VEIL: A Storage and Communication Efficient Volume-Hiding Algorithm

SHANSHAN HAN, VISHAL CHAKRABORTY, MICHAEL T. GOODRICH, and SHARAD MEHROTRA, University of California, Irvine, USA
SHANTANU SHARMA, New Jersey Institute of Technology, USA

This paper addresses *volume leakage* (*i.e.*, leakage of the number of records in the answer set) when processing keyword queries in encrypted key-value (KV) datasets. Volume leakage, coupled with prior knowledge about data distribution and/or previously executed queries, can reveal both ciphertexts and current user queries. We develop a solution to prevent volume leakage, entitled VEIL, that partitions the dataset by randomly mapping keys to a set of equi-sized buckets. VEIL provides a tunable mechanism for data owners to explore a trade-off between storage and communication overheads. To make buckets indistinguishable to the adversary, VEIL uses a novel padding strategy that allow buckets to overlap, reducing the need to add fake records. Both theoretical and experimental results show VEIL to significantly outperform existing state-of-the-art.

CCS Concepts: • **Security and privacy → Management and querying of encrypted data**; • **Information systems** → *Information retrieval query processing*; Data management systems.

Additional Key Words and Phrases: Volume leakage, bucketization, padding, secure query processing

ACM Reference Format:
Shanshan Han, Vishal Chakraborty, Michael T. Goodrich, Sharad Mehrotra, and Shantanu Sharma. 2023. VEIL: A Storage and Communication Efficient Volume-Hiding Algorithm. *Proc. ACM Manag. Data* 1, 4 (SIGMOD), Article 265 (December 2023), 27 pages. https://doi.org/10.1145/3626759

1 INTRODUCTION

Over the past two decades, secure data outsourcing to untrusted clouds has emerged as an important research area. Techniques to support operations such as keyword search [4, 8, 11, 18, 24, 31, 49], range search [14, 23, 37, 40, 43, 51, 53, 54], join computation [2, 7, 28, 41, 50], aggregations [45, 50], as well as, techniques to support SQL queries [2, 41] have been developed. Many such techniques use cryptographic primitives that allow the untrusted cloud to match queries by checking whether the ciphertexts corresponding to the query keys are stored at the cloud without having to decrypt the data. One of the first methods proposed as searchable encryption [4, 11, 24, 31, 46, 49] embeds a trapdoor/token (which is an encrypted query predicate) for a given query key into a random string, such that the equality/inequality of a query key can be checked against ciphertext. Several order-preserving encryption techniques [3, 35] to support range queries have subsequently been proposed. To find matching records over ciphertext in sublinear time, several encrypted indexing techniques have been proposed [4, 11, 17, 18, 29–31].

These cryptographic techniques suffer from information leakages via access patterns and volumes (or, output-size). Access patterns refer to the identity of the returned records. Volumes refer to the number of records returned to answer queries. The impact of access pattern leakage has been

Authors' addresses: Shanshan Han, shanshan.han@uci.edu; Vishal Chakraborty, vi.c@uci.edu; Michael T. Goodrich, goodrich@uci.edu; Sharad Mehrotra, sharad@ics.uci.edu, University of California, Irvine, Irvine, California, USA, 92697; Shantanu Sharma, shantanu.sharma@njit.edu, New Jersey Institute of Technology, Newark, New Jersey, USA.

2836-6573/2023/12-ART265
https://doi.org/10.1145/3626759

extensively discussed in literature [9, 16, 26, 32]. Oblivious Random Access Memory (ORAM) [19, 20, 36] and its improved version, known as Path-ORAM [47] are well-known tools to hide access patterns. However, both ORAM and Path-ORAM do not hide volume. Furthermore, they suffer from two problems: (*i*) query inefficiency, as they require fetching poly-logarithmic amount of data for a query, and (*ii*) limited throughput and limited support for concurrent users due to the underlying tree structure, which can support 0.055MB/s throughput (see Table 1 of [5]) and 30 concurrent clients [6], while existing DBMS, such as MySQL, offer high throughput and support for concurrent users.

In contrast, volume-hiding techniques have not been given much attention, until recently.The seminal work by Kellaris et. al. [33] and subsequent work [13, 22, 34, 42, 52] have shown that systems that hide access patterns can still be vulnerable to attacks that allow an adversary to reconstruct the database counts. *e.g.*, if an adversary has prior knowledge about the number of records corresponding to a query, it can potentially deduce/narrow down the query by observing the number of records in the answer set.

Volume hiding can be achieved by ensuring that the number of records returned is always equal, irrespective of the query being asked. Such records must include all the matching records, though they may also return additional records that are then filtered out by the querier. In the context of keyword queries (*e.g.*, as in key-value stores), this necessitates that the cloud returns at least the number of results that is greater than or equal to the number of key-value pairs corresponding to any key, *i.e.*, the maximum key size, denoted by L_{max}. Clearly, a technique returning less than L_{max} records will reveal to the adversary that the corresponding query is not for the key corresponding to the L_{max} values. A trivial solution to prevent volume leakage is to use Path-ORAM [47]. However, Path-ORAM will incur a huge communication cost of $O(L_{max} \times \log^2 |\mathcal{D}|)$, where $|\mathcal{D}|$ is the total number of key-value pairs (as discussed in §1.1 of [29]).

To avoid such communication cost, recent volume-hiding techniques [1, 17, 29, 30, 38, 44] have explored alternative approaches that do not use ORAM. As will become clear, such techniques store spurious (fake) encrypted records to prevent volume leakage. These methods may return more than L_{max} records in response to a query, or may store a typically small number of data records on the local side in plaintext, referred to as a "*stash*". Note that an optimal approach would not store any fake records, would not use a stash, and would retrieve L_{max} records for any query. We can, thus, compare the volume-hiding techniques based on the following metrics:

- ***Query amplification*** (***QA***): the ratio of the number of records returned for a query and L_{max} (optimal is 1).
- ***Storage amplification*** (***SA***): the ratio of the number of records stored in the encrypted database and the number of records in the plaintext dataset (optimal is 1).
- ***Stash ratio*** (***SR***): the ratio of the number of records stored locally and the number of records in the plaintext dataset (optimal is 0).

These metrics offer trade-offs, and we can easily design schemes with zero stash size (or, *SR*) that are optimal for one of *QA* or *SA* (but not for both). For instance, a strategy that retrieves the entire ciphertext database for any keyword query prevents volume leakage and has optimal storage overhead (*SA* is 1 since it does not store any fake records). But it has abysmal *QA* (which could be $O(|\mathcal{D}|)$, if, for instance, L_{max} is considered a constant and $|\mathcal{D}|$ is the size of the database). An alternate baseline strategy is to store key-value pairs as a multimap [1, 17, 29, 30, 38, 44], wherein volume leakage is prevented by storing enough fake records in the multimap associated with each key to ensure the number of records for each key is equal to L_{max}. Such a strategy will always

retrieve L_{max} records (hence optimal *QA*), but may result in very poor *SA*. In particular, if L_{max} is $O(|\mathcal{D}|)$, then the resulting ciphertext dataset may have $O(|\mathcal{D}|^2)$ key-value pairs![1]

One could possibly design schemes that are more efficient in terms of *QA* compared to retrieving the entire database, and more efficient in terms of *SA* compared to padding each key to L_{max}. For example, using Path-ORAM over multi-map, one can design an approach to obliviously retrieve L_{max} records including all records associated with the query key, but at the cost of *QA* to be $\log^2(|\mathcal{D}|)$, which is impractical. Instead, the existing volume-hiding techniques [17, 29, 30, 38, 48] have explored significantly better strategies for better *QA* and/or *SA* (and some of them [17, 29, 30] are likely integrated into MongoDB). For instance, one of these strategies, dprfMM [38], uses cuckoo hash to achieve *QA* and *SA* of 2, with a small stash having tight upper bounds. The most recent approach, XorMM [48] uses an XOR filter [21] to store the dataset and achieves *QA* of 1 and *SA* of 1.23. We will discuss these and other related strategies in §7.

Our contribution: We develop a novel strategy, entitled VEIL, that, given a key k, retrieves records associated with the key from the encrypted key-value store while preventing volume leakage.[2] Our approach is based on bucketing, wherein keys are mapped to a set of buckets that are then encrypted and stored. To retrieve the records for a given key, VEIL retrieves all the buckets that could contain the records of the given key. Unlike prior approaches, VEIL allows the database owner (or, user) to fine-tune the parameters/metrics – viz., *QA* and *SA*, which are input parameters to the system. A user can set the values of *QA* and *SA* to be any value more than or equal to 1, which VEIL can subsequently guarantee by using a local stash. Like existing volume-hiding techniques, VEIL also does not focus on hiding access-patterns.

Given that [38] already achieves *QA* and *SA* values of 2 with a small stash, our primary exploration in VEIL is the resulting stash size when *QA* and *SA* are below 2, *i.e.*, the number of records retrieved remains below $2L_{max}$ and the total number of records in the ciphertext database are below $2|\mathcal{D}|$. We show both analytically and experimentally that even when we choose relatively small values of these parameters (*e.g.*, $SA = 1.2$ and *QA* close to 1), VEIL achieves a very small stash, which experimentally is significantly smaller than that of the scheme in [38]. Thus, VEIL is a significantly better strategy that guarantees the prevention of volume leakage and achieves a near-optimal value of *QA* and acceptably small values of *SA* compared to the best-known strategy. We further optimize on VEIL by developing a modified strategy that allows buckets to share common records to further reduce *SA*. The modified strategy requires significantly less fake records to be added without increasing *QA* or the stash size. In summary, our contributions are:

- A flexible volume-hiding strategy that allows tuning storage overhead and query overhead to achieve a trade-off.
- A random bucketing strategy that distributes records of a key to buckets in a greedy way.
- Two strategies to add fake values to the created buckets, including a basic strategy that pads each created bucket to equal size, and an overlapping strategy that further reduces the number of fake records using a d-regular graph.
- Experimental evaluation shows that VEIL achieves flexible tuning of *SA* and *QA* and uses a small stash.

2 SETTINGS

This section describes the problem more concretely, including the adversarial model and the security model in VEIL.

[1]Consider a key with $L_{max} = |\mathcal{D}|/2$ records and the rest of the keys with only a single record each. The number of fake records is $(|\mathcal{D}|/2 - 1)^2$, which is $O(|\mathcal{D}|^2)$.

[2]Implementation of VEIL: https://github.com/han-shanshan/VEIL.

2.1 Problem Definition

We consider a key-value (KV) dataset $\mathcal{D}$ with a set of unique keys $\mathcal{K}$, where each key $k_i \in \mathcal{K}$ is associated with $|k_i|$ records. The maximum number of records associated with any key in $\mathcal{K}$ is denoted by L_{max}, *i.e.*, $L_{max} = \texttt{MAX}\{|k_i|\}_{k_i \in \mathcal{K}}$. The dataset $\mathcal{D}$ is encrypted in a ciphertext-secure manner, ensuring that no information is revealed from the ciphertexts, and subsequently outsourced to an untrusted public cloud server. Users, or the database owner, can query the encrypted dataset by sending encrypted queries for a key k_i to the cloud. In the absence of volume-hiding techniques, an adversary learns the number of records returned in response to an encrypted query, and this leakage enables the adversary to deduce the plaintext query key based on prior knowledge of the data distribution.

Our goal in this paper is to develop solutions to hide real volumes, *i.e.*, the number of records associated with the query key, during query processing. Before presenting an overview of our approach, we first discuss the adversarial model.

2.2 Adversarial Model

We consider a powerful adversary who knows the data distribution. That is, for a KV dataset $\mathcal{D}$ and its key set $\mathcal{K}$, the adversary is aware of each key $k_i \in \mathcal{K}$ and its volume $|k_i|$. The adversary also has full access to the ciphertext database. Let q be a query. Let $\mathsf{Cipher}(q)$ be the ciphertexts that q touches. Consequently, on executing the query, the adversary learns the association between q and the ciphertexts $\mathsf{Cipher}(q)$. The adversary may also know the keywords corresponding to (a subset of) prior queries. Let Q be the set of queries that have been executed in the database so far, and let $Q' \subseteq Q$. Suppose for each $q \in Q'$, the adversary knows the plaintext keyword associated with q. In the worst case, $Q' = Q$, *i.e.*, the adversary knows the query keyword for all prior queries executed in the database.

The adversary's objective is to determine the query keyword. It can achieve this by deducing the real volume of the current query key or other keys by observing queries and then mapping the ciphertexts to the corresponding plaintexts based on prior knowledge and/or query execution. It can also use known keys from previous queries to infer information about the plaintext query key. *The goal of hiding volumes is to prevent the adversary from deducing the query keys and obtaining information for other keys based on the knowledge of past queries.*

Security Requirement (VSR). Consider a KV dataset $\mathcal{D}$ with a key set $\mathcal{K}$ and a series of past queries to keys $\mathcal{K}_Q = \{k_1, \ldots, k_m\}$. We assume that there are at least two keys, say $k_1, k_2 \in \mathcal{K}$ that have never been queried before, *i.e.*, $|\mathcal{K}_Q| \leq |\mathcal{K}| - 2$, $k_1, k_2 \in \mathcal{K}$, and $k_1, k_2 \notin \mathcal{K}_Q$. Consider an adversary (based on the adversarial model discussed above) with the knowledge of: (*i*) data distribution, (*ii*) the corresponding ciphertext records and volume for each queried key $k_i \in \mathcal{K}_Q$, and (*iii*) the plaintext key for some queried keys $k_i \in \mathcal{K}_Q$. Suppose the adversary observes a new query to key k_α, where $k_\alpha \in \mathcal{K} - \mathcal{K}_Q$, the goal of the adversary is to deduce whether $k_\alpha = k_1$ or $k_\alpha = k_2$. A technique will be volume-hiding if the following condition holds:

$$Prob[k_\alpha = k_1 | Adv] = Prob[k_\alpha = k_2 | Adv] \tag{1}$$

That is the probability of the adversary (Adv) for guessing $k_\alpha = k_1$ is identical to the probability of guessing $k_\alpha = k_2$.

3 OVERVIEW OF VEIL

This section overviews Veil, a secure volume-hiding strategy for key-value stores. Veil partitions a KV dataset into buckets by associating records for a given key with one or more buckets. Given a query key, the buckets corresponding to the key, which may potentially store the records for the

key, are retrieved. These retrieved buckets may include extra records that are not associated with the query key. Such records are filtered out to obtain the query answer.

3.1 Notation

To describe Veil formally, we define the following notations. Let $\mathcal{D}$ be a key-value dataset, $\mathcal{K}$ be the set of keys in $\mathcal{D}$, and $\mathcal{B} = \{B_1, \ldots, B_n\}$ be the set of n buckets created over $\mathcal{D}$. Veil associates each key $k_i \in \mathcal{K}$ with a set of f buckets from $\mathcal{B}$, where f is referred to as the *fanout*. We define a function MAP that associates/maps a given key k_i to a set of buckets.

Definition 3.1 (MAP). *We define* $\mathsf{MAP} : \mathcal{K} \rightarrow \mathcal{P}(\mathcal{B})$ *where* $\mathcal{P}(\mathcal{B})$ *is the powerset of* $\mathcal{B}$*. For a key* $k_i \in \mathcal{K}$*, the function* $\mathsf{MAP}(k_i)$ *returns a set of* f *bucket-ids, denoted as* $\mathcal{B}[k_i]$*, that corresponds to* f *buckets in* $\mathcal{B}$*, i.e.,* $|\mathcal{B}[k_i]| = f$*, such that each record of* k_i *may reside in one of the buckets in* $\mathcal{B}[k_i]$*. Further,* $\forall k_i$ *and* $\forall B_j$ *such that* $B_j \notin \mathsf{MAP}(k_i)$*, the records of* $k_i \notin B_j$*.* □

Note that based on the definition above, records corresponding to a key k_i may or may not be in bucket B_j. We illustrate the notation above using the example below.

Example 3.1 (Example of MAP(*)*).* Consider a key-value dataset $\mathcal{D}$ that contains three records: $\mathcal{D} = \{\langle k_1, v_1\rangle, \langle k_1, v_2\rangle, \langle k_2, v_3\rangle\}$. Let $\mathcal{B} = \{B_1, B_2, B_3\}$ be the set of buckets created over $\mathcal{D}$, where bucket B_1 contains $\{\langle k_1, v_1\rangle\}$, bucket B_2 contains $\{\langle k_1, v_2\rangle, \langle k_2, v_3\rangle\}$, and bucket B_3 is empty. Here, $\mathsf{MAP}(k_1) = \{B_1, B_2\}$ and $\mathsf{MAP}(k_2) = \{B_2, B_3\}$ indicates that records of k_1 reside in B_1 and/or B_2, and records of k_2 reside in B_2 and/or B_3. □

Below we define *well-formed buckets*. Intuitively, we say that a set of buckets formed over a dataset is *well-formed* if each bucket is of the same size and MAP is defined appropriately.

Definition 3.2 (Well-Formed Buckets). *Let* MAP *be the function as defined above. The buckets are well-formed, if and only if the following hold:*

(1) Equal bucket size. *For all buckets* B_p *and* $B_q \in \mathcal{B}$*,* $|B_p| = |B_q|$*.*

(2) Disjoint Buckets. *For all buckets* B_p *and* $B_q \in \mathcal{B}$*,* $B_p \cap B_q = \emptyset$*.*

(3) Consistent mapping. *For all buckets* $B_p \in \mathcal{B}$ *and all key* $k_i \in \mathcal{K}$*, if* B_p *contains one or multiple records of* k_i*, then* $B_p \in \mathsf{MAP}(k_i)$*.*

We denote the set of well-formed buckets after padding by $\mathcal{B}_f$*.* □

Buckets can be made equal-sized by adding fake records to them appropriately. Suppose θ_1, θ_2 and θ_3 refer to "fake" records. Consider, again, Example 3.1. We add θ_1 to bucket B_1 and θ_2 and θ_3 to bucket B_3. Thus, we have $B_1 = \{\langle k_1, v_1\rangle, \theta_1\}$, bucket $B_2 = \{\langle k_1, v_2\rangle\langle k_2, v_3\rangle\}$, and bucket $B_3 = \{\theta_2, \theta_3\}$. The set of buckets $\{B_1, B_2, B_3\}$ is now well-formed.

3.2 Components of Veil

Veil is characterized by the following five operations: Bucket Creation, Padding, Data Outsourcing, Query Evaluation, and Filtering.

Bucket Creation $\mathcal{BC}$ *(*$\mathcal{D}$*,QA, SA, f)* $\rightarrow \langle$MAP, $\mathcal{B}$, *stash*$\rangle$*:* The function $\mathcal{BC}$ takes the dataset $\mathcal{D}$, the parameter *QA* (query amplification), the parameter *SA* (storage amplification), and a fanout f as inputs and returns a mapping MAP from keys to buckets, a set of buckets $\mathcal{B} = \{B_1, \ldots, B_n\}$, and a stash (containing a few records that do not fit in any buckets). Observe that $\mathcal{B}$ is consistent to MAP but buckets in $\mathcal{B}$ may be of unequal size.

Padding. To make the buckets well-formed, Veil adds fake records to make the buckets equi-sized. Let ℓ_b be the bucket size (§4 will explain the method of computing ℓ_b). For each bucket $B_j \in \mathcal{B}$, if $|B_j| < \ell_b$, we add fake records to B_j to pad it to size ℓ_b. We denote the set of well-formed buckets after padding by $\mathcal{B}_f$.

Data Outsourcing. This operation takes the well-formed buckets $\mathcal{B}_f$ and produces the following:

- *Encrypted Record Set:* that includes the set of all real or fake records in each bucket $B_i \in \mathcal{B}_f$. All such records are appropriately encrypted and outsourced as a record set. Each record is associated with a RID that will be used in a multimap index. For example, a key-value pair or a record $\langle k_i, v\rangle$ in a bucket is represented as: $\langle RID, E(k_i, v)\rangle$, where E is an encryption function, such as AES256 [12].[3]
- *Multimap Index, $Mmap(B_i)$:* contains a map for each bucket $B_i \in \mathcal{B}_f$ consisting of a list of RIDs associated with records in B_i.

The encrypted record set and the multimap index for each bucket are outsourced to a server. Furthermore, Veil also maintains information at the client for converting user queries into appropriate server-side queries. In particular, Veil stores the following information at the client:

- f: fanout that is the number of buckets in which records of a key may get mapped to.
- n: total number of buckets created by bucketing.
- Stash.

Aside. Note that the above strategy for outsourcing the $Mmap(B_i)$ corresponds to implementing the multimap index as a secondary index over an encrypted database of records. We could, instead, also implement $Mmap(B_p)$ as a primary index in which case the $Mmap(B_p)$ would store encrypted records instead of RIDs to the encrypted record stored in the encrypted record store.

Query Evaluation $\mathcal{QE}(k_i) \rightarrow \mathcal{B}[k_i]$: $\mathcal{QE}$ takes a query key k_i as input from the user and fetches encrypted buckets stored at the public cloud. Particularly, the client utilizes MAP function, $\mathsf{MAP}(k_i)$, to determine the f bucket-ids that may store the encrypted records of k_i and sends the f bucket-ids to the cloud to fetch the f buckets. Depending on the way the data is stored at the cloud (either in the form of a primary index or a secondary index, *e.g.*, multimap index), the cloud returns all the f buckets to the client.

Filtering: This operation takes the query key k_i and the encrypted records in the buckets retrieved by $\mathcal{QE}$ as inputs and decrypts them. All the records that are not corresponding to k_i are discarded. Further, the client reads the stash to find records having the key k_i.

Observe that in Veil, irrespective of the key, the volume of data retrieved remains the same because (1) buckets generated by $\mathcal{BC}$ are well-formed (hence equal-sized); (2) $\mathcal{QE}$ always retrieves the same number of buckets, *i.e.*, f.

Leakage from $\mathcal{BC}$. A potential leakage arises when the adversary possesses knowledge of the algorithm used by $\mathcal{BC}$ to create buckets. This serves as a motivation to address the design requirement of the $\mathcal{BC}$ algorithm developed in Veil, which will be discussed in §4. We illustrate a scenario where the adversary is aware that the algorithm $\mathcal{BC}$ used to create buckets is the first-fit decreasing[4] (FFD) [27] algorithm.

Example 3.2. Consider a dataset $\mathcal{D}$ with three keys k_1, k_2, and k_3 containing 3, 2, and 1 records, respectively. The bucket size is 3 (same as the largest number of records associated with any key in $\mathcal{D}$). Using FFD [27], we allocate the keys to buckets by first placing the largest key, *i.e.*, k_1, in bucket B_1. Since bucket size is 3 and k_1 has three corresponding records, B_1 lacks space for k_2. We create a new bucket B_2 for k_2. We also add k_3 to B_2. Thus, each query retrieves a bucket with 3 records from an adversary's perspective.

[3] AES produces an identical ciphertext for more than one appearance of a cleartext. Since each $\langle k_i, v\rangle$ pair is unique (*i.e.*, differs in at least one bit), all the ciphertext values will be non-identical (due to avalanche effect property [15]). In case there are two or more appearances of a $\langle k_i, v\rangle$ pair due to insert operation, we could add a random number to $\langle k_i, v, r\rangle$ before encryption to produce non-identical ciphertext.

[4] FFD is a well-known bin-packing algorithm that sorts keys in descending order by the size and places each key into the first available bucket with sufficient space. If a key cannot fit in an existing bucket, the algorithm creates a new bucket for the key. FFD creates at most $\frac{11}{9} \times OPT$ equisized buckets [27], where OPT is the least number of buckets required to store the database.

Consider an adversary (based on the adversarial model in §2.2) that has the following knowledge:
- Data distribution, *i.e.*, knowledge that k_1, k_2, and k_3 associate with 3, 2, and 1 records respectively.
- Prior queries, *i.e.*, the adversary knows that a prior query that retrieved B_2 corresponds to k_2.

Given the data distribution, the adversary can execute the FFD algorithm by itself to determine that of the two buckets, one contains records of k_1 and the other has records corresponding to k_2 and k_3. Now, if a query retrieves B_1, the adversary can infer that the query is for the key k_1. Likewise, if the query returns B_2, the adversary can determine that the query is for either k_2 or k_3 but NOT for k_1.

The example above illustrates leakage in scenarios where keys are mapped to a single bucket using FFD, *i.e.*, the fanout f is 1. Analogous examples can be created for other bucket creation algorithms, whether they yield a fanout of 1 or more, as these strategies construct buckets based on key sizes and a desired bucket size, which may inadvertently aid adversaries to infer information about keys. A possible solution to avoid such a leakage when using FFD is to use ORAM to fetch a bucket; however, this will come with communication overhead, as has been discussed in §1.

In this paper, we develop *VEIL, a bucketization-based strategy* that prevents the attack illustrated above and *ensures that the adversary cannot determine the volume of the results and thus cannot gain any information from the queries.*

To pad the buckets to well-formed, we also propose two strategies for padding, including a basic strategy that adds fake records to buckets to make them equal-size (in §4), and a sophisticated overlapping strategy that allows buckets to "borrow" records from other buckets to further minimize the number of fake records added while making the buckets well-formed (in §5).

Throughout the rest of this paper, we mainly focus on the bucket creation operation, which allocates records to buckets and pads them to achieve a well-formed structure. We do not extensively discuss the outsourcing and filter operations, as they are relatively straightforward. Since the query operation is intrinsically connected to the way we create buckets, we discuss it alongside bucket creation.

4 VEIL WITH RANDOM BUCKET CREATION

VEIL uses a random bucket creation strategy, denoted by $\mathcal{BC}$, that allows users to specify input parameters of query amplification *QA*, storage amplification *SA*, as well as the fanout f. $\mathcal{BC}$ generates a randomized mapping between keys and buckets, denoted by MAP, based on which it creates a set of buckets $\mathcal{B}$. In $\mathcal{BC}$, since keys are assigned randomly to buckets (*i.e.*, MAP is a randomized function), there is always a chance that not all records in $\mathcal{D}$ fit into the buckets in $\mathcal{B}$. Records that do not fit into the assigned buckets, *i.e.*, records in $\mathcal{D} - \cup_{B_j \in \mathcal{B}} B_j$ are assigned to a local storage that we refer to as a *stash*. Such a stash is stored at the local site (and not the public cloud). With the possibility of a stash, the retrieval algorithm in VEIL is slightly modified.

Given a query key k_i, in addition to retrieving all the corresponding buckets in $\mathsf{MAP}(k_i)$, the user also checks the stash for presence of records for key k_i. Note that the effectiveness of the strategy depends upon the size of the stash which we would like to be as small as possible. The stash size in $\mathcal{BC}$, as will become clear, depends upon factors including the *QA*, *SA*, and f. We theoretically show that even when we set these factors close to their optimal values , *i.e.*, 1, the expected size of the stash remains very small. This is also reflected by our experiments which clearly establish the superiority of the $\mathcal{BC}$ not only in terms of *QA* and *SA* but also in a much smaller size of stash compared to dprfMM [38] which also exploits the stash to store spillover records that do not fit into the cuckoo hash tables.

The MAP function in VEIL is implemented using a hash function $\mathcal{H}$ such as SHA-256 [39]. To generate bucket-ids for a query key k_i, VEIL appends an integer counter γ to the key k_i, where $1 \leq \gamma \leq f$. Subsequently, the hash function $\mathcal{H}$ processes the concatenated strings and produces f distinct

Algorithm 1: Map Algorithm

Inputs: k_i: a query key; n: total number of buckets; $\mathcal{H}$: a hash function. **Outputs:** $\mathcal{B}[k_i]$: a list of f bucket-ids corresponding to k_i.

Function MAP($k_i, n, \mathcal{H}$) **begin**

1 $\quad \mathcal{B}[k_i] \leftarrow []$
2 $\quad$ **for** $\gamma \in [1, f]$ **do**
3 $\quad\quad \mathcal{B}[k_i].append(\mathcal{H}(k_i|\gamma)\ \%\ n)$ ▷ Map to $[0, n-1]$
4 $\quad$ **return** $\mathcal{B}[k_i]$

Algorithm 2: $\mathcal{BC}$: Random Bucket Creation Algorithm.

Inputs: SA: a desired storage amplification. QA: a desired query amplification. f: fanout. L_{max}: maximum key size in the dataset. $\mathcal{D}$: dataset. $\mathcal{K}$: key set. MAP: a map function for each key and its corresponding buckets.

Outputs: $\mathcal{B}$: a list of buckets; $\mathcal{S}$: a stash.

1 $\ell_b \leftarrow \lceil QA \cdot L_{max}/f \rceil$ ▷ bucket size
2 $n \leftarrow \lceil SA \cdot |\mathcal{D}|/\ell_b \rceil$ ▷ the total number of buckets
3 $\mathcal{B} \leftarrow create\_empty\_buckets(n)$ ▷ create n empty buckets
4 $shuffle(\mathcal{D})$ ▷ mix the dataset $\mathcal{D}$
5 $\mathcal{S} \leftarrow []$ ▷ a local stash
6 **for** *each tuple* $t \in \mathcal{D}$ **do**
7 $\quad key \leftarrow t.extract\_key()$ ▷ extract the key from t
8 $\quad bucket\_id \leftarrow find\_smallest\_bucket(\text{MAP}(key))$ ▷ find the smallest bucket from the f buckets
9 $\quad$ **if** $\mathcal{B}[bucket\_id]$ *is not full* **then**
10 $\quad\quad \mathcal{B}[bucket\_id].add(t)$ ▷ add t to the smallest bucket
$\quad$ **else**
11 $\quad\quad \mathcal{S}.add(t)$ ▷ all buckets are full: add t to stash
12 **return** $\mathcal{B}, \mathcal{S}$

bucket-ids that are then associated with the key k_i. The algorithm for MAP is shown in Algorithm 1.

Veil Steps. We next discuss the components of Veil based on the randomized bucket creation.

Bucket Creation $\mathcal{BC}$. The pseudo-code for $\mathcal{BC}$ is presented in Algorithm 2. Initially, the algorithm calculates the bucket size ℓ_b, based on the maximum key size L_{max}, a user desired query amplification (QA), and the fanout f (Line 1). Subsequently, the algorithm determines the number of buckets n according to the desired storage amplification (SA) and the data size $|\mathcal{D}|$ (Line 2).

$$\ell_b = \frac{QA \times L_{max}}{f}, n = \frac{SA \times |\mathcal{D}|}{\ell_b} \tag{2}$$

Next, the algorithm generates n empty buckets (Line 3) and shuffles the dataset to mix the key-value pairs of different keys (Line 4). Additionally, a local stash is established for key-value pairs that cannot fit into a bucket (Line 5). For each key-value pair $\langle k_i, v \rangle$ arranged in a random order, the algorithm identifies the corresponding f bucket-ids for k_i based on MAP(k_i) (*i.e.*, Algorithm 1) and locates the smallest among the f buckets corresponding to k_i (Line 8) to place the key-value pair (Line 10). If the bucket is at capacity, indicating that all f buckets corresponding to k_i are full, the key-value pair will be placed into the local stash (Line 11).

Note that shuffling (in Line 4) prevents the adversary from learning the order in which keys are inserted into buckets. If we insert keys in the order in which they appear in the database, the adversary may be able exploit such information to gain information about the ciphertext. Say $\mathsf{MAP}(k_1) = \{B_1, B_2, B_3\}$ and $QA = 1$. Thus, all records in the 3 buckets are for k_1. Records of any other key mapped to these buckets will go to stash. Now, if an adversary gets to learn that a query q is for k_1, it will learn that all records in B_1, B_2, and B_3 are for the same key k_1. Shuffling prevents such a situation. Shuffling is implemented by permuting records in the database prior to creating buckets.

Also, note that $\mathcal{BC}$ utilizes user-defined SA and QA to determine the bucket size and the number of buckets. Consequently, the desired SA sets a limit on the number of fake records needed to pad the buckets later.

Padding. As the buckets created by Algorithm 2 may contain different numbers of records. In order to generate equi-sized buckets of size ℓ_b, we pad the buckets with fake records once $\mathcal{BC}$ has terminated.

Data Outsourcing. Let $\mathcal{B}_f$ be the set of well-formed buckets created by VEIL (Algorithm 2) after padding. Finally, the encrypted record set and multimap index are created and outsourced to the cloud, as explained in §3.2.

Query Evaluation and Filter. A query for k_i is executed by fetching f buckets from the cloud, after which the filter operation is executed at the client, as has been explained in §3.2.

Discussion: Analysis of VEIL based on both performance & security is formally presented in the extended version [25]. From the performance perspective, VEIL provides guaranteed storage and query amplification as specified by the user while ensuring that the expected size of stash is zero. From the security perspective, VEIL is secure based on the security requirement VSR in §2.2.

5 REDUCING STORAGE AMPLIFICATION

VEIL creates equal-sized buckets by padding each bucket to the same size, as discussed in §4. The resulting buckets are "disjoint", indicating that no two buckets share common records. In this section, we propose an optimization to VEIL that reduces storage amplification SA *without* increasing the stash size or query amplification QA. We introduce an alternate strategy called VEIL-O that allows buckets to share records, thereby decreasing the need to insert fake records to equalize the bucket sizes. The letter O represents "overlapping" between buckets due to the common records.

Definition 5.1. *Given a set of buckets $\mathcal{B}$ where each bucket contains key-value pairs, we say two distinct buckets B_p and B_q in $\mathcal{B}$ are* overlapping *if there exists a key-value pair $\langle k, v\rangle$ in both B_p and B_q. The records shared by the overlapping buckets are referred to as* common records. *The number of common records in $B_p \cap B_q$, i.e., $|B_p \cap B_q|$, is the* overlapping size. □

Intuitively, the optimized strategy allows buckets to borrow key-value pairs from other buckets instead of inserting fake records to pad themselves to the desired bucket size ℓ_b, thereby reducing the storage amplification SA.

5.1 Overlapping Buckets

Creating well-formed buckets by borrowing records from other buckets may seem straightforward. However, borrowing records indiscriminately may lead to information leakage.

Example 5.1. Consider a dataset $\mathcal{D}$ with three keys, k_1, k_2, k_3, containing 3, 1, and 3 values, respectively. Let the desired fanout f be 1, and let $\mathcal{D}$ be partitioned into three buckets $\mathcal{B} = \{B_0, B_1, B_2\}$ of size 3, where B_0 contains records of k_1, B_1 contains records of k_2, and B_2 contains records of k_3. To create equal-sized buckets, we allow B_1 to borrow records from B_0 and B_2 to increase its size to 3, as illustrated in Fig. 1. Observe that in this example, the buckets are well-formed: each bucket is equal-sized and consistent with the MAP function, which maps k_1, k_2, and k_3 to B_0, B_1, and B_2, respectively.

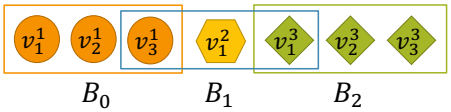


Fig. 1. An Intuitive Example of Unsecure Buckets

Such overlapping buckets can, however, lead to leakage. An adversary may infer buckets for some keys using the data distribution and access pattern (*i.e.*, the records retrieved when a query is executed). Let's examine how an adversary could infer that $\mathsf{MAP}(k_2) = B_1$ in the example above. Assume the adversary observes three queries that retrieve records in buckets B_0, B_1, and B_2, respectively. B_0 and B_1 share a common record (in the intersection of B_0 and B_1), so do B_1 and B_2. Note that B_0 and B_2 do not intersect. Given that k_1 and k_3 each have 3 records, and k_2 has 1 record, the adversary can easily deduce that neither k_1 nor k_3 could be mapped to B_1. If either were, all records in B_1 would correspond to k_1 (or k_3), in which case the other key with 3 records (*i.e.*, k_3 or k_1) would not have enough space for its records in either B_0 or B_2, since at least one of the records in those buckets (*i.e.*, the intersecting record with B_1) belongs to k_1 (or k_3). As a result, it must be the case that $\mathsf{MAP}(k_2) = \{B_1\}$, and the first three records and the last three records in Fig. 1 are for keys k_1 or k_3, respectively. Thus, the adversary not only learns which query is for key k_2, but also which ciphertext (in this case, the ciphertext in B_1 that does not intersect with either B_0 or B_2) corresponds to the key-value pair for key k_2.□

As shown in the example above, while allowing buckets to overlap can help ensure that results returned for queries are equi-sized, the resulting access pattern (which specific records get retrieved) could lead to inferences about the keyword. Of course, if the underlying system used an access pattern hiding technique such as ORAM to prevent access pattern leakage, the leakage above would not occur. But as we mentioned in §1, techniques such as ORAM can be computationally prohibitive. Instead, in Veil-O we devise clever ways to share records amongst buckets such that resulting access patterns do not leak additional information. This is described in the remainder of this section.

To prevent leakage for overlapping buckets, we must ensure that the adversary cannot distinguish between buckets based on the intersections between them. For a bucket, we define the notion of its **neighborhood** as the set of all buckets it overlaps with.

Definition 5.2. *Given a set of buckets $\mathcal{B}$, for a bucket $B_j \in \mathcal{B}$, we define its* neighborhood *as $\mathcal{N}(B_j) = \{B_p \in \mathcal{B} \mid |B_j \cap B_p| > 0\}$. We call each bucket in $\mathcal{N}(B_j)$ a neighbor of B_j.*

The well-formed bucket criteria in Definition 3.2, while sufficient, is not necessary to prevent leakage, thus we generalize the well-formed criteria for overlapping buckets.

Definition 5.3 (Well-Formed Buckets with Overlap.). *Let $\mathcal{D}$ be a key-value dataset, $\mathcal{K}$ be the set of keys in $\mathcal{D}$, and $\mathcal{B}$ be the set of n buckets created over $\mathcal{D}$. Let $\mathsf{MAP}(^*)$ be the function that maps keys in $\mathcal{K}$ to f buckets in $\mathcal{B}$. We say that the buckets in $\mathcal{B}$ are well-formed if and only if*

(1) Equal bucket size. *For all buckets B_p and $B_q \in \mathcal{B}$, we have $|B_p|=|B_q|$.*

(2) Constraints on Overlap.

- Equal sized neighborhood. *For all buckets B_p and $B_q \in \mathcal{B}$,$|\mathcal{N}(B_p)| = |\mathcal{N}(B_q)|$.*
- Equal overlapping size. *For all $\mathcal{B}_1$ and $\mathcal{B}_2$ in $\mathcal{P}(\mathcal{B})$, where $\mathcal{P}(\mathcal{B})$ is the power set of $\mathcal{B}$, with*
  - *$|\mathcal{B}_1| = |\mathcal{B}_2|$,*
  - *for all $B_1, B_1'' \in \mathcal{B}_1$, buckets B_1' and B_1'' are overlapping, and for all $B_2', B_2'' \in \mathcal{B}_2$, buckets B_2' and B_2'' are overlapping, we have $|\bigcap_{B\in\mathcal{N}(B_p)} B| = |\bigcap_{B'\in\mathcal{N}(B_q)} B'|$.*

(3) Consistent mapping. *For all buckets $B_p \in \mathcal{B}$ and all keys $k_i \in \mathcal{K}$, if B_p contains one or multiple records of k_i, then $B_p \in \mathsf{MAP}(k_i)$.* □

Intuitively, the well-formed definition above requires each bucket to overlap with the same number of buckets, and the number of common records between a subset of overlapping buckets of any size to be the same. This ensures buckets are indistinguishable from each other. Note that the well-formed definition in §3.2 is subsumed by the definition above - *i.e.*, if buckets are not overlapping, the above definition reduces to that of the well-formed buckets as in Definition 3.2. Henceforth, by well-formed, we refer to the well-formed buckets with overlap.

To form well-formed overlapping buckets, we superimpose a structure of a random d-regular graph over $\mathcal{B}$, the set of buckets. In the graph, $G = (V, E)$, each vertex $v_i \in V$ corresponds to the bucket $B_j \in \mathcal{B}$, and edges correspond to d neighbors, which we will see are assigned randomly. For illustration, let us consider a situation where d is 3, thus, each node has exactly 3 neighbors (*i.e.*, intersection of any four buckets is empty). Fig. 2 illustrates such a 3-regular graph with an initial set of buckets $\mathcal{B}$= $\{B_0, B_1, B_2, B_3\}$ and the corresponding neighbors. Neighboring buckets borrow/lend records to each other, and hence may overlap.

With the d-regular graph $\mathcal{G} = (V, \mathcal{E})$, we further associate the following: (*i*) **Weight** (denoted by δ) that specifies the number of records shared between one bucket and each of its neighbors. Note that δ is a constant - *i.e.*, the number of records shared between neighbors is always equal to δ. (*ii*) **Direction** that is a function $dir(v_p, v_q) : E \rightarrow \{0, 1\}$ that assigns to each edge in G a direction representing which corresponding bucket borrows/lends to its neighbor. Suppose bucket B_p borrows from bucket B_q then $dir(v_p, v_q) = 0$ and $dir(v_q, v_p) = 1$. (*iii*) **Labels** that indicate the specific common records that neighbors borrow/lend from each other. For an edge (v_p, v_q), if $dir(v_p, v_q) = 0$, then the label $label(v_p, v_q)$ is the set of the records that B_p borrows from B_q. Conversely, if $dir(v_p, v_q) = 1$, then $label(v_p, v_q)$ is the set of the records that B_q borrows from B_p. We illustrate an example graph $\mathcal{G}$. To simplify notations, we only show values in the buckets.

Suppose the generated buckets $\{B_0, B_1, B_2, B_3\}$ contain 4, 2, 1, and 3 records, respectively, with the bucket size to be 4. We can pad each bucket to the size of 4 as follows. B_0 currently contains 4 records and, thus, cannot accept additional values. It can, however, provides one value to each of its neighbors B_1, B_2, and B_3. In turn, B_3 receives one record from B_0, achieving the target bucket size of 4, and contributes one record to each of B_1 and B_2. Now, B_1 receives two records from B_0 and B_3 to reach the desired size of 4, subsequently offering one record to B_2. Finally, B_2 receives three values from B_0, B_1, and B_3 to fulfill its size requirement of 4. In Fig. 2, each bucket overlaps with 3 buckets, the weight over each edge is 1, and each bucket is padded to size 4, thus the buckets are indistinguishable from the adversary's perspective.

Observe that overlapping strategies reduce *SA* by allowing sharing records across buckets. In Fig. 2, each bucket overlaps with 3 buckets to achieve the desired bucket size. The number of fake records required is 0. On the other hand, when simply padding each bucket to size 4 using fake records, B_1, B_2, and B_3 require 2, 3, and 1 fake records, respectively, and the total number of fake records is 6.

Based on the idea to pad buckets to make them equi-sized by lending/borrowing records from neighboring buckets, we develop a strategy, entitled Veil-O, that ensures the security requirement VSR. Veil-O uses the same randomized strategy to create initial buckets $\mathcal{B}$ as that used in Veil. It, however, allows buckets to overlap to make the buckets equi-sized as discussed in the following subsection.

5.2 Veil-O: Padding

Veil-O uses a padding strategy that allows sharing of records between buckets to reduce the number of fake records added, thereby reducing *SA*. Given a KV-dataset $\mathcal{D}$, Veil-O first creates a set of n buckets $\mathcal{B} = \{B_0, \ldots, B_{n-1}\}$ using the randomized bucket creation algorithm $\mathcal{BC}$ (§4), then implements a padding strategy that, unlike Veil, enables buckets to borrow records from their neighboring buckets, creating well-formed buckets with overlap as defined in Definition 5.3. The padding strategy consists of a sequence of steps (an overview of the strategy is presented in Algorithm 3) that starts by first creating a d-regular graph from the buckets in $\mathcal{B}$ (function Graph Creation (GC)). Such a graph has n nodes, each corresponding to a bucket in $\mathcal{B}$, and undirected edges between overlapping buckets (*i.e.*, buckets that share records). The graph $\mathcal{G} = (V, \mathcal{E})$ is represented as $\mathcal{B}$, representing the set of buckets (vertices), and $\mathcal{M}$, an adjacency matrix corresponding to $\mathcal{E}$. The next step determines the maximum possible overlap size between neighbors that will still result in well-formed overlapping buckets, *i.e.*, function Maximum Overlap Determination (MOD). The MOD function returns the number of records that a bucket B_p can borrow/lend from/to a specific neighbor B_q. Next, Veil-O uses the function Edge Direction Determination (EDD) to determine the directions of edges, *i.e.*, for each pair of neighbors B_p and B_q whether B_p borrows or lends records from/to B_q. This determines $dir(v_p, v_q)$ for each $(v_p, v_q) \in \mathcal{E}$. We represent $dir(v_p, v_q)$ by transforming $\mathcal{M}$ into an adjacency matrix $\overline{\mathcal{M}}$ that represents directed edges. Thus, if $dir(v_p, v_q) = 0$ and $dir(v_q, v_p) = 1$ (*i.e.*, B_p borrows from B_q), we remove the edge (v_p, v_q) from $\overline{\mathcal{M}}$ but let the edge (v_q, v_p) remain. Next, fake records are added to buckets to ensure that each bucket consists of all records assigned to it originally, all the records it borrows from its neighbors, and fake tuples add up to exactly the bucket size. Addition of fake tuples changes the buckets in $\mathcal{B}$ resulting in $\mathcal{B}^f$, where f represents "full". A bucket $B_p \in \mathcal{B}^f$ contains fake records in addition to the original records associated with B_p. Next, the algorithm determines the labels to be associated with each edge in the d-regular graph $\mathcal{G}$. Each label represents the records that are borrowed/lent between two overlapping buckets. As a final step, using the labels generated, and the buckets in $\mathcal{B}^f$, Veil-O generates the set of well-formed buckets which are then outsourced.

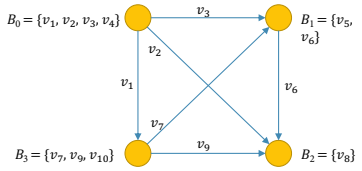


Fig. 2. A 3-Regular Graph.

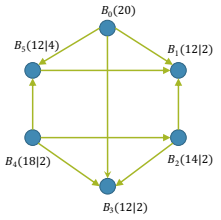


Fig. 3. A 3-Regular Graph of Veil-O

	B_0	B_1	B_2	B_3	B_4	B_5
B_0		✓		✓		✓
B_1						
B_2		✓		✓		
B_3						
B_4			✓	✓		✓
B_5		✓				

Fig. 4. Adjacency Matrix $\overline{\mathcal{M}}$

We illustrate the above described algorithm in Algorithm 3 and then describe each step in details. The algorithm takes the set of buckets $\mathcal{B}$ created using $\mathcal{BC}$ as inputs and returns δ, the overlapping size, *i.e.*, the number of records shared by any two overlapping buckets, and a matrix $\overline{\mathcal{M}}$, representing the d-regular graph containing information about neighbors and a directed edge

Algorithm 3: VEIL-O Padding Algorithm

Inputs: $\mathcal{B}$: buckets created using $\mathcal{BC}$ (Algorithm 2); d: the number of neighbors for each bucket; ℓ_b: bucket size.
Outputs: $\mathcal{B}_f$: A set of well-formed overlapping buckets; **begin**

$\mathcal{B}, \mathcal{M}, \leftarrow \boldsymbol{GC}(\mathcal{B}, d)$
$\delta_{MAX} \leftarrow \boldsymbol{MOD}(\mathcal{B}, \mathcal{M})$
$\mathcal{B}, \overline{\mathcal{M}}, \delta \leftarrow \boldsymbol{EDD}(\mathcal{B}, \mathcal{M}, \delta_{MAX}, \ell_b)$
$\mathcal{B}_f, \overline{\mathcal{M}}, \delta \leftarrow \boldsymbol{AFV}(\overline{\mathcal{M}}, \delta)$
$\mathcal{L} \leftarrow \boldsymbol{LC}(\mathcal{B}_f, \overline{\mathcal{M}}, \delta)$
$\mathcal{B}_f \leftarrow \boldsymbol{WFBC}(\mathcal{B}_f, \mathcal{L})$
return $\mathcal{B}_f$

representing which one of each two neighboring buckets borrow/lends to the other. The strategy of which specific records a bucket B_p will borrow from/lend to its neighbor B_q such that the resulting buckets become well-formed will be discussed separately.

STEP 1. GRAPH CREATION (GC). [Algorithm 4] Input to this step are buckets in $\mathcal{B}$ and a value of d and the output consists of a d regular graph $\mathcal{G}$ with nodes corresponding to buckets in $\mathcal{B}$ and undirected edges between buckets that will be neighbors to each bucket.[5] It is assumed that at least one of d and $|\mathcal{B}|$ are even, as the number of edges in a d-regular graph is computed as $d|\mathcal{B}|/2$, and if both d and $|\mathcal{B}|$ are odd then a d-regular graph cannot be constructed. The graph is represented with $\mathcal{B}$, the set of vertices, and an adjacency matrix $\mathcal{M}$ denoting the edges. Let $\mathcal{B} = \{B_0, B_1, \ldots, B_{n-1}\}$. For a bucket $B_p \in \mathcal{B}$, VEIL-O assigns a set of d neighboring buckets B_p^j ($j = 1, \ldots, d$) using an *ordered set* of carefully chosen d functions $\mathcal{F} = \langle F_1, F_2, \ldots, F_d \rangle$, where for each function $F_j \in \mathcal{F}$, $F_j : \mathcal{B} \rightarrow \mathcal{B}$. We denote $F_j(B_p)$ as B_p^j and refer to it as the j^{th} neighbor of B_p. The functions in $\mathcal{F}$ are such that if a bucket B_q is a neighbor of B_p (*i.e.*, for some function $F_j \in \mathcal{F}$, $B_q = F_j(B_p)$), then B_p is also a neighbor of B_q, *i.e.*, there exists a $1 \leq j' \leq d$ such that $F_{j'}(B_q) = B_p$.

We define a set of such functions in $\mathcal{F} = \{F_1, \ldots, F_d\}$ as follows.

- For $1 \leq i \leq \lfloor d/2 \rfloor$ we have $F_i(B_p) = B_q$ where $q = (p + i) \mod n$
- For $1 \leq i \leq \lfloor d/2 \rfloor$, we have $F_{d-i+1}(B_p) = B_q$ where $q = (p - i + n) \mod n$
- If d is odd $F_{(d+1)/2}(B_p) = B_q$ where $q = (p + (n/2)) \mod n$.[6]

We illustrate the functions used for assigning neighbors using the example below.

Example 5.2. Consider six buckets $B_0, \ldots, B_5$ in Fig. 3 Thus, $n = 6$. Let us consider d to be 3. According to rule 1, we get $F_1(B_p) = B_{(p+1) \mod 6}$, which finds neighbor bucket B_1 for $B_0, \ldots, B_5$ for B_4, and B_0 for B_5. According to rule 2, we get $F_3(B_p) = B_{(p-1+6) \mod n}$, which finds neighbor bucket B_5 for B_0, B_0 for B_1, ..., and B_4 for B_5. According to rule 3, we get $F_2 = B_{(p+\frac{n}{2}) \mod n}$, which finds neighbor bucket B_3 for B_0, B_4 for B_1, ..., and B_2 for B_5. Thus, neighbors for buckets will be: $\mathcal{N}(B_0) = \{B_5, B_1, B_3\}$; $\mathcal{N}(B_1) = \{B_2, B_0, B_4\}$; $\mathcal{N}(B_2) = \{B_3, B_1, B_5\}$; $\mathcal{N}(B_3) = \{B_4, B_2, B_0\}$; $\mathcal{N}(B_4) = \{B_5, B_3, B_1\}$; and $\mathcal{N}(B_5) = \{B_6, B_4, B_2\}$. □

The example above illustrates that the functions used to define the neighbors ensure that if B_p is a neighbor of B_q, then B_q is a neighbor of B_p. Since the neighbor relationship to buckets is symmetric, we can represent the set of buckets and their neighborhood in the form of a graph $\mathcal{G} = (V, \mathcal{E})$ with the set $V = \{0, \ldots, n-1\}$ of vertices corresponding to the n buckets. The set

[5] We will experimentally show the selection of d in Experiment 5 in §8.
[6] Note that in this case n must be even.

Algorithm 4: VEIL-O:Graph Creation (GC).

Inputs: $\mathcal{B}$:Set of buckets n generated using $\mathcal{BC}$. d: degree of the graph
Outputs: $\mathcal{B}$: the set of n buckets; $\mathcal{M}$: A adjacency matrix of a d-regular (undirected) graph

Function GC($\mathcal{B}, d$) begin
  $\mathcal{M} \leftarrow 0$ ▷ Initialise matrix to 0
  for $p \in [0, n-1]$ **do**
    for $j \in [1, d]$ ▷ *Assigning d neighbors to* p^{th} *bucket* **do**
      $B_q \leftarrow F_j(B_p)$, $\mathcal{M}[p][q] \leftarrow 1$
  return $\mathcal{B}, \mathcal{M}$

Algorithm 5: VEIL-O:Maximum Overlap Size Determination (MOD).

Inputs: $\mathcal{B}$: the set of n buckets; $\mathcal{M}$:Undirected graph; d: degree of graph
Outputs: δ_{MAX}: Maximum overlapping size

Function MOD($\mathcal{B}, \mathcal{M}, d$) begin
  $B_{max}^{\mathcal{N}(B_p)} \leftarrow$ *find_largest_neighbor_of_full_buckets*($\mathcal{B}$)
  $L_{max}^{\mathcal{N}(B_p)} \leftarrow$ *compute_bucket_size*($B_{max}^{\mathcal{N}(B_p)}$)
  $B_{min} \leftarrow$ *find_smallest_buckets*($\mathcal{B}$)
  $L_{min} \leftarrow$ *compute_bucket_size*(B_{min})
  $\delta_1 \leftarrow \ell_b - L_{max}^{\mathcal{N}(B_p)}$, $\delta_2 \leftarrow \lfloor \frac{\ell_b}{d} \rfloor$, $\delta_3 \leftarrow \lfloor \frac{\ell_b - L_{min}}{d} \rfloor$
  return $\mathsf{MIN}\{\delta_1, \delta_2, \delta_3\}$

$\mathcal{E} = \{\langle B_p, B_q \rangle | B_p \in \mathcal{N}(B_q)\}$ are edges. Note that the resulting graph $\mathcal{G}$ is d-regular graph. Fig. 3 illustrates the resulting d-regular graph for the buckets in Example 5.2.

After constructing the d-regular graph for a given set of input buckets, VEIL-O next determines the maximum number of records that can be lent/borrowed, *i.e*, the weight of each edge denoted with δ; the direction of the edges, *i.e.*, which bucket lends/borrows; the labels, *i.e.*, the records that will be shared by neighboring buckets. We will illustrate this in the following steps.

STEP 2. MAXIMUM OVERLAP DETERMINATION (MOD). [Algorithm 5] This step finds an initial value (upper bound) of the overlapping size, *i.e.*, the weight of the edges in the graph, denoted as δ. VEIL-O determines the limiting value for δ as follows:

(1) For each bucket $B_p \in \mathcal{B}$ such that $|B_p| = \ell_b$ (*i.e.*, full buckets), it must be the case that $\delta \leq \ell_b - L_{max}^{\mathcal{N}(B_p)}$, where $L_{max}^{\mathcal{N}(B_p)}$ denotes the number of records in the largest neighbor of B_p. More specifically, $L_{max}^{\mathcal{N}(B_p)} = \mathsf{MAX}_{B_q \in \mathcal{N}(B_p)} |B_q|$. To see this, suppose B_q is the largest neighbor of B_p. Then it can only borrow no more than $\ell_b - |B_q|$ records and this limits δ from being larger.
(2) It always holds that $\delta \leq \frac{\ell_b}{d}$, as in a d-regular graph, each node has d edges, *i.e.*, each bucket has d neighbors, and each pair of neighboring buckets has to share an equal number of records.
(3) If L_{min} is the size of the smallest bucket in $\mathcal{B}$, then the overlapping size is at most $\delta \leq \lfloor \frac{\ell_b - L_{min}}{d} \rfloor$. To construct a d-regular graph successfully, the smallest bucket must borrow records from each of its d neighbors. Moreover, it must borrow an equal number of records from each of its neighbors.

The maximum possible overlapping size is then determined by taking the minimum of the values in (1)-(3). Note that the overlapping size is not final and can be adjusted in Step 3.

Algorithm 6: VEIL-O:Edge Direction Determination (EDD)

Inputs: $\mathcal{B}$: A set of n buckets, d: degree of graph; $\mathcal{M}$:Adjacency matrix of the d-regular graph; δ_{MAX}: maximum overlapping size

Outputs: $\mathcal{B}$; Set of n buckets; $\overline{\mathcal{M}}$: An adjacency matrix encoding the direction of each edge in $\mathcal{M}$

```
Function EDD(B, M, δ_MAX) begin
    sort(B)                                   ▷ Sort buckets by their sizes in increasing order
    δ ← δ_MAX
    M̄ ← []
    for B_j ∈ B do
        L_j ← compute_bucket_size(B_j)
        N(B_j) ← find_neighbors(B_j, M)
        sort(N(B_j))                          ▷ In decreasing order by sizes
        for B_p ∈ N(B_j) do
            if M̄[p][j]! = 1 and M̄[j][p]! = 1 then
                if L_j + δ < ℓ_b then
                    M̄[p][j] = 1, L_j ← L_j + δ    ▷ Assign an edge between B_p and B_j
                else
                    M̄[j][p] = 1                   ▷ Assign an edge between B_j and B_p
                    L_p ← compute_bucket_size(B_p)
                    if L_p + δ > ℓ_b then
                        reduce δ
    return B, M̄, δ
```

Example 5.3. In Fig. 3, the overlapping sizes δ determined using the three rules are 8, 6, and 2, respectively, the initial value of $\delta = 2$.

STEP 3. EDGE DIRECTION DETERMINATION (EDD). [Algorithm 6] To determine the directions of edges, VEIL-O starts from the bucket in $\mathcal{B}$ with the smallest size and proceeds in increasing order of their sizes. This is because buckets with fewer records are less likely to get full, and thus should borrow as many records as possible from their neighbors. For each bucket that is not full, it must borrow records starting from its neighbor with the maximum size and continue doing so from other neighbors in decreasing order of bucket size. This is to maximize the overlapping size δ, as buckets with more records are more likely to become full to restrict δ, so they should lend values whenever possible. The algorithm creates an adjacency matrix $\overline{\mathcal{M}}$ containing the directions for each edge in $\mathcal{M}$.

Example 5.4. Continuing Example 5.3 ($\delta = 2$, $\ell_b = 20$, $d = 3$), the Algorithm EDD generates $\overline{\mathcal{M}}$ as shown in Fig. 3. For instance, for bucket B_1 with neighbors B_0 and B_2, $\overline{\mathcal{M}}$ shows edges from both B_0 and B_2 to B_1. Thus, B_1 borrows δ (which is 2) records from each of these buckets.

Step 4: Adding fake values. Once edge directions have been determined, for each bucket B_p, we add appropriate fake records to ensure that all buckets are equisized. Let L_p^{home} refer to the number of records in B_p when it was created by $\mathcal{BC}$. For each neighbor $B_q \in \mathcal{N}(B_p)$ such that there is an incoming edge from B_q to B_p, the bucket B_p borrows δ records. Let there be L_p^{in} such neighbors. To ensure that bucket B_p has ℓ_b records associated, VEIL-O adds $L_p^{fake} = \ell_b - (L_p^{home} + \delta L_p^{in})$ fake records to B_p. The resulting set of buckets with fake values added to them is denoted with $\mathcal{B}^{padded}$.

Example 5.5. Based on $\overline{\mathcal{M}}$ as shown in Fig. 3, B_1 adds 20 - (12 + 2×2) = 4 fake records. Likewise, B_3 adds 2, and B_5 adds 4 fake records. Thus, 12 fake records are added in total. Contrast this with the disjoint strategy that would have required 32 fake records to be added in total.

Algorithm 7: Label Creation (LC)

Input: $\mathcal{B}_f$: A set of buckets representing the vertices of a d-regular graph;
$\overline{M}$: Adjacency matrix with dir for each edge in the graph;
δ : The weight of each edge in the graph
Output: $\mathcal{L}$: a map of labels for each edge in the graph.

```
L ← ∅
for B_p ∈ B_f do
    for k ∈ [1, ..., n] do
        if M̄[p][k] is 1 then
            labels ← ∅
            for ℓ ∈ [1, δ] do
                labels ← labels ∪ B_p^<[(k − 1)δ + ℓ]
            L[(p, k)] ← L[(p, k] ∪ labels
            L[(k, p)] ← L[k, p] ∪ labels
```

Step 5: Label Creation (LC). For each edge (v_p, v_q) in the graph $\mathcal{G}$, we next determine exactly which records are borrowed/lent between neighbors. Let $B_q = F_j(B_p)$ be a neighbor of B_p such that B_p borrows from B_q. Furthermore, let $F_k(F_i(B_p)) = B_p$ (*i.e.*, B_p is the k^{th} neighbor of B_q). Consider B_q that contains $L_q^{home} + L_q^{fake}$ records. Henceforth, we will consider B_q to be an ordered set of $L_q^{home} + L_q^{fake}$ records and denote it as $B_q^<$.[7] When there is no ambiguity we will still continue to refer to it as B_q for notational simplicity. We will denote the i^{th} record in a bucket $B_p^<$ as $B_p[i]$. Thus, when the bucket B_p borrows δ records from a bucket B_q, it does so based on the ordering of the records in B_q. In particular, it borrows records $B_q[(k-1)\delta+1], B_q[(k-1)\delta+2], \ldots, B_q[(k-1)\delta+\delta]$ which are added to the $label[(v_q, v_p)]$. Note that the above strategy ensures that no records of bucket B_q are in the label for more than one neighboring bucket, thereby ensuring that intersection of any three neighboring buckets are always empty.

Example 5.6. Consider bucket B_3 in Fig. 2 where B_3 borrows from each of its neighbors B_0, B_2 and B_4 with $\delta = 2$. Since $B_3 = F_3(B_0)$, $B_3 = F_1(B_2)$ and $B_3 = F_2(B_4)$, it will borrow the $B_0[5]$ and $B_0[6]$ from B_0. Likewise it will borrow $B_2[1]$ and $B_2[2]$ from B_2. Finally, it will borrow $B_4[3]$ and $B_4[4]$ from B_4. Thus, labels for $\mathcal{L}[(3,0)] = \mathcal{L}[(0,3)] = \{B_0[5], B_0[6]\}$; $\mathcal{L}[(3,2)] = \mathcal{L}[(2,3)] = \{B_2[1], B_2[2]\}$; $\mathcal{L}[(3,4)] = \mathcal{L}[(4,3)] = \{B_4[3], B_4[4]\}$; □

Step 6: Well-Formed Bucket Creation (WFBC). Once labels have been generated for every two neighboring buckets, the well-formed overlapping buckets are generated as follows: for each bucket $B_p \in \mathcal{B}^f$, for each of its neighbor $B_q \in \mathcal{N}(B_p)$, if $dir(B_q, B_p)$ is 1 (*i.e.*, B_p borrows from B_q), we add to B_p all records in $\mathcal{L}[(q,p)]$ which correspond to the set of records B_p borrows from B_q. Note that if buckets B_p and B_q are neighbors, they will contain δ common records.

5.3 Veil-O: Outsourcing and Querying

Outsourcing and querying in Veil-O is identical to that in Veil. First, all real and fake tuples in $\mathcal{B}^f$ are shuffled, encrypted and outsourced as a set of encrypted records along with the RID. Then, to create a multimap index, we construct for each bucket B_p a map $Mmap[B_p]$ consisting of RID of the records in B_p. As before, the encrypted record set, along with the $Mmap[B_p]$ for each bucket B_p, is outsourced to the server.

To execute a query for a keyword k, first, the keyword is mapped to the f buckets as in Veil. For each such bucket B_p, the server retrieves the RID of every record in the multimap index $Mmap[B_p]$

[7]The exact ordering does not matter, it could be any random ordering

and uses these RIDs to retrieve data from the encrypted record store which are then returned to the client.

Note that unlike VEIL, in VEIL-O the same RID may appear in more than one $MMap[B_p]$ since VEIL-O allows for overlap between buckets. Since we restrict a RID to be replicated in at most two buckets, the additional number of RIDs replicated equals $\frac{d\delta|\mathcal{B}|}{2}$. Given a pointer is four bytes, the number of bytes of overhead can be computed as $2d\delta|\mathcal{B}|$ bytes for savings of $\frac{d\delta|\mathcal{B}|}{2}$ fake records. Such an overhead remains significantly small even when record sizes are relatively small, but are much more pronounced when individual records are large in which case, the overhead remains a small fraction of the savings.

5.4 Discussion

VEIL-O, like VEIL, remains secure under VSR (The proof of VEIL-O is presented in the extended version [25]). Thus, an adversary cannot determine which key the user is accessing based on how queries are processed. However, in VEIL-O, since the maximum overlap between buckets is data dependent, the adversary could distinguish between databases outsourced based on observed overlap (and, thus, the effective *SA* achieved) by VEIL-O. A slight modification can, however make VEIL-O achieve indistinguishability. In the modified version, the desired overlap between buckets is specified by the user (independent of the database) - let us refer to it as $O_{desired}$. VEIL-O learns the maximum overlap that can be supported (given random neighbor assignment) as in the original protocol - let us denote it by O_{max}. If $O_{desired} \leq O_{max}$ we simply revert back to $O_{desired}$ and continue with the rest of the VEIL-O to generate buckets. Alternatively, if $O_{desired} > O_{max}$, then for each pair of neighbor buckets B_i and B_j such that B_j is a receiver and B_i a lender of records, we perform the following check. Let f_{B_j} be the number of RIDs pointing to fake records in B_j. We first replace those fake records by RIDs to additional records from B_i, thereby increasing the amount of overlap between B_i and B_j. If $O_{desired} - O_{max}$ is greater than the number of fake records f_{B_j}, to ensure overlap of $O_{desired}$, we shift $(O_{desired} - O_{max}) - f_{B_j}$ real records from B_j to the stash, thereby creating space for equivalent number of RIDs to be borrowed from B_i to B_j. Note that in the modified VEIL-O strategy, the number of overlapping records between any two buckets are equal to $O_{desired}$ and independent of the database being indexed. As a result, the adversary cannot gain information about the database being indexed from the data representation, query access patterns, and volume in addition to being unable to learn query keywords. We refer to the modified version of VEIL-O as VEIL-O′. Note that VEIL-O′ may have a higher stash compared to VEIL-O but the effective *SA* it achieves could be even better compared to VEIL-O since it could reduce number of fake records stored on the server side. In the experiment section, we will study impact of the above modified strategy (to make VEIL-O secure based on the security model used in [38, 48]) on increase in stash.

6 SUPPORTING DYNAMIC CHANGES

So far, similar to [38, 48], we have discussed VEIL and VEIL-O under a static setting where the database to be outsourced is pre-known. While a full development and evaluation of dynamic operations in VEIL is outside the scope of this paper, we briefly discuss how VEIL can be extended to support dynamic operations. Indeed, the flexibility and ease of VEIL in supporting dynamic operations compared to other prior work is one of its advantages as will be discussed in §7. We focus on the case of insertion though the discussion below which can be extended to updates and deletions as well. In VEIL, adding new data does not require re-execution of the bucketization strategy as long as the value of L_{max} after insertions does not exceed $L_{max} \times QA$. Insertion can be supported by retrieving the set of records in the f buckets corresponding to the key for the record being inserted, replacing one of the fake tuples (if present) in the buckets by the newly

inserted tuples, re-encrypting the records in the buckets, and re-outsourcing the modified buckets. Of course, if the f buckets do not contain any fake tuples (and hence have no residual capacity to store more data), the newly inserted tuple is stored in a stash, as would have been the case had the buckets become full in the static situation when constructing the original buckets. The above insertion strategy would continue to work, as long as, the new value of L_{max} after insertions, say L'_{max}, remains below $QA \times L_{max}$.[8] As L_{max} increases, since we are not changing the fanout f or the bucket size, effectively the QA value reduces which, in turn, increases the probability of a record having to be stored in the stash. When L'_{max} goes above $QA \times L_{max}$, we can either continue to map new records to stash (increasing client side overhead) and/or reorganize the data. While we do not conduct a formal analysis or experimental validation, we believe that the strategy above will allow a large number of insertions between reorganizations since L_{max} can be expected to grow slowly. Furthermore, a user can begin with a larger value of QA to control how often reorganization is required.

7 RELATED WORK

Volume hiding (VH) as a security goal, has gained attention starting with the seminal work by Kamara *et al* in [29, 30] that utilizes a multimap data structure for encrypted keyword search. VH is easier to achieve when access pattern (AP) hiding technque, such as ORAM [19, 20, 36, 47], is already being used. Given prohibitive nature of AP hiding, recent work [1, 17, 29, 30, 38, 44] has explored VH without requiring ORAM. Veil falls into this category of work.

Below, we focus on two techniques most relevant to Veil: dprfMM [38] that uses a cuckoo hash based strategy, and XorMM [48] that uses the XOR filter to support volume hiding in key-value stores. We compare Veil and its variants to these approaches experimentally in the following section. In the remainder of this section, we focus on a qualitative comparison of the schemes in terms of several criteria including support for dynamic updates, security offered, applicability as an indexing technology, and expected performance. We also present a detailed related work in the extended version [25].

Supporting Dynamic Changes. Different approaches, viz., XorMM, dprfMM and Veil offer different levels of ease in extending them to support dynamic operations (insertions, updates, deletion). In particular, Veil and dprfMM offer significant flexibility and ease to support dynamic operations compared to XorMM as shown in §6. DprfMM, like Veil, also offers flexibility and can allow periodic reorganization by storing new data in the cuckoo hash and/or stash. In contrast, XorMM cannot support insertions, as it requires the XOR filter to be recontructed every time there is an insertion. This is because the XOR filter is constructed through a sequential process that creates dependencies between cell representations, *i.e.*, a record inserted later into the filter may depend upon the cipher representation of the record inserted earlier. As a result, when new data has to be inserted (whether or not it results in change to L_{max}), the filter has to be recomputed from scratch. The challenge of supporting dynamic operations, coupled with the flexibility to choose QA and SA parameters to control the overheads, are the primary advantages of Veil.

Security Offered. We developed Veil and its variants under the security goal VSR (§2.2) that roughly corresponds to ensuring that an adversary cannot differentiate between query keywords. One could also consider a stronger security model that also ensures indistinguishability amongst databases (§5.4) used in [38, 48]. We note that Veil is already secure against the enhanced security model used in [38, 48], that we refer to as *indistinguishability*, - we show this formally in the extended version of the paper [25]. Veil-O, as discussed in §5.4, however, while ensuring VSR, does

[8] In fact, it can continue to work beyond that except that all new records after the point of increase in L'_{max} beyond $QA \times L_{max}$ will need to be stored locally in the stash.

not ensure indistinguishability directly. It can, however, be modified with a simple post-processing step to offer the same level of security as [38, 48], with the post processing step causing a slight increase in the stash storage.

Probabilistic versus Guaranteed Success. Given a database $\mathcal{D}$, dprfMM and Veil are always successful in forming the multimap representation on the server, while XOR filters are probabilistic, and there is a chance (however, small) that the technique may fail to create an appropriate XOR filter [48]. Hence the XOR filter is not guaranteed to create a multimap to support volume-hiding in all situations.

Primary versus Secondary Index. The basic Veil approach can outsource records in the form of buckets containing encrypted data records, or in the form of buckets with RIDs that point to an encrypted record store. Likewise, dprfMM can also be extended to store either RIDs or the records. In contrast, xorMM can only be used to store records and cannot easily be extended to store RIDs, since the security requirement of the XOR filter requires data in the filter to be encrypted. Thus, storing RIDs in the filter will introduce additional round of communication between client and server for the client to decrypt the RID and subsequently ask the server to retrieve the records corresponding to those RIDs. This will significantly increase communication costs. Also, the RIDs returned by the server may point to some record that does not exist in the data storage, and such information is revealed to the server, which directly leaks the volume. Veil-O, in contrast, provides benefit (reduced *SA*) when used as a secondary index with RIDs that are referenced to retrieve data from the encrypted store.

Expected Performance Comparison. While we conduct a thorough experimental comparison between strategies, we make a few observations about expected performance of different strategies.

• In terms of QA and SA, xorMM (if it can be successfully created) overshadows dprfMM since it offers fixed QA and SA or 1 and 1.23 as compared to 2 and 2.6. Veil supports a tunable QA and SA values - by appropriately choosing QA and SA, it can outperform xorMM.

• In terms of client storage, all schemes store very little meta data (*e.g.*, Veil needs to store just two numbers - fanout and the number of buckets) in addition to the stash containing some overflow data records. Stash is also stored in dprfMM which, as experiments will show, is higher than what is stored in Veil. xorMM does not maintain a client side stash, but as a result, has a non-zero probability of failing to form.

8 EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of Veil. We study:

- Impact of user-specified parameters (Storage Amplification *SA*, Query Amplification *QA*, and fanout f) on Stash Ratio *SR*. Based on Equation 2, we make the following observations:
  (1) For fixed *QA* and f, as *SA* increases, while the bucket size remains constant, the number of buckets increases linearly, and hence lower the expected SR (**Exp 2**).
  (2) For fixed *SA* and f, as *QA* increases, bucket size increases linearly, but the number of buckets decreases proportional to $\frac{1}{QA}$. For fixed *QA* and SA, increasing f causes bucket size to decrease, but increases the number of buckets. We explore how *SR* changes as a function of *QA* and f experimentally in **Exp 1** and **Exp 3**.
  (3) Since [38] already achieves a *QA* and *SA* of 2 with a small stash (viz., *SR*), we focus on *QA* and *SA* in the range 1 to 2.
  (4) Since *QA* influences the number of records retrieved from the server, it is desirable to keep it as close to 1 as possible while still ensuring a small *SR* and an *SA* below 2. This would be strictly better than the state-of-the-art.
- Effect of overlapping strategy Veil-O on reducing *SA* (**Exp 5**).

- Comparison of Veil with existing approaches, including dprfMM [38] and XorMM [48], in terms of *SA*, *QA*, and *SR* (**Exp 7**).
- Veil and Veil-O performance on larger data sets. (**Exp 8** and **9**).
- Setup time and query time for Veil and Veil-O and compare them with those for state-of-art (Exp **10-12**).

8.1 Setup

We evaluate Veil using the LineItem table from the variant of TPC-H dataset entitled TPC-H-SKEW [10]. TPC-H-SKEW generates the same tables, except that it allows us to control the skewness of the data using a "skew factor", denoted by z. We show the impact of changing z values in Table 1 by listing the L_{max} and the number of keys generated in the LineItem table for a scale factor 1 (which corresponds to 6M records in the LineItem table). As shown in Table 1, as z increases, the data gets more skewed with larger L_{max}. When z is zero, the data generated is non-skewed as in the original TPC-H dataset. We vary z from 0 to 1 and use a default of 0.4, which is consistent with real-world datasets that are most likely to be skewed. To generate key-value datasets, we use the Partkey (PK) column as the keys and use values of all other columns as their associated values. For most of our experiments, we use the TPC-H with scale factor 1 (*i.e.*, the datasize is 6M). We also include the results with the scale factor of 6 to see how Veil scales to a larger dataset. Our experiments were conducted on a MacBook Pro equipped with an M1 Pro processor and 32 GB of RAM, running the macOS Monterey operating system. We utilized SHA-256 [39] as the hash function and employed AES encryption (CBC mode) [12] for our symmetric encryption scheme.

Table 1. TPC-H datasets with different skew factor z

Skew Factor z	0	0.2	0.4	0.6	0.8	1
L_{max}	57	63	357	5,234	131,749	370,760
# of keys	200,000	200,000	200,000	200,000	81,153	199, 919

Comparison Metrics: We evaluated our approaches and the state-of-art approaches using the metrics defined in §1, including *query amplification* (*QA*), *storage amplification* (*SA*), and *stash ratio* (*SR*). To measure the physical storage overhead at the local side and at the server, we introduced the following two additional metrics:
Client Storage Amplification, denoted by *CSA*: that is the ratio of the size of total physical storage required at the local side to implement a given volume hiding scheme over the size of encrypted representation of the dataset. For instance, in Veil, clients need to store values of fanout and number of buckets (2 integers) in addition to the records in the stash. Thus, the *CSA* corresponds to the storage requirement of these divided by the encrypted representation of the data.
Server storage amplification, denoted by *SSA*: the ratio of the size of physical storage required at the server side to implement a given volume hiding scheme to the the size of encrypted representation of the dataset. For instance, in Veil, the server needs to store the encrypted record set as well as the multimap index §3.2). Thus, *SSA* corresponds to the ratio of the total storage needed at the server by the size of the encrypted record set.

8.2 Evaluation of Parameters on Stash Ratio

We conducted an ablation study of three factors, — the fanout f, the storage amplification *SA*, the query amplification *QA*, and the skewness factor z by fixing three others in each experiment to evaluate the impact of each factor on the overall performance of Veil. For each generated database, we ran Veil 5 times to map keys to buckets randomly and computed an average stash ratio (*SR*).

The outcomes of our experiments are illustrated in Fig. 5, Fig. 6, Fig. 7, and Fig. 8. We also recorded the average stash size, shown as the labels in Fig. 5, Fig. 6, Fig. 7, and Fig. 8.[9]

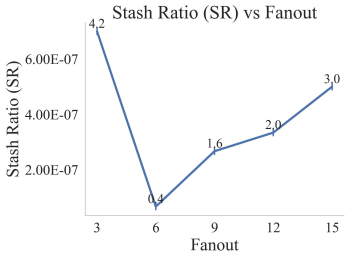


Fig. 5. Impact of f (SA=1.2, QA=1)**

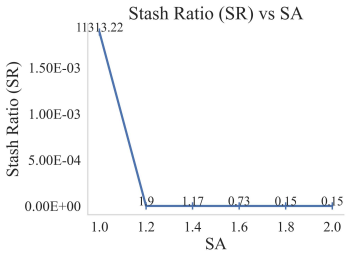


Fig. 6. Impact of SA (QA=1, f=6)**

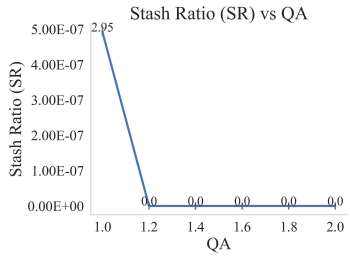


Fig. 7. Impact of QA (SA=1.2, f=6)**

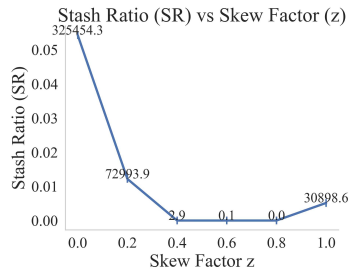


Fig. 8. Impact of z (f = 6)**

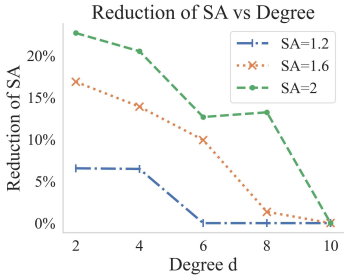


Fig. 9. Impact of d (QA=1, f = 6)

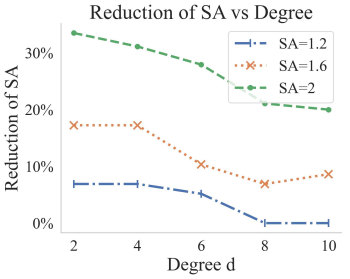


Fig. 10. Impact of d on 36M Data

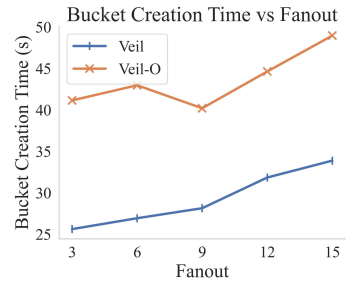


Fig. 11. Bucket Creation Time vs f

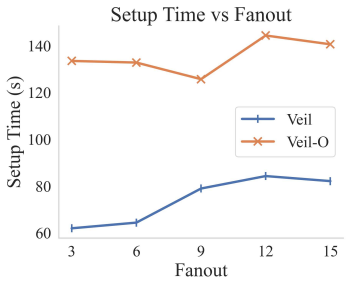


Fig. 12. Setup Time vs Fanout

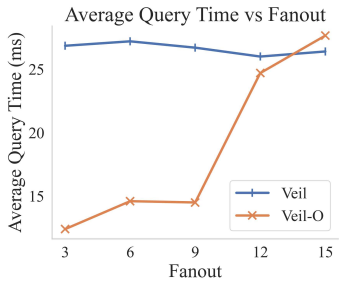


Fig. 13. Average Query Time vs f

Exp 1: Impact of fanout f. Fig. 5 presents SR as a function of f with fixed values of QA and SA, where QA is 1 and SA is 1.2.[10] The results show that SR decreases as the fanout f increases. This is because a larger f provides more "choices" when selecting buckets for records, and a record is placed into the local stash only if all the chosen f buckets are full. However, when the fanout reaches a certain threshold (*e.g.*, 6 in Fig. 5), this benefit is not observed. SR may increase if the

[9]** Numbers on top of the lines in the figures represent the average number of records in the stash. We will have similar numbers in all plots with Stash Ratio (Fig. 5, Fig. 6, Fig. 7, Fig. 8, and Fig. 14a).

[10] QA is more important in a secure outsourcing system compared with the low price for storage in a public cloud.

fanout f is further increased. This is because, at this stage, the bucket size, according to Equation 2, becomes small, making the buckets more likely to become full during the random bucketing process. *In the rest of the experiments, we will use 6 as an optimal value of f.*

Exp 2: Impact of storage amplification *SA***.** Fig. 6 illustrates the relationship between the stash ratio *SR* and the storage amplification *SA*, with fixed values of $QA = 1$ and fanout $f = 6$ (the optimal value according to **Exp** 1). The results show that when *SA* is 1, Veil uses a small stash, with *SR* to be approximately 0.002. When *SA* gets larger (*e.g.*, $SA \geq 1.2$), the *SR* reduces to nearly zero, with only a small number of values (less than 5) stored in the local stash. This is because when *QA* and fanout f are fixed, the bucket size ℓ_b remains constant according to Equation 2. Consequently, an increase in *SA* leads to the creation of more buckets, which in turn results in a reduced *SR*.

Exp 3: Impact of *QA***.** Fig. 7 illustrates the relationship between the stash ratio *SR* and the query amplification *QA*, with fixed values of $SA = 1$ and fanout $f = 12$. The results show that *SR* is considerably small and is even reduces to 0 when *QA* is greater than 1.2.

Exp 4: Impact of skew factor z**.** Fig. 8 illustrates the relationship between the stash ratio *SR* and the skew factor z, while fixing *SA* of 1.2, *QA* of 1, and f of 12. With the skew factor z increasing, the maximum key size L_{max} and bucket size increases. *SR* also decreases, since the bucket size is larger in comparison to the majority of other keys in the dataset, and there is a high likelihood of available space for a record within the buckets, resulting in a reduced probability of a record being placed in the stash. However, as z continues to increase, reaching a certain threshold, such as 0.8 in Fig. 8, the stash expands due to the significant disparity between L_{max} and the sizes of most keys in the dataset. With a fixed *SA*, the number of buckets decreases in accordance with Equation 2, causing the buckets corresponding to the keys with a large number of records to be more likely to reaching full capacity. Consequently, records mapped to these buckets are more likely to be placed into the stash.

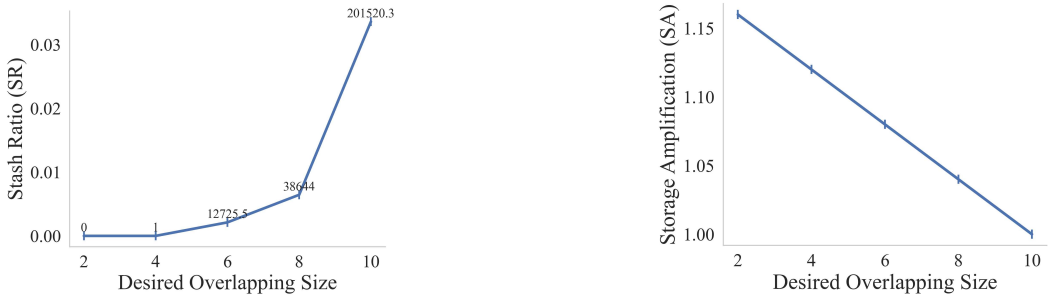
8.3 Padding Strategy Evaluations

Exp 5: Impact of the degree d **when creating a** d**-regular graph.** We fixed *SA* to 1.2, *QA* to 1, and varied the degree d to 2, 4, 6, 8, and 10 to evaluate the impact of d on the reduction of *SA* in Veil-O. The results in Fig. 9 indicate that when d is 2, the *SA* is the smallest, which is 1.08. As d increases, the SA increases, since each bucket has more "neighbors", and the probability of two large buckets being neighbors gets higher, making the number of common records for each two neighbored buckets (*i.e.*, the weights over edges in the d-regular graph) smaller. When the degree d is increased to 6, the overlapping size becomes zero, and the *SA* is equal to the desired value.

Exp 6: Impact of desired overlapping size in Veil-O′. We fixed *SA* to 1.2, *QA* to 1, degree d to 2, and varied the desired overlapping size to 2, 4, 6, 8, and 10 to evaluate the impact of the desired overlapping size on both *SR* and *SA*. Results are presented in Fig. 14. The results show that the *SA* decreases with an increasing desired overlapping size, as more common records are shared between neighboring buckets. Meanwhile, more records are placed into the stash due to the mandatory sharing of common records.

8.4 Comparison with the State-of Arts

Exp 7: Comparisons with dprfMM and XorMM. To compare Veil with dprfMM [38], we fixed the fanout f, the desired *SA* and *QA* of Veil to 12, 1.2 and 1, respectively, and set the *SA* and *QA* of dprfMM [38] to 2 and 2.6, in accordance with their experimental configurations. We also included XorMM [48], which has a fixed *SA* of 1.23 and *QA* of 1, for comparison. We used two different TPC-H datasets with 6M records, with varying skew factor z to be 0 (*i.e.*, the original TPC-H dataset) and 0.4, respectively. The results are presented in Table 2, and 3, respectively. The results demonstrate that Veil outperforms dprfMM when processing skewed datasets, utilizing a



(a) ** *SR* vs desired overlapping

(b) *SA* vs desired overlapping

Fig. 14. Impact of the desired overlapping size in VEIL-O′.

significantly smaller stash, almost zero when z is 0.4. Additionally, the overlapping strategy VEIL-O reduces *SA* from 1.2 to 1.12 when z is 0.4, as the bucket sizes for skewed datasets are larger in comparison to the number of records for most keys. However, this is not observed for non-skewed data which results in a higher stash ratio (*SR*).

Table 2. Comparisons on 6M non-skewed TPC-H dataset ($z = 0$).

	QA	*SA*	*SR*	*CSA*	*SSA*
dprfMM [38]	2	2	5.949E-5	1.467E-5	1.000
dprfMM [38]	2	2.6	0	1.287E-9	1.000
XorMM [48]	1	1.23	-	2.18E-8	1.000
VEIL	1	1.2	0.056	0.0229	1.020
VEIL-O (d= 2)	1	1.2	0.056	0.0033	1.041
VEIL	1	2	4.566E-5	1.552E-5	1.027
VEIL-O (d= 2)	1	2	4.649E-5	2.066E-6	1.050

Note: The *CSA* and *SSA* of dprfMM are low because dprfMM stores two tables that contain $2|\mathcal{D}|$ or $2.6|\mathcal{D}|$ records in total. The SR of VEIL and VEIL-O can be high when QA is optimal and SA is low (*e.g.*, when QA is 1 and SA is 1.2), as more collisions happen when allocating records into buckets.

Table 3. Comparisons on 6M skewed TPC-H dataset ($z = 0.4$)

	QA	*SA*	*SR*	*CSA*	*SSA*
dprfMM [38]	2	2	5.000E-7	1.411E-5	1.000
dprfMM [38]	2	2.6	0	1.287E-9	1.000
XorMM [48]	1	1.23	-	2.18E-8	1.000
VEIL	1	1.2	4.26E-07	2.71E-7	0.018
VEIL-O (d = 2)	1	1.2	3.67E-07	2.68E-8	0.0314
VEIL	1	2	0	3.477E-9	1.027
VEIL-O (d = 2)	1	2	0	7.524E-9	1.056

8.5 Performace on Large Datasets

To evaluate how our algorithms (VEIL and VEIL-O) handle large datasets, we generated a TPC-H dataset with 36 million records (corresponding to the scale factor of 6 in TPC-H), incorporating a

skew factor (z) of 0.4 and ran experiments to measure stash ratio SR with fixed QA and SA. We also evaluate how changing the degree d in Veil-O impacts the reduction of SA.

Exp 8: Stash Ratio on Large Datasets. We executed Veil on the 36M dataset 10 times to compute an average stash ratio (SR), which was found to be nearly zero, with 1.3 records placed in the stash on average. Specifically, in 7 out of the 10 experiments, the stash size is zero.

Exp 9: Reduction SA in Veil-O on Large Datasets. We ran Veil-O 10 times on the 36M dataset to compute an average practical SA. We varied the degree of the d-regular graph in Veil-O to 2, 4, 6, 8, and 10, and varied the desired SA to 1.2, 1.6, and 2.0. The results are in Fig. 10. The results show that Veil-O effectively reduces the number of fake values, especially when the desired SA is higher.

8.6 Running Time

Exp 10: Bucket creation time. We varied the fanout from 3 to 15 and evaluated the time required for bucket creation, including allocating each record to a bucket and padding. The results of this experiment are in Fig. 11. As the fanout increases, the time required for bucket creation increases for both the basic approach Veil and the overlapping approach Veil-O. This is because more buckets need to be created given user-desired SA and QA thereby increasing the total processing time.

Exp 11: Setup time. We varied the fanout from 3 to 15 at increments of three and evaluated the setup time, *i.e.*, the time taken to create, encrypt, and store the buckets. The results are in Fig. 12. We also take dprfMM [38] and XorMM [48] as a comparison. When SA was set to 2.6, the setup time for dprfMM was 210.09s, which dropped to 155.349s when SA was 2. The setup time for XorMM is 124.11s. The results show that Veil outperforms Veil-O, which can be attributed to the additional time Veil-O takes to generate a d-regular graph. Also, Veil required less setup time than XorMM, while Veil-O exhibited a setup duration comparable to that of XorMM. Furthermore, both Veil and Veil-O demonstrated shorter setup times than dprfMM.

Exp 12: Query time. To evaluate the time taken for query execution, we varied the fanout f from 3 to 15, with increments of three, and ran 20 queries to compute an average query time for each query. We also included dprfMM and XorMM as a comparison. The average query time for dprfMM is 201ms when SA is 2.6 and 63ms when SA is 2. The average query time for XorMM is 33.1ms. The results are shown in Fig. 13. Results show that regarding query time, Veil and Veil-O outperform both dprfMM and XorMM. Also, note that the query time for Veil is similar when the fanout varies. That is because fanout only determines the number of buckets to retrieve for each query and doesn't have any impact on the number of records retrieved. When processing a query, both Veil and Veil-O get the desired bucket-ids using Map and retrieve all buckets together.

9 CONCLUSION

This paper proposed Veil to prevent volume leakage in encrypted search. Veil selects multiple buckets for each key and employs a greedy strategy to distribute key-value pairs into different buckets. With the created buckets, Veil applies various strategies to add fake values to the created buckets to pad them to an equal size. This increases SA. To reduce the number of fake values that need to be added to buckets, we proposed Veil-O which allows sharing values among buckets by creating regular graphs with vertices corresponding to the buckets and edges connecting two vertices if the corresponding buckets share values. This significantly improves SA. We verified this using evaluations and showed that both of our approaches are strictly better than the state of art.

ACKNOWLEDGEMENTS

This work was supported by NSF Grants No. 2032525, 1545071, 1527536, 1952247, 2008993, 2133391, and 2245372. Vishal Chakraborty was also supported by the HPI Research Center in Machine Learning and Data Science at UC Irvine. The authors thank the reviewers for their feedback.

REFERENCES

[1] Ghous Amjad, Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. 2021. Dynamic Volume-Hiding Encrypted Multi-Maps with Applications to Searchable Encryption. (2021).

[2] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. 2016. SMCQL: secure querying for federated databases. *arXiv preprint arXiv:1606.06808* (2016).

[3] Dmytro Bogatov, George Kollios, and Leonid Reyzin. 2019. A comparative evaluation of order-revealing encryption schemes and secure range-query protocols. *Proceedings of the VLDB Endowment* 12, 8 (2019), 933–947.

[4] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. 2014. Dynamic searchable encryption in very-large databases: Data structures and implementation. *Cryptology ePrint Archive* (2014).

[5] Anrin Chakraborti and Radu Sion. 2017. Sqoram: Read-optimized sequential write-only oblivious RAM. *arXiv preprint arXiv:1707.01211* (2017).

[6] Anrin Chakraborti and Radu Sion. 2018. ConcurORAM: High-throughput stateless parallel multi-client ORAM. *arXiv preprint arXiv:1811.04366* (2018).

[7] Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. 2022. Towards Practical Oblivious Join. *Proceedings of the 2022 International Conference on Management of Data* (2022).

[8] Melissa Chase and Seny Kamara. 2010. Structured encryption and controlled disclosure. In *Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16*. Springer, 577–594.

[9] Guoxing Chen, Ten-Hwang Lai, Michael K. Reiter, and Yinqian Zhang. 2018. Differentially Private Access Patterns for Searchable Symmetric Encryption. *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications* (2018), 810–818.

[10] Alain Crolotte and Ahmad Ghazal. 2012. Introducing skew into the TPC-H benchmark. In *Topics in Performance Evaluation, Measurement and Characterization: Third TPC Technology Conference, TPCTC 2011, Seattle, WA, USA, August 29-September 3, 2011, Revised Selected Papers 3*. Springer, 137–145.

[11] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM conference on Computer and communications security*. 79–88.

[12] Joan Daemen and Vincent Rijmen. 1999. AES proposal: Rijndael. (1999).

[13] F Betül Durak, Thomas M DuBuisson, and David Cash. 2016. What else is revealed by order-revealing encryption?. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1155–1166.

[14] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. 2015. Rich queries on encrypted data: Beyond exact matches. In *Computer Security–ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II 20*. Springer, 123–145.

[15] Horst Feistel. 1973. Cryptography and computer privacy. *Scientific american* 228, 5 (1973), 15–23.

[16] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. 2016. TWORAM: Efficient Oblivious RAM in Two Rounds with Applications to Searchable Encryption. In *Annual International Cryptology Conference*.

[17] Marilyn George, Seny Kamara, and Tarik Moataz. 2021. Structured Encryption and Dynamic Leakage Suppression. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 370–396.

[18] Eu-Jin Goh. 2003. Secure indexes. *Cryptology ePrint Archive* (2003).

[19] Oded Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 182–194.

[20] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.

[21] Thomas Mueller Graf and Daniel Lemire. 2020. Xor filters: Faster and smaller than bloom and cuckoo filters. *Journal of Experimental Algorithmics (JEA)* 25 (2020), 1–16.

[22] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. 2019. Encrypted databases: New volume attacks against range queries. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 361–378.

[23] Yu Guo, Cong Wang, Xingliang Yuan, and Xiaohua Jia. 2018. Enabling privacy-preserving header matching for outsourced middleboxes. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*. IEEE, 1–10.

[24] Yu Guo, Chen Zhang, and Xiaohua Jia. 2020. Verifiable and forward-secure encrypted search using blockchain techniques. In *ICC 2020-2020 IEEE international conference on communications (ICC)*. IEEE, 1–7.

[25] Shanshan Han, Vishal Chakraborty, Michael Goodrich, Sharad Mehrotra, and Shantanu Sharma. 2023. Hiding Access-pattern is Not Enough! Veil: A Storage and Communication Efficient Volume-Hiding Algorithm. *arXiv preprint arXiv:2310.12491* (2023).

[26] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *Network and Distributed System Security Symposium*.

[27] David S Johnson. 1973. *Near-optimal bin packing algorithms*. Ph. D. Dissertation. Massachusetts Institute of Technology.
[28] Charanjit S. Jutla and Sikhar Patranabis. 2021. Efficient Searchable Symmetric Encryption for Join Queries. In *IACR Cryptology ePrint Archive*.
[29] Seny Kamara and Tarik Moataz. 2019. Computationally volume-hiding structured encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 183–213.
[30] Seny Kamara, Tarik Moataz, and Olya Ohrimenko. 2018. Structured encryption and leakage suppression. In *Annual International Cryptology Conference*. Springer, 339–370.
[31] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. 2012. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 965–976.
[32] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic Attacks on Secure Outsourced Databases. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016).
[33] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. 2016. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1329–1340.
[34] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 2018. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 297–314.
[35] Kevin Lewi and David J Wu. 2016. Order-revealing encryption: New constructions, applications, and lower bounds. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1167–1178.
[36] Rafail Ostrovsky. 1990. Efficient computation on oblivious RAMs. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*. 514–523.
[37] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. 2014. Blind seer: A scalable private DBMS. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 359–374.
[38] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. 2019. Mitigating leakage in secure cloud-hosted data structures: volume-hiding for multi-maps via hashing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 79–93.
[39] Wouter Penard and Tim van Werkhoven. 2008. On the secure hash algorithm family. *Cryptography in context* (2008), 1–18.
[40] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2016. Arx: A Strongly Encrypted Database System. *IACR Cryptol. ePrint Arch.* 2016 (2016), 591.
[41] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M Hellerstein. 2021. Senate: A Maliciously-Secure MPC Platform for Collaborative Analytics.. In *USENIX Security Symposium*. 2129–2146.
[42] Rishabh Poddar, Stephanie Wang, Jianan Lu, and Raluca Ada Popa. 2020. Practical volume-based attacks on encrypted databases. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 354–369.
[43] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the twenty-third ACM symposium on operating systems principles*. 85–100.
[44] Kui Ren, Yu Guo, Jiaqi Li, Xiaohua Jia, Cong Wang, Yajin Zhou, Sheng Wang, Ning Cao, and Feifei Li. 2020. Hybridx: New hybrid index for volume-hiding range queries in data outsourcing services. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 23–33.
[45] Xuanle Ren, Le Su, Zhen Gu, Sheng Wang, Feifei Li, Yuan Xie, Song Bian, Chao Li, and Fan Zhang. 2022. HEDA: Multi-Attribute Unbounded Aggregation over Homomorphically Encrypted Database. *Proceedings of the VLDB Endowment* 16, 4 (2022), 601–614.
[46] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. 2000. Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE symposium on security and privacy. S&P 2000*. IEEE, 44–55.
[47] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: an extremely simple oblivious RAM protocol. *Journal of the ACM (JACM)* 65, 4 (2018), 1–26.
[48] Jianfeng Wang, Shi-Feng Sun, Tianci Li, Saiyu Qi, and Xiaofeng Chen. 2022. Practical Volume-Hiding Encrypted Multi-Maps with Optimal Overhead and Beyond. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2825–2839.
[49] Qiao Wang, Yu Guo, Hejiao Huang, and Xiaohua Jia. 2018. Multi-user forward secure dynamic searchable symmetric encryption. In *Network and System Security: 12th International Conference, NSS 2018, Hong Kong, China, August 27-29, 2018, Proceedings 12*. Springer, 125–140.
[50] Yilei Wang and Ke Yi. 2021. Secure Yannakakis: Join-Aggregate Queries over Private Data. *Proceedings of the 2021 International Conference on Management of Data* (2021).
[51] Songrui Wu, Qi Li, Guoliang Li, Dong Yuan, Xingliang Yuan, and Cong Wang. 2019. ServeDB: Secure, verifiable, and efficient range queries on outsourced database. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*.

IEEE, 626–637.
[52] Jing Yao, Yifeng Zheng, Yu Guo, and Cong Wang. 2020. Sok: A systematic study of attacks in efficient encrypted cloud data search. In *Proceedings of the 8th International Workshop on Security in Blockchain and Cloud Computing*. 14–20.
[53] Xingliang Yuan, Yu Guo, Xinyu Wang, Cong Wang, Baochun Li, and Xiaohua Jia. 2017. Enckv: An encrypted key-value store with rich queries. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 423–435.
[54] Xingliang Yuan, Xinyu Wang, Cong Wang, Baochun Li, Xiaohua Jia, et al. 2018. Enabling encrypted rich queries in distributed key-value stores. *IEEE Transactions on Parallel and Distributed Systems* 30, 6 (2018), 1283–1297.

Received 15 April 2023; revised 20 July 2023; accepted 23 August 2023