataset at an untrusted $\mathcal{SP}$, a system needs to satisfy the following security properties:

Ciphertext indistinguishability: property requires that any two or more occurrences of a cleartext value look different in the ciphertext. Thus, by observing the ciphertext, an adversary cannot learn anything about encrypted data. Concealer satisfies this property by producing unique ciphertext for each tuple using Algorithm 1 (Line 7).

Data integrity: property requires that if the adversary injects any false data into the real dataset, it must be detected by a trusted entity. Concealer ensures integrity property by maintaining hash chains over the encrypted tuples and sharing encrypted verifiable tags, which helps SGX to detect any inconsistency between the actual data shared by $\mathcal{DP}$ and the data SGX accesses from the disk at $\mathcal{SP}$.

Query execution security: requires satisfying output-size prevention, indistinguishability under chosen keyword attacks (IND-CKA), and forward privacy.

Output-size prevention: property requires that the number of tuples corresponding to a value, *e.g.*, $\mathcal{L}$, (or a value corresponding to a combination of attributes, *e.g.*, $\mathcal{L}$ and $\mathcal{T}$) *i.e.*, the volume of the value, is not revealed, and only the maximum output-size/volume of the value is revealed. Concealer ensures this property by retrieving a fixed-size bin from DBMS into SGX, regardless of the query predicates.

IND-CKA. IND-CKA [11] prevents leakages other than what an adversary can gain through information about (*i*) *metadata*, *i.e.*, size of database/index, known as *setup leakage* $\mathfrak{L}_s$ in [11], and (*ii*) *query execution* that results in *query leakage* $\mathfrak{L}_q$ in [11] and includes search-patterns and access-patterns. Again, note that by revealing the access-patterns, IND-CKA is prone to attacks based on the output-size.

While Concealer leaks $\mathfrak{L}_s$ by the size of database/index and $\mathfrak{L}_q$ by fetching data in the form of a bin, it does not reveal information based on the output-size, except a constant output-size for all query predicates. (Also, since by fetching a bin, it does not reveal which rows of the bin satisfy the answer, it hides partial access-patterns.) Thus, Concealer improves the security guarantees of IND-CKA.

Forward privacy: property requires that newly inserted tuples cannot be linked to previous search queries, *i.e.*, the adversary that have collected trapdoors for previous queries, cannot use them to match newly added tuples. Concealer ensures forward privacy by, first, producing a different ciphertext of an identical value over two different epochs using two different keys (as mentioned in §3), and then, re-encrypting the tuples of different epochs using different keys during query execution spanning over multiple epochs (as mentioned in §6).

Now, we discuss information leakages from different the above-mentioned query execution methods.

BPB information leakage discussion. BPB method prevents the attacks based on output-size by fetching an identical number of tuples for answering any query. It reveals the dataset and index sizes stored in DBMS (as $\mathfrak{L}_s$ condition of IND-CKA [11]). BPB method, also, reveals *partial* access- and search-patterns, which means that for a group of queries it reveals a fixed bin of tuples, and thus, hides which of the tuples of bin satisfy a particular query ($\mathfrak{L}_q$ conditions of IND-CKA). Recall that an index, *e.g.*, B-tree index, on the desired attribute is created by the underlying DBMS. To show that the index will not lead to additional leakages other than $\mathfrak{L}_s$ and $\mathfrak{L}_q$, we follow the identical strategy to prove a technique is IND-CKA secure or not. In short, we need to show that a simulator not having the original data can also produce the

[6] We are not interested in hiding different numbers of tuples in different rounds, but using fake tuples it can be prevented, if desired.

[7] The key k for re-encryption is generated as: $k \leftarrow s_k \| eid \| counter$, where SGX maintains a counter for each round, and increments it by one whenever the data of a round is read in SGX and rewritten.

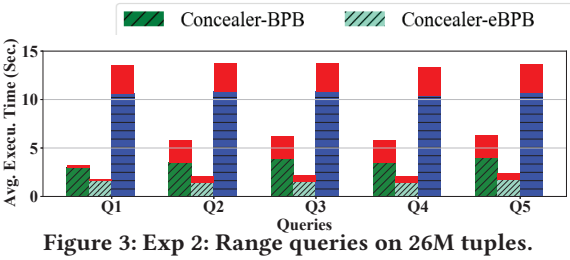

Figure 3: Exp 2: Range queries on 26M tuples.

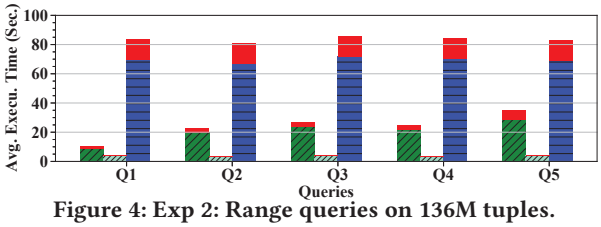

Figure 4: Exp 2: Range queries on 136M tuples.

index attribute based on $\mathfrak{L} = \{\mathfrak{L}_s, \mathfrak{L}_q\}$, *i.e.*, BPB method is secure if a "fake" attribute can mimic the real index attribute, (and hence, mimic the real index). Note that like SSEs, the simulator having only $\mathfrak{L}$ can generate a fake dataset, and hence, the index attribute can mimic the real index attribute; thus, the adversary cannot deduce additional information based on $\mathfrak{L}$.

Also, note that in oblivious Step 3, the enclave generates the same number of real/fake queries regardless of a bin and sorts them using a data-independent algorithm, which hides access-patterns in SGX. Also, it processes all retrieved tuples against the query and does oblivious sorting in Step 4. Thus, it also does not reveal access-patterns (by missing any tuple to process).

eBPB and winSecRange information leakage discussion. eBPB method incurs leakages $\mathfrak{L}_s$ and $\mathfrak{L}_q$. Based on $\mathfrak{L}_q$, we may reveal that a range query is spanning over at most ℓ cells. Hence, *it may also reveal the exact data distribution*, by fetching the same real tuple multiple times for multiple range queries, which we illustrated in Example 5.1.2. To overcome such information leakage, winSecRange fetches a fixed size interval, regardless of the query range. Thus, while winSecRange reveals $\mathfrak{L}_s$ and $\mathfrak{L}_q$, it does not reveal any information based on the output-size.

Insert operation information leakage discussion. Our insert operation satisfies forward privacy property. Since for encrypting tuples of an epoch, we generate a key that is unique among all keys generated for any epoch. Thus, based on the previous query trapdoors, the adversary cannot use them to link new tuples. Furthermore, our insert operation hides the distribution leakage due to executing queries over multiple epochs, since we fetch additional tuples from each epoch that lies in the query range and re-write all tuples.

8 EXPERIMENTAL EVALUATION

This section shows the experimental results of Concealer under various settings and compares them against prior cryptographic approaches.

8.1 Datasets, Queries, and Setup

Dataset 1: Spatial time-series data generation. To get a real spatial time-series dataset, we took our organization WiFi user connectivity dataset over 202 days having 136M(illion) rows. The IT department manages more than 2000 WiFi access-points (AP) by which they collect tuples of the form $\langle l_i, t_i, o_i \rangle$ on which they implemented Algorithm 1 prior to sending WiFi data to us. In this data, each of 2000+ APs is considered as a location. We created two types of WiFi datasets: (*i*) a small dataset of 26M WiFi connectivity rows collected over 44 days, and (*ii*) a large dataset of 136M rows (of 14GB) collected over 202 days. For Concealer Algorithm 1, which produces encrypted data as shown in Table 2c, we fixed a grid of $490 \times 16,000$ and allocated 87,000 cell-ids that resulted in two vectors *cell_id*[] and *c_tuple*[] of size 31MB. Data was encrypted using AES-256. This dataset has also skewed over the number of tuples at locations in a given time. For example, the minimum number of rows at all locations in an hour was ≈6,000, while the maximum number of rows at all locations in an hour was ≈50,000.

Dataset 2: TPC-H dataset. Since WiFi dataset has only three columns, to evaluate Concealer's practicality in other types of data with more columns, we used 136M rows of LineItem table of TPC-H benchmark. We selected nine columns (Orderkey (OK), Partkey (PK), Suppkey(SK), Linenumber (LN), Quantity, Extendedprice, Discount, Tax, Returnflag). This dataset contains large domain values, also; *e.g.*, in OK column, the domain value varies for 1 to 34M. We created (*i*) two indexes on attributes ⟨OK, LN⟩ and ⟨OK, PK, SK, LN⟩, (*ii*) two filters on concatenated values of ⟨OK, LN⟩ and ⟨OK, PK, SK, LN⟩, and (*iii*) one value that is the encryption of the concatenated values of all remaining five attributes. We used a $112,000 \times 7$ grid for index ⟨OK, LN⟩ and a $1500 \times 100 \times 10 \times 7$ grid for index ⟨OK, PK, SK, LN⟩. Each grid was allocated 87,000 cell-ids. The size of *cell_id*[] and *c_tuple*[] vectors for both grids was 54MB. Data is encrypted using Algorithm 1 and AES-256.

Queries. Table 4 lists sample queries supported by Concealer on spatial time-series data. These queries as mentioned in §2.1 provide aggregate (Q1-Q3) and individualized (Q4-Q5) applications. On TPH-C data, we executed count, sum, min/max queries.

Setup. The IT department (worked as $\mathcal{DP}$) had a machine of 16GB RAM. Our side (worked as $\mathcal{SP}$) had a 16GB RAM Intel Xeon E3 machine with Intel SGX. At $\mathcal{SP}$, MySQL is used to store data, and ≈8,000 lines of code in C is written for query execution.

We evaluate both versions of Concealer depending on the security of SGX: (*i*) one that assumes SGX to be completely secure against side-channel attacks, denoted byConcealer, and (*ii*) another that assumes SGX is not secure against side-channel attack (cache-line, branch shadow, page-fault attacks) and hence performs the oblivious computation in SGX (given in §4.3). denoted by Concealer+. In all our experiments, we show the overhead of preventing the side-channel attacks using red color.

8.2 Concealer Evaluation

This section evaluates Concealer on different aspects such as scalability, dynamic data insertion, the impact of the range length, and the number of cell-ids.

Exp 1: Throughput. Since Concealer is designed to deal with data collected during an epoch arriving continuously over time, we measured the throughput (rows/minute) that Concealer can sustain to evaluate its overhead at the ingestion time. Algorithm 1 can encrypt ≈37,185 WiFi connectivity tuple per minute. Also, it sustains our organization-level workload on the relatively weaker machine used for hosting Concealer.

Exp 2: Scalability of Concealer. To evaluate the scalability of Concealer, we compare the five queries as specified in Table 4 on the two WiFi datasets.

Point query. For point query, we executed a variant of Q1 when the time is fixed to be a point (instead of a range). Table 5 shows the average time taken by 5 randomly selected point queries (each executed 10 times). Note that, in Concealer, since the time taken by point queries is dependent upon the number of tuples allocated to the same cell-id (*i.e.*, the bin size) that was 2,378 rows

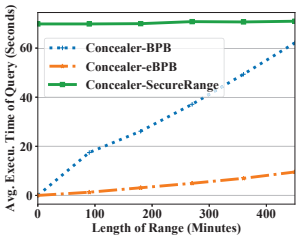

Figure 5: Exp 3: Range length impact.

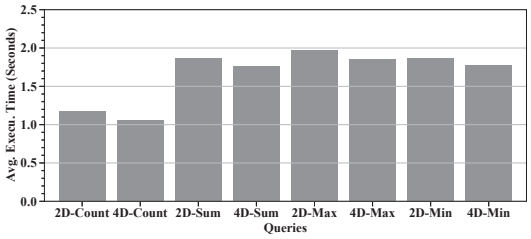

Figure 6: Exp 5: Query performance on TPC-H data.

from small and 6,095 rows from large datasets. Table 5 shows that Concealer with secure SGX using BPB method took 0.23s on small and 0.90s on large datasets, while Concealer+ with the current non-secure SGX using BPB method took 0.37s on small and 1.38s on large datasets. ***Time in Concealer+ increases compared to Concealer***, since we need to obliviously form the queries and obliviously filter the tuples in Concealer+, (and that implements a data-independent sorting algorithm; see §4.3). Here, executing the same query on cleartext data took 0.03s on small and 0.05s on large datasets.

	Small dataset (26M)	Large dataset (136M)
Cleartext processing	0.03s	0.05s
Concealer (secure SGX)	0.23s	0.90s
Concealer+ (non-secure SGX)	0.37s	1.38s

Table 5: Exp 2: Scalability of point query.

Range queries. To evaluate range queries, we set the default time range for queries Q1-Q5 specified in Table 4 to 20min (Exp 4 will study the impact of different range lengths). Figures 3 and 4 show the results as an average over 5 queries (each executed 10 times). We compare BPB, eBPB (§5.1), and winSecRange (§5.2) with both Concealer and Concealer+.

Recall that BPB method answers a range query by converting it into many point queries and fetches bins corresponding to each point query; while eBPB method fetches rows corresponding to top-ℓ cells, which cover the given range. In Concealer, a cell covers ≈18min. Thus, for a range of 20min, BPB and eBPB methods fetch at most 3 bins and at most 3 cells, respectively, for query Q1. Thus, for answering Q1, BPB fetches ≈6K rows from small and ≈18K rows from large datasets, and eBPB fetches ≈1.5K rows from small and ≈3K rows from the large dataset. Since eBPB retrieves few numbers of rows compared to BPB, in sGX, eBPB performs better than BPB (see Figures 3 and 4). Concealer+ again takes more time compared to Concealer for both eBPB and BPB, due to oblivious operations. Note that in queries Q2-Q5, we use more locations; thus, the number of rows retrieved changes accordingly, and hence, the processing time also changes, as shown in Figures 3 and 4.

For winSecRange, we set the range length on the time attribute to 8 hours in case of small and ≈1-day in case of large datasets. Thus, by fetching data for 1-day in the case of the large dataset, the enclave can execute any range query that is of a smaller time length. As expected, winSecRange took more time to execute range queries on both datasets, since it fetches and processes more rows (≈70K rows from small and ≈400K from large datsets). While it takes more time compared to BPB and eBPB, it prevents the attack by sliding the time window (as shown in Example 5.1.2), thereby, prevents revealing output-size attacks due to the sliding time window. Further, winSecRange under Concealer+ took more time compared to winSecRange under Concealer, due to oblivious operations. Recall that under Concealer, SGX architecture is vulnerable to side-channel attacks.

Exp 3. Impact of range length. Figure 5 shows the impact of the length in a range query on Concealer, by executing Q1 (see Table 4) with different time lengths over the large dataset and compares three approaches BPB, eBPB, and winSecRange. In Concealer, a cell covers ≈18min. Thus, for instance, for a range of 100min, BPB and eBPB methods fetch at most 7 bins and at most 7 cells, respectively. As expected, as the length of range increases, the number of rows to be fetched from the DBMS also increases, thereby, the processing time at secure hardware increases. As mentioned in Exp 2, for the large dataset, the range length is set to ≈1-day for winSecRange method; hence, fetching/processing more tuples takes more time and remains almost constant for the given length of queries.

Exp 4. Impact of dynamic insertion. We also investigated how does Concealer support dynamic insertion of WiFi dataset. We initiated Algorithm 1 for an hour of WiFi data at the peak hour, which included ≈50K tuples. For each insertion round, the grid size was $20 \times 1,250$ with 400 allocated cell-ids, and vectors $cell\_id[]$ and $c\_tuple[]$ of size ≈100KB were generated. In non-peak hours, we received at least ≈6K real rows. Recall that we are not interested in hiding peak vs non-peak hour data. Thus, all rows of each hour were sent using Algorithm 1. The query execution performance on dynamically inserted data depends on the number of rounds over which a query spans. For each round, we need to load the two vectors and fetch $\log |Bin|$ bins, as described in §6. For peak hour data, we obtained 146 bins storing ≈400 tuples, in each, using BPB method (§4.2) that resulted in ≈3K row retrieval. On this data, it took at most ≈4s to execute a query, re-encrypting tuples, and writing them, for Concealer.

Exp 5. Concealer on TPC-H data. To evaluate Concealer's practicality in other types of queries, we executed two-dimensional (2D) and four-dimensional (4D) count, sum, maximum, and minimum queries on LineItem table using Concealer. 2D (and 4D) queries involved OK and LN (and OK, PK, SK and LN) attributes. Similar to the point query execution on WiFi dataset, 2D and 4D queries on TPC-H data require to fetch tuples allocated the same cell-id (in the same bin) according to Algorithm 2. Thus, the query execution performance is dependent on the bin size, which was 400 rows for 2D grid and 6,258 for 4D grid. The query execution results are shown in Figure 6.

Figure 6 shows that Concealer using BPB method took ≈1s to 2s on TPC-H dataset. Also, observe that the performance of count queries is ≈36%–40% better than the others queries, since count queries do not need to decrypt retrieved rows and it executed string matching on the filter column to produce the answer. In contrast, other queries that require exact values of the attributes decrypt retrieved rows, and hence, incur the overhead.

8.3 Other Cryptographic Techniques

Since in our setting $\mathcal{SP}$ uses secure hardware, we need to compare Concealer against a system that supports database operations using SGX. Thus, we selected an open-source SGX-based system: Opaque [42].

Comparison between Opaque and Concealer. Opaque supports mechanisms to execute databases queries on encrypted data by first reading the entire data in the enclave, decrypting them, and then providing the answer. Note that both Opaque

System	Q1	Q2	Q3	Q4	Q5
Opaque	>10 m	>10 m	>10 m	>10 m	>10 m
Concealer eBPB	3.6 s	2.8 s	3.4 s	3 s	4s
Concealer winSecRange	70 s	67.2 s	71.9 s	70.2 s	68.9 s

Table 6: Exp 7: Range queries: Opaque vs Concealer.

and Concealer assume that SGX is secure against side-channel attacks, and hence, both reveal access-patterns. Thus, this is a fair comparison of the two systems, while Concealer avoids reading the entire dataset due to using the index and pushing down the selection predicate. Under this comparison, we execute point and range queries using Opaque and Concealer.

Further, note that since Concealer+ completely hides access-patterns inside SGX and partially hides access-patterns when fetching data in the form of fixed-size bins from the disk, we do not directly compare Concealer+ and Opaque due to different level of security offered by two systems.

Exp 6: Point queries on WiFi data. Opaque took more than 10min on both WiFi datasets for executing a variant of Q1 when the time is fixed to be a point, since *Opaque requires reading the entire dataset*. For the same query, Concealer took at most 0.23s on 26M and 0.9s on 136M rows.

Further, to execute the same query, Concealer+ took ≈1.4s. Thus, it shows that Concealer+, which hides access-patterns inside the enclave and prevents the output-size attack, is significantly better than Opaque.

Exp 7: Range queries on WiFi data. In all range queries Q1-Q5 on WiFi data, Concealer's eBPB and winSecRange algorithms take at most 4s and 71.9s over the large dataset compared to Opaque that took at least 10min in any query; see Table 6.

Further, to execute the same queries, Concealer+ takes at most 90s over the large dataset, which shows better performance of Concealer+ than Opaque in the case of range queries also.

Note. Except for Opaque, we did not experimentally compare Concealer against cryptographic techniques, since such techniques either offer different security levels [12, 22, 23], or do not scale to large data (*e.g.*, [4, 15]) for which we have designed Concealer, or are not publicly available. Thus, we decide to compare Concealer results with the reported result in different papers. Previous works on secure OLAP queries either support limited operations, reveal data due to DET or OPE, or scale to a smaller dataset. For example, Monomi [38], Seabed [29], [14], and [24] reveal data due to DET or OPE. Nevertheless, Seabed supports a huge dataset (≈1.75B rows). Novel SSEs, *e.g.*, [12, 22, 23], are very efficient, as given in their experiments; however, over 5M rows and susceptible to output-size attacks. We also checked access-pattern-hiding cryptographic work (*e.g.*, DSSE [15] and Jana [4]) that are prone to output-size attacks; however, as expected, these systems are slow due to using highly secure cryptographic techniques that incur overheads and/or do not support a large dataset. E.g., an industrial MPC-based system Jana took 9 hours to insert 1M LineItem rows, while executing a simple query took 532s.

9 CONCLUSION

This paper proposed Concealer that blends a carefully chosen encryption method with mechanisms to add fake tuples and exploits secure hardware to efficiently answer OLAP-style queries. We applied Concealer to real spatial time-series datasets, as well as, synthetic TPC-H data, and demonstrated scalability to large-sized data. Since Concealer allows indexing, its performance is similar to SSEs.Concealer offers two key advantages over existing SSEs: first, it does not require new data structures to incorporate into databases and leverages existing index structures of modern databases. Second (and perhaps more importantly), Concealer offers a higher level of security, in addition to being IND-CKA, which existing SSEs are, by preventing leakage of data distributions via output-size. *Due to space restrictions, we omit the query workload handling method, proof of Theorem 4.1, and experiments related to the overhead of verification, impact of the number of cells, and impact of different bin-sizes from this paper, and will be given in the full version (https://arxiv.org/abs/2102.05238).*

REFERENCES

[1] Blyncsy Inc. https://blyncsy.com/mercury-contact-tracing.
[2] R. Agrawal et al. Order-preserving encryption for numeric data. In *SIGMOD*, pages 563–574, 2004.
[3] P. Antonopoulos et al. Azure SQL database always encrypted. In *SIGMOD*, pages 1511–1525, 2020.
[4] D. W. Archer et al. From keys to databases - real-world applications of secure multi-party computation. *Comput. J.*, 61(12):1749–1771, 2018.
[5] K. E. Batcher. Sorting networks and their applications. In *AFIPS*, 1968.
[6] J. Bater et al. SMCQL: secure query processing for private data networks. *Proc. VLDB Endow.*, 10(6):673–684, 2017.
[7] D. Cash et al. Leakage-abuse attacks against searchable encryption. In *CCS*, pages 668–679, 2015.
[8] E. G. Coffman, Jr. et al. Approximation algorithms for NP-hard problems. chapter Approximation algorithms for bin packing: a survey. 1997.
[9] V. Costan et al. Intel SGX explained. *IACR Cryptology ePrint*, 2016:86, 2016.
[10] V. Costan et al. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX*, pages 857–874, 2016.
[11] R. Curtmola et al. Searchable symmetric encryption: Improved definitions and efficient constructions. *JCS*, 19:895–934, 2011.
[12] I. Demertzis et al. Practical private range search revisited. In *SIGMOD*, pages 185–198, 2016.
[13] A. Dhar et al. Proximitee: Hardened SGX attestation by proximity verification. In *CODASPY*, pages 5–16, 2020.
[14] T. Ge et al. Answering aggregation queries in a secure system model. In *VLDB*, pages 519–530, 2007.
[15] Y. Ishai et al. Private large-scale databases with distributed searchable symmetric encryption. In *CT-RSA*, pages 90–107, 2016.
[16] M. S. Islam et al. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.
[17] S. Jarecki et al. Updatable oblivious key management for storage systems. In *CCS*, pages 379–393, 2019.
[18] S. Kamara et al. Computationally volume-hiding structured encryption. In *EUROCRYPT*, pages 183–213, 2019.
[19] G. Kellaris et al. Generic attacks on secure outsourced databases. In *CCS*, pages 1329–1340, 2016.
[20] S. Lee et al. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX*, pages 557–574, 2017.
[21] T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Trans. on Computers*, 100(4):344–354, 1985.
[22] R. Li et al. Fast range query processing with strong privacy protection for cloud computing. *PVLDB*, 7(14):1953–1964, 2014.
[23] R. Li et al. Adaptively secure conjunctive query processing over encrypted data for cloud computing. In *ICDE*, pages 697–708, 2017.
[24] C. C. Lopes et al. Processing OLAP queries over an encrypted data warehouse stored in the cloud. In *DaWaK*, pages 195–207, 2014.
[25] P. Mishra et al. Oblix: An efficient oblivious search index. In *IEEE SP*, pages 279–296, 2018.
[26] M. Naveed. The fallacy of composition of oblivious RAM and searchable encryption. *IACR Cryptology ePrint Archive*, 2015.
[27] M. Naveed et al. Inference attacks on property-preserving encrypted databases. In *CCS*, pages 644–655, 2015.
[28] O. Ohrimenko et al. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, pages 619–636, 2016.
[29] A. Papadimitriou et al. Big data analytics over encrypted datasets with seabed. In *OSDI*, pages 587–602, 2016.
[30] V. Pappas et al. Blind seer: A scalable private DBMS. In *IEEE SP*, 2014.
[31] S. Patel et al. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In *CCS*, pages 79–93, 2019.
[32] R. Poddar et al. Arx: An encrypted database using semantically secure encryption. *Proc. VLDB Endow.*, 12(11):1664–1678, 2019.
[33] C. Priebe et al. Enclavedb: A secure database using SGX. In *IEEE SP*, pages 264–278, 2018.
[34] M. Shih et al. T-SGX: eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.
[35] D. X. Song et al. Practical techniques for searches on encrypted data. In *IEEE SP*, pages 44–55, 2000.
[36] E. Stefanov et al. Path ORAM: an extremely simple oblivious RAM protocol. *J. ACM*, 65(4):18:1–18:26, 2018.
[37] A. Trivedi and other. WiFiTrace: Network-based contact tracing for infectious diseases using passive WiFi sensing. *CoRR*, abs/2005.12045, 2020.
[38] S. Tu et al. Processing analytical queries over encrypted data. *PVLDB*, pages 289–300, 2013.
[39] J. Wang et al. Interface-based side channel attack against Intel SGX. *CoRR*, abs/1811.05378, 2018.
[40] W. Wang et al. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS*, pages 2421–2434, 2017.
[41] M. Xu et al. Hermetic: Privacy-preserving distributed analytics without (most) side channels.
[42] W. Zheng et al. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, pages 283–298, 2017.