Organizing Records for Retrieval in Multi-Dimensional Range Searchable Encryption

Mahdieh Heidaripour[1], Ladan Kian[1] *, Maryam Rezapour[2], Mark Holcomb[1], Benjamin Fuller[2], Gagan Agrawal[3], and Hoda Maleki[1]
[1]Augusta University {mheidaripour,lkian,mholcomb, hmaleki}@augusta.edu
[2] University of Connecticut. {maryam.rezapour, benjamin.fuller}@uconn.edu
[3]University of Georgia. gagrawal@uga.edu

April 25, 2024

Abstract

Storage of sensitive multi-dimensional arrays must be secure and efficient in storage and processing time. Searchable encryption allows one to trade between security and efficiency. Searchable encryption design focuses on building indexes, overlooking the crucial aspect of record retrieval. Gui et al. (PoPETS 2023) showed that understanding the security and efficiency of record retrieval is critical to understand the overall system. A common technique for improving security is partitioning data tuples into parts. When a tuple is requested, the entire relevant part is retrieved, hiding the tuple of interest.

This work assesses tuple partitioning strategies in the dense data setting, considering parts that are random, 1-dimensional, and multi-dimensional. We consider synthetic datasets of $2, 3$ and 4 dimensions, with sizes extending up to 2M tuples. We compare security and efficiency across a variety of record retrieval methods. Our findings are:

1. For most configurations, multi-dimensional partitioning yields better efficiency and less leakage.
2. 1-dimensional partitioning outperforms multi-dimensional partitioning when the first (indexed) dimension is any size as long as the query is large in all other dimensions except the (the first dimension can be any size).
3. The leakage of 1-dimensional partitioning is reduced the most when using a bucketed ORAM (Demertiz et al., USENIX Security 2020).

1 Introduction

Scientific data is often organized as multi-dimensional arrays [Rus23]. Datasets' size necessitates efficient solutions regarding storage, processing time, and communication. Searchable encryption [SWP00, BW07, BFOR08, CK10, TKMZ13, BHJP14, FVY$^+$17, KKM$^+$22][1] allows a client $\mathbf{C}$ to outsource a database $\mathcal{DB}$[2], to a server $\mathbf{S}$. The client should then be able to retrieve records corresponding to a query $\mathbf{q}$ efficiently from the $\mathcal{DB}$ without the server learning about the contents of $\mathcal{DB}$ or $\mathbf{q}$. This work considers multi-dimensional range queries [MT19, FMC$^+$20, ACF$^+$20, MFTS21, MFET23]. For a number of dimensions ℓ, a database $\mathcal{DB} = (\mathcal{DB}_1, ..., \mathcal{DB}_n)$ is a collection of tuples $\mathcal{DB}_j$ where

$$\mathcal{DB}_j \in \left(\prod_{i=1}^{\ell}[1, m_i]\right) \times \mathcal{R}.$$

where m_i is the maximum value for dimension i. We use the following terms:

*Corresponding author. M. Heidaripour and L. Kian, in alphabetical order, share lead authorship.

[1]Other approaches include multi-party computation [MGW87, BOGW88], fully homomorphic encryption [Gen09, CGGI20, MAAM20], and cryptographic obfuscation [BGI$^+$01, GGH$^+$16].

[2]Unlike prior work, we treat the $\mathcal{DB}$ as a vector of records rather than a set. This is because scientific data is usually stored in a sorted manner. The position in the database provides information about the underlying dimensions. Making the database a vector allows us to reason about leakage on the underlying position.

1. The values x_i are the *dimensions* of a tuple,
2. The value m_i is the *domain* of a dimension, and
3. $\mathcal{R}$ is associated data and is called a *record*.

A range query $\mathbf{q} := \prod_{i=1}^{\ell}[a_i, b_i]$ finds all $\mathcal{DB}_{\mathbf{q}} := \{\mathcal{DB}_j | \forall 1 \leq i \leq \ell, a_i \leq x_{j,i} \leq b_i\}$. Searchable encryption design usually focuses on building an index (and corresponding protocol) that calculates the subset of records that have to be retrieved. That is, they design an index retrieval mechanism called RetrieveIndexes that returns a set $\mathcal{I} \subseteq [1, n]$ of matching records. A second, often unspecified mechanism is used to retrieve the actual records. We call this method RetrieveData. In an full system, these protocols are used in sequence for each query.

Recent work [GPPW23,GPP23] shows the efficiency and security aspects of record retrieval are crucial to understanding the overall system. Given the unequal attention paid to the two stages, this work focuses exclusively on RetrieveData assuming a correct RetrieveIndexes stage. Like search, initialization consists of two components, SetupIndexes, and SetupData.

1.1 Prior Work on Record Retrieval

We provide a brief overview of methods used to provide security during record retrieval. We then discuss the prior combination of these techniques. Techniques for hiding record access are:

Shuffling changes the associated position of each record, at initialization and/or retrieval.

Caching stores retrieved value at the client for a period of time, before returning them to the server.

Partitioning groups together data tuples into *parts*. All of a part is retrieved when an included tuple is needed. Empty data tuples may be included in a part as appropriate.

Query flattening [GKL$^+$20, MVA$^+$23] flattens the query distribution to be indistinguishable from the uniform distribution. Roughly, these systems (either by input or by learning) know the query distribution and issue the "inverse" of the distribution as fake queries.

We consider the following research question:

How are leakage and efficiency impacted by the organization of tuples into parts?

We make five simplifying assumptions:

1. During retrieve index, the system retrieves the correct records; this excludes systems that use approximate range covers [DPP$^+$16, FMET23]
2. During retrieve data, no "fake queries" are issued. This excludes systems that flatten the distribution [GKL$^+$20, MVA$^+$23].
3. That each tuple appears in a single part.
4. Empty tuples are only used when the dataset size is not divisible by part size.
5. That the partition is static; only parts are moved.

Assumption 1 is made for scoping reasons, however, considering approximately correct systems such as those that use range covers is an important piece of future work discussed in Section 8. Assuming a query distribution without flattening (Assumption 2) is necessary to study the research question. Assumptions 3-5 are satisfied by the cryptographic retrieval methods discussed below. We introduce representative record retrieval mechanisms. These mechanisms are used to assess the leakage of a partition strategy. These approaches are:

Shuffled Records positions are shuffled during SetupData. Records are kept in the same position throughout queries. In this approach, RetrieveData reveals to the server identifiers of returned records, known as access pattern [FVY$^+$17, BKM19, KKM$^+$22].

Oblivious methods One uses oblivious RAM (ORAM) [Gol87,GO96] or private information retrieval [KO97, CG97,CKGS98,BIM00,CHR17,BIPW17,LMW23] with encryption. With these techniques, one can retrieve a set of records, leaking only how many records are accessed. Throughout this work, we collectively refer to oblivious methods as ORAM, noting that PIR is appropriate for read-only data.

ν-bucketed oblivious For a query requiring retrieval of k parts, one retrieves $\nu^{\lceil \log_\nu k \rceil}$ items using an oblivious method [DPPS20].

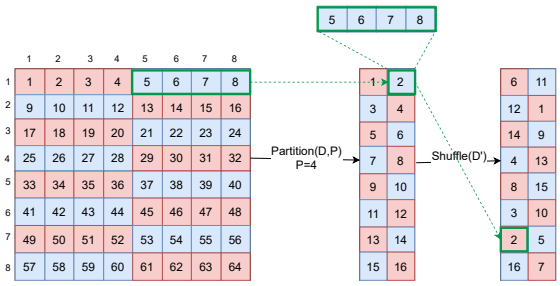


Figure 1: DRW-shuffle- a two-dimensional example of a $[8] \times [8]$ dataset. The numbers inside cells show the logical address location of the tuples in the storage. For example, tuples (5,1),(6,1),(7,1),(8,1), with addresses 5,6,7 and 8 in the original dataset, go to location 13 in the shuffled dataset.

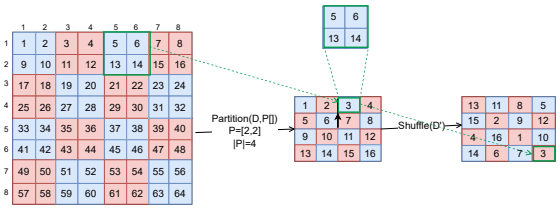


Figure 2: SLW-shuffle- a two-dimensional example of a $[8] \times [8]$ dataset. The numbers inside cells show the logical address location of the tuples on the storage. For example, tuples (5,1),(6,1),(5,2),(6,2), with addresses 5,6,13, and 14 in the original dataset, go to location 16 in the shuffled dataset on the right side of the figure.

SWiSSSE [GPPW23] Positions are shuffled during SetupData. When a part is retrieved, it is placed in a *cache* and evicted. Parts are evicted from the cache to available positions. Oblivious methods that leak only volume. SWiSSSE's eviction strategy allows the server to learn information about when the item was cached. The leakage function of this approach is complex. SWiSSSE calls this leakage function a "partial access pattern" (see Section 3).

We consider three partitions:

Divided-Row Shuffling or DRW-shuffle. This corresponds to row-major ordering in the data storage literature. In this approach, tuples are sorted by dimension 1, then dimension 2, ..., and finally by dimension ℓ. Records are then grouped into parts based on this order. Viewed as a hypercube, this takes tuples from the first row, second row, etc.. This is shown visually in Fig. 1 for parts of size 4. Depending on the size of a part, it may include some of a row or multiple rows. Looking ahead, we vary sizes across dimensions for all of our queries, so ordering by the first dimension is sufficient.

Slab-Wise Shuffling or SLW-shuffle. As in DRW-shuffle, tuples are first sorted. Instead of including a "line" of the hypercube in a part, one includes a small hypercube of the same width in each dimension. This is known as a *slab* in the data storage literature. This is shown visually in Fig. 2.

Record-Wise Shuffling or RCW-shuffle. In this setting, as illustrated in Fig. 3, the address of tuples is randomly permuted; parts then include contiguous regions based on the new addresses. This corresponds to a random creation of parts. One does not expect this method to be competitive with the prior two shuffling methods.

1.2 Our Contribution

We evaluate the cross product of the above data retrieval mechanisms and partitioning strategies. Our evaluation is with respect to efficiency and security on dense, synthetic two-, three-, and four-dimensional data of size up to $2M$ records. Our evaluation supports three major conclusions.

Security and Efficiency are Usually Aligned In Section 3, we argue for metrics to assess the security when one instantiates record retrieval with each candidate system (shown in Table 1).

Prior leakage attacks [KKM+22] require either a large variety in the number of parts returned or require many parts to be jointly returned by a query (to compute a co-occurrence matrix [CGPR15]).

Figure 3: RCW-shuffle: A two-dimensional example of a $[4] \times [4]$ dataset. The numbers inside cells show the logical address location of the tuples on the storage. We shuffle the locations of the tuples. For example, tuple (3,2) with address 7 in the original dataset goes to location 10 in the shuffled dataset on the right side of the figure. Parts are then created based on new organizations of equal size. (Any partition strategy has the same outcome after a random permutation.)

	Leakage	
Indexes Retrieval Mechanism	Profile	Metric
Shuffled	Access Pattern	Parts/Query
SWiSSSE	SWiSSSE	Parts/Query
ORAM	Volume	H(Parts)
bucketed ORAM	Bucketed Volume	H(Parts)

Table 1: Association between storage mechanism and assessed leakage metric. For all schemes, we consider the average # parts returned as the efficiency metric. For the ν-bucketed ORAM, the volume is padded to the next power of ν. H is the entropy function on the distribution of part numbers.

Having a smaller variance in the number of retrieved parts makes volumetric attacks more difficult. Driving down the number of parts returned makes access pattern attacks more difficult to execute. Thus, a well-organized system can benefit both efficiency and security simultaneously. The SWiSSSE and ν-bucketed ORAM systems are relatively new. There have not been any attacks against these systems, so our analysis of these systems is speculative. Grubbs et al. [GLMP18] argue partitioning makes volume attacks quantitatively more difficult but does not change them qualitatively.

For Shuffled and SWiSSSE, the page organization that requests the fewest number of pages also has the best security. However, efficiency and security are not aligned when using (ν-bucketed) ORAM. For both ORAM variants, one can actually reduce the entropy of reseponse size by having most queries request the maximum number of parts. RCW-shuffle, which requests nearly all parts, has the lowest values for both entropy metrics despite its poor efficiency. For our system on the largest dataset, we set $\nu = 2$, meaning the set of possible retrieved parts is the powers of 2. The RCW-shuffle system frequently requests the maximum number of parts, yielding very low entropy (below 1 across all tested query types).

The Impact of Query Geometry We consider four query types (Section 5):

1. Isotropic: each dimension has the same width.
2. Bisected anisotropic: separate dimensions into two parts with equal widths for each.
3. Gradual anisotropic: dimensions gradually reduce in width.
4. Outlier anisotropic: all but one dimension are of equal width. In these queries, we always make the *first* dimension differ. We split the query type by whether the first dimension is smaller or larger than the other dimensions, called *min* and *max*, respectively.

Our experiments demonstrate that SLW partitioning generally enhances efficiency and reduces data leakage across various query shapes, with one exception. DRW partitioning exhibit higher performance and lower leakage on outlier queries with an arbitrary sizes where dimensions $2, ..., \ell$ have large sizes (and the size of the first dimension is arbitrary).

The Partition Matters Both DRW-shuffle and SLW-shuffle outperform RCW-shuffle as expected and as shown in Table 2. Table 2 shows the percentage of tuples retrieved from the parts that were in the specified query. RCW-shuffle usually requires at least three times as many parts as the other two methods. Our detailed results, presented in Tables 4 and 5, show that in general **SLW-shuffle presents superior security and efficiency compared to DRW-shuffle**. There are two notable exceptions:

1. DRW-shuffle is sensitive to the width of dimensions $i > 1$, performing well when the query is large in other dimensions. (When the width of each dimension is the same in all dimensions, which

	Relevant Tuple %			
Dimension	Query Type	RCW	DRW	SLW
4	Isotropic	15.6	43.6	41.2
	Bisected	12.3	35.7	**40.6**
	Gradual	4.0	16.1	**23.9**
	Outlier Min	5.8	**76.5**	20.2
	Outlier Max	.1	.4	**2.8**
3	Isotropic	26.3	66.5	**73.3**
	Bisected	2.2	6.3	**8.1**
	Gradual	1.2	4.4	**6.7**
	Outlier Min	.5	**11.5**	3.1
	Outlier Max	.0	.2	**1.0**
2	Isotropic	34.0	66.3	**85.1**
	Bisected	25.1	49.7	**80.4**
	Gradual	24.5	49.2	**80.0**
	Outlier Min	.5	**64.4**	8.0
	Outlier Max	.5	.5	**8.1**

Table 2: Relevant tuple percentage $> 1M$ record datasets across number of dimensions. Size of desired record set over size of returned record set. Summarizes Tables 4 and 5. Bolded entries are at least 10% better than other methods for the same data and query set.

dimension is indexed is irrelevant.) This is displayed in the relevant query % in outlier queries in Table 2. In Outlier Min, where dimension 1 is small compared to other dimensions, DRW-shuffle performs much better than SLW-shuffle. However, when dimension 1 is larger than other dimensions SLW-shuffle performs better than DRW-shuffle though all methods perform poorly.

2. DRW-shuffle demonstrates more of a security benefit from the use of ν-bucketed volume than SLW-shuffle. This is due to more variety in part numbers for DRW-shuffle.

The above schemes are implemented, code is published in a public GitHub repository.

Organization The body of this work is dedicated to supporting the above three conclusions. Section 2 covers mathematical preliminaries, Section 3 describes prior work in data retrieval methods and argues for the leakage metrics in Table 1, Section 4 describes the part organization techniques, Section 5 the query types, Section 6 describes our evaluation methodology, Section 7 gives results, and Section 8 concludes.

2 Preliminaries

Let $\lambda \in \mathbb{N}$ be a security parameter. Throughout all algorithms are collections of algorithms indexed by security parameter λ. However, λ is often omitted from notation for simplicity.

For integers a, b let $[a, b] = \{x \in \mathbb{Z} | a \leq x \leq b\}$ and let $[b]$ be shorthand for $[1, b]$. We consider a database $\mathcal{DB}$ where each tuple $\mathcal{DB}_i$ consists of ℓ dimensions and an associated record. For $1 \leq i \leq \ell$, let m_i denote the maximum value of the ith dimension which ranges from $[m_i]$. A database $\mathcal{DB} = (\mathcal{DB}_1, ..., \mathcal{DB}_n)$ is an *ordered* collection of tuples where each

$$\mathcal{DB}_j \in \left(\prod_{i=1}^{\ell} [m_i] \right) \times \mathcal{R}.$$

where $1 \leq x_i \leq m_i$. For a query $\vec{q} = (a_1, b_1), ..., (a_\ell, b_\ell)$ where $1 \leq a_i \leq b_i \leq m_i$ let

$$\mathcal{DB}_{\vec{q}} = \{\mathcal{DB}_j | \forall 1 \leq i \leq \ell, a_i \leq x_{j,i} \leq b_i\}.$$

Separating lookup and retrieval Let $\mathcal{DB}$ be a database of size N. We separately define two functions:

1. $\texttt{Index} : [m_1] \times ... \times [m_\ell] \to 2^{\{0,1\}^N}$.

2. $\texttt{Storage} : 2^{\{0,1\}^N} \to (\mathcal{R} \cup \perp)^N$.

Definition 1 (Multi-Dimensional Range-Query Searchable Encryption)**.** *A Searchable Encryption scheme is a set of algorithms $\Sigma =$ (Gen, SetupIndexes, SetupData, Trapdoor, RetrieveIndexes, DecryptIndexes, RetrieveData, DecryptRecords) such that:*

- $sk \leftarrow Gen(1^\lambda)$*, generates secret state* sk*.*
- $\mathcal{I} \leftarrow SetupIndexes(sk, \mathcal{DB})$*, inputs* sk *and* $\mathcal{DB}$*, generating an encrypted index set.*
- $\mathcal{D} \leftarrow SetupData(sk, \mathcal{DB})$*, inputs* sk *and* $\mathcal{DB}$*, generating an encrypted array data* $\mathcal{D}$*.*
- $t_{ind} \leftarrow Trapdoor(sk, \vec{q})$*, inputs* $\vec{q}$*, generating a token* t_{ind}*.*
- $\mathbf{eid} \leftarrow RetrieveIndexes(\mathcal{I}, t_{ind})$*, on an encrypted index* $\mathcal{I}$ *and token* t_{ind}*, outputs* $\mathbf{eid}$*.*
- $t_{data} \leftarrow DecryptIndexes(\mathbf{eid}, sk)$ *takes the encrypted pointers, creating a token to retrieve the relevant data records.*
- $Y \leftarrow RetrieveData(\mathcal{D}, t_{data})$*, on* $\mathcal{D}$ *and the trapdoor* t_{data}*, outputs a set* Y *of encrypted records.*
- $X \leftarrow DecryptRecords(sk, Y)$*, decrypts the final result.*

Notes: All algorithms are defined non-interactively and defined with a fixed $\mathcal{I}, \mathcal{D}$ at the server. Notation naturally extends if either of these assumptions does not hold.

Correctness The search scheme is *correct* if for some negligible function $\texttt{ngl}(\lambda)$:

$$\Pr\left[X = \mathcal{DB}_{\vec{q}} \;\middle|\; \begin{array}{r} \mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda) \\ \mathcal{I} \leftarrow \mathsf{SetupIndexes}(\mathsf{sk}, \mathcal{DB}) \\ \mathcal{D} \leftarrow \mathsf{SetupData}(\mathsf{sk}, \mathcal{DB}) \\ t_{ind} \leftarrow \mathsf{Trapdoor}(\mathsf{sk}, \vec{q}) \\ \mathbf{eid} \leftarrow \mathsf{RetrieveIndexes}(\mathcal{I}, t_{ind}) \\ t_{data} \leftarrow \mathsf{DecryptIndexes}(\mathbf{eid}, \mathsf{sk}) \\ Y \leftarrow \mathsf{RetrieveData}(\mathcal{D}, t_{data}) \\ X \leftarrow \mathsf{Decrypt}(\mathsf{sk}, Y) \end{array}\right] \geq 1 - \texttt{ngl}(\lambda).$$

Security We consider a real ideal adaptive notion of security where the simulator is given leakage function $\mathcal{L} = (\mathcal{L}^{\texttt{setup}}, \mathcal{L}^{\texttt{RetrieveInd}}, \mathcal{L}^{\texttt{RetrieveData}})$. To achieve this, we define two games: the real game represents the execution of the actual SSE protocol, while the ideal game is a simulation of the real game. In the ideal game, the goal is to mimic the behavior of the real game using only the formulated leakage information.

Definition 2 (System-Wide Security of Static SSE)**.** *Let Σ be an SSE scheme and consider the following probabilistic experiment where $\mathcal{A}$ is a stateful adversary, $\mathcal{S}$ is a stateful simulator, and $\mathcal{L}$ is the leakage function.*

For a leakage function $\mathcal{L}$, we say that SSE is adaptively semantically secure if, for all polynomial n_q and all polynomial-size adversary $\mathcal{A}$, there exists a non-uniform polynomial-size simulator $\mathcal{S}$ such that for all polynomial-size distinguishers D,

$$|Pr[D(Real_{SSE,\mathcal{A}}(\lambda, n_q)) = 1] - Pr[D(Sim_{SSE,\mathcal{A},\mathcal{S}}(\lambda, n_q)) = 1]| \leq \texttt{ngl}(\lambda)$$

where $Real_{SSE,\mathcal{A}}(\lambda)$ and $Sim^{\mathcal{L}}_{SSE,\mathcal{A},\mathcal{S}}(\lambda)$ are defined in Fig. 4.

3 Prior Work: Retrieval Mechanisms and Leakage Metrics

This section provides an overview of retrieval methods discussed in the Introduction (shuffled, ORAM, ν-bucketed ORAM, SWiSSSE). This cross-section of retrieval methods covers the main techniques of caching, shuffling, and partitioning. (As a reminder, we ignore query flattening, as it destroys any optimization of partitioning based on query load.) Throughout our discussion, we consider a persistent adversary [GRS17] as codified in Definition 2.

$\underline{Real_{SSE,\mathcal{A}}(n_q, \lambda)}$

1. $sk \leftarrow \mathsf{Gen}(1^\lambda)$,
2. $\mathcal{DB} \leftarrow \mathcal{A}(1^\lambda)$,
3. $\mathcal{I} \leftarrow \mathsf{SetupIndexes}(\mathsf{sk}, \mathcal{DB})$,
4. $\mathcal{D} \leftarrow \mathsf{SetupData}(\mathsf{sk}, \mathcal{DB})$,
5. For $1 \leq i \leq n_q$:
   (a) $(q_i, st_\mathcal{A}) \leftarrow \mathcal{A}(\mathcal{I}, \mathcal{D}, \mathtt{view}_1, ..., \mathtt{view}_{i-1})$
   (b) $t_i \leftarrow \mathsf{Trapdoor}(sk, q_i)$,
   (c) $\mathbf{eid}_i \leftarrow \mathsf{RetrieveIndexes}(\mathcal{I}, t_i)$,
   (d) $t_{data,i} \leftarrow \mathsf{DecryptIndexes}(\mathbf{eid}_i, \mathsf{sk})$,
   (e) $Y_i \leftarrow \mathsf{RetrieveData}(\mathcal{D}, t_{data,i})$,
   (f) $\mathtt{view}_i = (t_i, \mathbf{eid}_i, t_{data,i}, Y_i)$.
6. Output $\mathcal{A}(\mathtt{view}_{n_q})$.

$\underline{Sim_{SSE,\mathcal{A},\mathcal{S}}(n_q, \lambda)}$

1. $\mathcal{DB} \leftarrow \mathcal{A}(1^\lambda)$
2. $(\mathcal{I}, \mathcal{D}) \leftarrow \mathcal{S}(\mathcal{L}^{\mathtt{setup}}(\mathcal{DB}))$
3. For $1 \leq i \leq n_q$:
   (a) $q_i \leftarrow \mathcal{A}(\mathcal{I}, \mathcal{D}, \mathtt{view}_1, ..., \mathtt{view}_{i-1})$
   (b) $\mathtt{view}_i \leftarrow \mathcal{S}(\mathcal{L}^{\mathtt{RetrieveInd}}(\mathcal{DB}, q_i), \mathcal{L}^{\mathtt{RetrieveData}}(\mathcal{DB}, q_i))$
4. Output $\mathcal{A}(\mathtt{view}_{n_q})$.

Figure 4: SSE Real-Ideal security game

This section is organized as follows: Section 3.1 introduces the defense schemes and describes the relevant leakage functions, and Section 3.2 argues for simple quantitative metrics on the quality of a partition with respect to the leakage functions (see [KKM+22] for an overview of leakage profiles). Before introducing defensive schemes, we introduce common terminology used to describe leakage functions.

- **Access Pattern.** Reveals the (persistent) identifiers of all returned parts.
- **Search/Equality Pattern.** Reveals if two queries are equal.
- **Co-occurrence Pattern.** Reveals parts that are jointly returned by queries.
- **Volume Pattern.** Reveals the number of parts returned by each query.

3.1 Prior Retrieval Mechanisms and Leakage Analysis

We consider four record retrieval methods. Each of these mechanisms yields a different leakage profile. In Section 3.2, we introduce 1-dimensional metrics to assess leakage.

Shuffled During SetupData one selects a random permutation of parts. For a query, when one receives the set of $\mathcal{I}$ matching indices from RetrieveIndexes, one recomputes the permutation and asks for the permuted positions. This method has access pattern leakage but does not leak any information on the original location in the $\mathcal{DB}$.

ORAM ORAM [Gol87, GO96] compiles storage and memory access so that accessed positions are independent of the requests. In ORAM schemes, write-backs and shuffling happen with every request. A system that uses ORAM for RetrieveData leaks how many records are accessed (assuming the tuple and parts are fixed size).

ν-bucketed Oblivious SEAL [DPPS20] allows fine-tuning of leakage during the setup phase. The construction of SEAL involves adjustable ORAM (α) and adjustable padding (ν). They use two main techniques. The first is partition memory to 2^α slots where the accessed position inside of each slot is hidden. This is the opposite of the partitions we consider in this work, which hides the position within a part (see Section 4). However, the high-level approach of hiding a portion of the accessed position is the same. The second technique used is ν-adjustable padding for the number of accesses from an ORAM. For a constant ν whenever one wants to retrieve k records one instead retrieves $\nu^{\lceil \log_\nu k \rceil}$ records. This reduces the volume pattern leakage because there are $\lceil \log_\nu N \rceil$ distinct sizes.

SWiSSSE [GPPW23] System-Wide Security for Searchable Symmetric Encryption or SWiSSSE presents new SSE constructions designed to minimize the system-wide access pattern and volume pattern leakages while maintaining reasonable performance. The core technique in designing SWiSSSE consists of a two-fold approach: partitioning to address the volume pattern leakage problem and implementing a client stash with pseudo-random writebacks to tackle the access pattern leakage problem.

In this approach, the server has two different key-value pairs stored as encrypted data structures. First is the mapping of each keyword to associated encrypted indexes set in the SetupIndexes phase, and second is the map of encrypted documents to those addresses in the memory set in the SetupData.

In the SetupData, SWiSSSE orders the keywords and divides them into equal-sized partitions of size B. Parts are padded with fake data to equalize the frequency of all keywords in each partition. As we consider dense range data, there is no need for padding (except for the "last" parts in dimensions). A larger B (fewer partitions) incurs lower leakage but more padding and greater search overheads.

During search, upon each query w_i by the client, the system checks if w_i is present in the client's stash. If not, a two-round process involving RetrieveData and RetrieveIndexes is executed, after which the client stores the decrypted entries in their stash memory. The server evicts all retrieved entries from both key-value pairs. Every few queries, the client randomly chooses a proportion of their memory stash, re-encrypts it, and sends it back to the server. This data is stored in currently available positions (where the data has been evicted)

3.2 Prior Attacks and Leakage Metrics

We argue for metrics to assess a partitioning strategy across retrieval methods. This analysis considers both the leakage function and existing attacks. Leaker [KKM$^+$22] provides an overview of attacks that perform data and query-recovery (without the use of either auxiliary or known data and queries).

Lacharite et al. [LMP18] showed a range attack from *access pattern* leakage on dense 1D databases with uniformly distributed queries. Their attack fully reconstructs data with $N(\log N + 3)$ queries for $N \geq 26$ where N is the number of distinct values in the range. They also demonstrate $N \log N/2 - O(N)$, as a lower bound for the expected number of queries needed for full reconstruction from access pattern leakage. The core of access pattern attacks is the formation of a co-occurrence matrix. In attacks rows are chained together to gain more constraints about records stored in each position. One builds a graph on the known relationship between data elements (with the number of queries they co-occur in being the weight). The reconstruction space [KMPP22] shrinks as one observes more of this graph. **We use the average number of parts returned as our metric for this setting.** A low average volume ensures that fewer edges will be drawn in the co-occurrence graph with each query

Gui et al. [GPPW23] consider the impact of the *SWiSSSE* system on: 1) Access Pattern leakage, 2) Co-occurrence leakage, 3) Volume leakage, and 4) Search Pattern Leakage, which occurs sometimes based on the state of the scheme.

Due to the use of client stash and delayed writebacks, only a single row of a co-occurrence matrix is exposed to the adversary. As access items are shuffled, one can partially associate the new address with the old address. The hardness of creating this association depends on the query load resulting in a complex, partial, and probabilistic access pattern. To test this, Gui et al. implemented an attack where an attacker learns a "highly refined co-occurrence leakage," which contains leakages from both the index retrieval and document retrieval stages. This attack was successful at recovering high-frequency keywords more effectively than low-frequency codewords. The attack requires a large fraction of auxiliary information about the dataset.

SWiSSSE adjusts part sizes and uses larger part for the most frequent keywords, and smaller parts for the rest of the keywords. The size and subsequently the number of parts for each query is a parameter that affects leakage. To this end, **we compute the average number of accessed parts per query to quantify co-occurrence**. The same metric is relevant for the shuffling mechanism.

Grubbs et al. [GLMP18] showed that with only having *volume* leakage of range queries, under the condition that we see every value at least once, one can reconstruct a database. For one dimensional values where every possible value occurs in the dataset, reconstruction is possible if the number of records $R \geq N^2/2$. N is the number of possible values. Grubbs et al. [GLMP18] attack works as follows. Having the set of all possible volumes, one can see them as an elementary range $[1, N_i]$ for the value N_i. Then the attacker draws a graph with all N_i's as the node and draws an edge between each two nodes that their difference is also a value in the set. Starting with the minimum and maximum (R) value of this set as necessary nodes, the attacker tries to find the clique of the graph and eventually finds the right values of each record in the database.

For defense mechanisms that leak volume patterns, such as ORAM, the above attack relies heavily on observing all ranges. Grubbs et al.'s [GLMP18] attack works for any query distribution; however, when the distribution is uniform, $N^2 \log(N)$ queries are necessary to observe all possible queries. When the query distribution is skewed and looks more like a real-world distribution, for example, for Zipf distribution, the number of queries to observe all ranges is $N^2 \log^2(N)$.

When one moves from one-dimensional to multi-dimensional ranges the number of ranges increases from N^2 to $N^{2\ell}$ making it more difficult for the adversary to observe each range. **Thus, to quantify leakage for volume attacks, we consider the entropy on the number of parts accessed.** This is a one-dimensional measurement of the closeness to the uniform distribution. We use this metric both for traditional volumetric leakage and ν-bucketed ORAM as described by SEAL [DPPS20].

SEAL [DPPS20] observes that the bigger the value of ν is, the smaller the chance of seeing a successful attack is. However, larger ν would cause a large overhead in search computations. The discussion about the effect of the record organization on the introduced leakage metrics can be found in Section 7.

4 Partition Organization

We implement a multi-dimensional search to measure different partitions using a three-phase multimap as follows:

1. **Creation of an Index Structure:** In this initial phase, an index structure is established by running SetupData, and SetupIndexes. We use a k-d tree [Ben75]for finding the appropriate records, all exact methods are equivalent for evaluating the record retrieval stage.
2. **Data Shuffling:** The second phase entails shuffling the source data on disk.
3. **Dictionary Generation:** In the third phase, a dictionary is created where a range from the index structure is linked with a set of values derived from the newly shuffled data.

We consider three distinct shuffling schemes.

Divided-Row Shuffling (DRW-shuffle) In Divided-Row Shuffling we adopt a row-based storage order for the dataset. It is shown in Figure 1. DRW-shuffle sequentially divides the dataset into partitions of size $|P|$ through the stored source data. For example, consider a dataset with dimensions $[8] \times [8]$ and a partition size of $|P| = 4$. In this scenario, we obtain 16 partitions that transform the dataset into a $[8] \times [2]$ matrix. Subsequently, the shuffling is performed by applying a pseudorandom permutation to the parts.

The shuffled matrix forms the foundation for the subsequent steps. From this point, we can construct a multi-dimensional indexing structure by utilizing the starting points of each partition. In the illustrative example provided in Figure 1, the starting points for the partitions are indicated as $(1, 1)$, $(5, 1)$, $(1, 2)$, $(5, 2)$, and so forth, extending up to $(1, 8)$ and $(5, 8)$.

Slab-Wise Shuffling (SLW-shuffle) In Slab-Wise Shuffling (SLW-shuffle) we adopt a partitioning strategy for datasets that are Slab-oriented, denoted as P, and are characterized by a d-dimensional partition shape of $[p_1] \times [p_2] \times \ldots \times [p_d]$. This partitioning is depicted in Figure 2. In the specific illustration presented, a dataset initially sized $[8] \times [8]$ is segmented into a collection of 16 distinct $[4] \times [4]$ matrices. This results in a transformation to a $[4] \times [4]$ dataset structure. Following this Slab-oriented organization, we begin a shuffling process targeting this newly arranged dataset. The shuffling is guided by the locations of the individual slabs within the structure.

Record-Wise Shuffling (RCW-shuffle) RCW-shuffle is a record-wise permutation of the source data. Record-Wise Shuffling unlike the preceding methods, requires no logical grouping of values; hence the record-wise designation.

Fig. 3 illustrates a two-dimensional example. In this example, we have a $[4] \times [4]$ two-dimensional array of tuples. We store the tuples in row-based storage and assign logical location addresses to the tuples in a row-based order from 1 to 16. Then we permute the locations, mapping them so they are stored in a new spot. For example, a record located at (1,1) is originally located in location 1 and, after mapping, is relocated to 6, while record (3,2) is located in location 7, but is mapped to the new location 10. Afterward, we build the multi-dimensional index structure based on the points in their new locations.

Dataset Size	# queries	DRW	SLW
$16^4 = 2^{16}$	1000	256	$\lvert[4]^4\rvert = 256$
$32^4 = 2^{20}$	10000	4096	$\lvert[8]^4\rvert = 4096$
$64^3 = 2^{18}$	1000	512	$\lvert[8]^3\rvert = 512$
$128^3 = 2^{21}$	5000	4096	$\lvert[16]^3\rvert = 4096$
$256^2 = 2^{16}$	1000	256	$\lvert[16]^2\rvert = 256$
$1024^2 = 2^{20}$	5000	4096	$\lvert[64]^2\rvert = 4096$

Table 3: Dataset information and partition settings.

5 Query Distribution

We consider four types of queries based on the relative queried width of each dimension. Isotropic queries, denoted as $\mathcal{Q}_{\text{iso}}$, have a uniform scaled length in each dimension. For a query $\mathbf{q} = [a_1, b_1] \times \cdots \times [a_d, b_d]$ let $\forall i, w_i := |a_i - b_i|$. A query $\vec{q}$ belongs to the set $\mathcal{Q}_{\text{iso}}$ if

$$\mathcal{Q}_{\text{iso}} = \{\mathbf{q} | \forall i, j \, |w_i - w_j| \leq \epsilon\}$$

where ϵ is a deviation parameter. In our implementation, we set $\epsilon = 0$, as we only consider queries that are perfectly isotropic. Anisotropics have scaled lengths of their intervals that vary across dimensions. The set of anisotropic queries is the complement of isotropic queries: $\mathcal{Q}_{\text{aniso}} = \mathcal{Q}_{\text{iso}}^{\complement}$. We categorize anisotropic queries into *Bisected, Gradual,* and *Outlier*.

- Bisected anisotropic queries (BAQ) partitions the dimensions into two parts with a width gap of at most ϵ within each part. In four dimensional datasets, this means two sets of two dimensions have the same width. In three dimensional data, two dimensions have the same width. In two dimensional data, this represents a general anisotropic query.
- Gradual anisotropic (GAQ) queries scale the width of each dimension down by a factor of c. In our implementation, we randomly choose $c \in \{1, 2, \ldots, \frac{\mathcal{DB}.max}{n} \times \frac{1}{w}\}$, where w is the width of the query's widest dimension.
- Outlier anisotropic queries (OAQ) are isotropic in all but one dimension which has either a smaller or larger width. We call a `OAQ` query min if the dimension of differing size is smaller and max if it is larger. In our implementation, the outlier is always dimension 1 to highlight the differences between our shuffling techniques.

6 Evaluation Methodology

We experimentally evaluate the performance of our schemes. We used a randomly generated dataset because the actual values in the datasets are unlikely to impact our measurements. Our conclusions depend on the dataset's size, number of dimensions, and the width of each dimension. We consider six datasets. We focus on four-dimensional datasets of sizes $1048576 = 32^4$ and $65536 = 16^4$. We also consider two sets of two and three-dimensional datasets respectively comprising $65536 = 256^2$, $1048576 = 1024^2$, $262144 = 64^3$, and $2097152 = 128^3$. These datasets are summarized in Table 3. To the best of our knowledge, no standard benchmarks of queries on array-based data are available. **A uniform distribution was used for our experiments to generate Hyper-Rectangular range query samples for each query shape according to the shapes discussed in Section 5.** For dense data, only the relative size of each dimension matters; we believe our query shapes explore this parameter space well.

Experimental Setup: We implemented our schemes in Python 3.10.12 and conducted all our experiments on a computer with 8-core processors and 16G RAM. We utilized Python's cryptography library version 40.0.2 [pyc23] and employed AES-128 encryption in XTS mode with a 256-bit key size for symmetric encryption, using line positions as tweaks. Our code is published in a public GitHub repository.

6.1 Implementation Details

We present a hierarchical multimap data structure based on k-d trees, tailored for managing our dense dataset $\mathcal{DB}$ of encrypted elements within a multi-dimensional index $\mathcal{I}$, while concurrently supporting efficient range queries.

The multi-dimensional index $\mathcal{I}$ is a multi-dimensional index tree (k-d tree), where the leaf nodes function are fixed-size parts $B = \{b_1, b_2, \ldots, b_m\}$, each capable of accommodating B records. Given the dense nature of $\mathcal{DB}$, each partition b_i is fully occupied. These partitions link to a value dictionary $\mathcal{D} : B \to P(\mathcal{DB})$ that maps each part b_i to a corresponding set v_i of encrypted values. We review some details below:

1. The design of $\mathcal{I}$ must be optimized to facilitate swift multi-dimensional range queries, given that all its leaf nodes are fully populated.
2. The lack of any "partition management" due to the dense dataset simplifies the architecture but necessitates an upfront computational cost for initializing $\mathcal{I}$ and V.
3. We use AES-128 for both encryption and shuffling, and assume it will not be the target of any attack.

7 Experiment Results

Our primary efficiency measure is the number of parts accessed per query. For best security, one seeks a small number of returned parts with small variance. This saves both on memory access for the return while presenting less leakage. We computed the leakage metrics introduced in Section 3, shown in Table 4 and 5 .

Experiment Results for $[32]^4$ Dataset We begin by discussing results on the dataset of size $[32]^4$. Figure 5(a) illustrates the distribution of part access numbers for the $[32]^4$ dataset with respect to the four different query shapes: isotropic queries, bisected anisotropic queries, gradual anisotropic queries, and outlier anisotropic queries using violin graphs. The width of a violin graph represents frequency while the y-axis represents the number of parts accessed in a query. In all of our results, we present the size of the query for comparison (Figure 5(b)). A solution with 100% relevant tuple percentage would have these two identical graphs.

Table 4 shows the (ν-bucketed) entropy and average number of parts accessed across all queries. For ν-parts, we have rounded the number of parts to the closest power of ν, which at least the number of accessed parts, and then computed our leakage metrics over them. As discussed in Section 3, Access Pattern and SWiSSSE leakage are minimized by minimizing the number of partitions that are accessed through the retrieval process. To minimize (bucketed) volume leakage, one desires low entropy in the volume of accessed parts.

The use of ν-parts is most useful when there is a small increase in the amount of required parts but a large decrease in the entropy of parts returned. As an example, the results in Table 4 shows that for DRW-shuffle when we have outlier anisotropic queries max, the average number of parts increases by 10% while entropy of parts decreases by roughly 66%. On the other hand, for SLW-shuffle for isotropic queries, the average number of parts increases by 22% with the entropy decreasing by only 17%.

As shown in Table 4, across query shapes RCW-shuffle has lowest value of ν-bucketed entropy, followed by DRW-shuffle, and then SLW-shuffle. Of course, one can minimize entropy by always retrieving the maximum number of parts, but it doesn't mean the partition is effective. Table 4 shows that the average number of partitions that have been accessed for RCW-shuffle is close to the overall number of parts in all cases (256 in the 32^4 dataset). RCW-shuffle is the least desirable shuffling scheme from the performance perspective. However, it displays low volumetric leakage because it frequently requests all parts.

It can also be inferred from the table that the benefit of ν-bucketing for the DRW-shuffle is more significant than for the SLW-shuffle as the entropy decreases more in DRW-shuffle. Looking at Figure 5(a), this can be the result of having a more continuous distribution in DRW-shuffle than in the SLW-shuffle. For outlier anisotropic queries we see the most discontinuity in SLW-shuffle distribution (5(a)-(IV) and (V)), and this is when bucketing does not have any effect on the entropy of accessed parts.

DRW-shuffle and SLW-shuffle have very different performance and security on outlier queries, they differ by a factor of 2 on both average and entropy of parts. For isotropic queries, where the query has the same width along all dimensions on 4D data, DRW-shuffle performs better than SLW-shuffle. On the security side, DRW-shuffle has a lower average number of parts but higher entropy.

Size	Query Type	Shuffling Technique	Parts Avg	Parts STD	Parts H	ν-Parts Avg	ν-Parts STD	ν-Parts H
$[16]^4$	**Isotropic**	RCW-shuffle	200	96	3	210	93	1.2
	Avg = 9700	DRW-shuffle	73	66	3.9	100	93	2.7
	STD = 14000	SLW-shuffle	79	91	3.5	96	100	2.9
	Bisected	RCW-shuffle	220	67	3.6	230	59	0.8
	Avg = 8400	DRW-shuffle	78	65	6.3	106	90	2.6
	STD = 12000	SLW-shuffle	72	73	4.1	96	96	2.7
	Gradual	RCW-shuffle	230	58	2.3	240	54	.34
	Avg = 3300	DRW-shuffle	60	49	4	76	63	2.4
	STD = 2500	SLW-shuffle	41	29	3.7	58	42	2.4
	Outlier Min	RCW-shuffle	260	.11	.08	260	0	0
	Avg = 5400	DRW-shuffle	28	12	3.1	38	20	1.6
	STD = 2400	SLW-shuffle	79	27	0.8	79	27	0.8
	Outlier Max	RCW-shuffle	106	77	5.3	140	100	1.6
	Avg = 180	DRW-shuffle	29	12	3.2	39	20	1.6
	STD = 160	SLW-shuffle	7.7	5.8	1.5	7.7	5.8	1.5
$[32]^4$	**Isotropic**	RCW-shuffle	230	71	1.7	230	66	.65
	Avg = 150000	DRW-shuffle	82	68	5.5	110	95	2.8
	STD = 207000	SLW-shuffle	87	92	3.5	105	100	2.9
	Bisected	RCW-shuffle	250	38	1.5	250	32	.28
	Avg = 120000	DRW-shuffle	85	69	6.7	110	91	2.6
	STD = 180000	SLW-shuffle	74	74	4.2	99	95	2.7
	Gradual	RCW-shuffle	250	33	.76	250	30	.13
	Avg = 40900	DRW-shuffle	62	48	5	83	65	2.4
	STD = 36000	SLW-shuffle	42	30	3.6	60	43	2.5
	Outlier Min	RCW-shuffle	260	0	0	260	0	0
	Avg = 61000	DRW-shuffle	19	8.8	2.6	21.8	10	1.5
	STD = 28000	SLW-shuffle	74	23	.6	74	23	.6
	Outlier Max	RCW-shuffle	171	91	4.9	200	97	.8
	Avg = 730	DRW-shuffle	40	14	2.9	43	15	.94
	STD = 710	SLW-shuffle	6.3	4.3	1.3	6.3	4.3	1.3

Table 4: Leakage Metrics for 4D datasets. ν set to 2, other small values displayed similar trends. H is the entropy of the (bucketed) part distribution.

However, this improvement in average returns and entropy found from the different shuffling schemes is not consistent with the other query shapes or other datasets; For all other query types, the average number of parts and entropy for SLW-shuffle is smaller compared to DRW-shuffle.

Experiment Results for $[16]^4$ Dataset Figure 6(a) illustrates the distribution of part access numbers for the $[16]^4$ dataset with respect to the four different query shapes: isotropic queries, bisected anisotropic queries, gradual anisotropic queries, and outlier anisotropic queries. The distribution of the sample query sizes for this experiment is shown in Fig. 6(b). This dataset supports similar conclusions to the $[32]^4$ dataset.

Experiments Results for 2D and 3D Datasets Experiments on lower dimensions shows that for the isotropic queries, SLW-shuffle has been performance than DRW-shuffle. Furthermore, we see a decrease both in entropy and the average number of parts accessed in SLW-shuffle in comparison to DRW-shuffle. In the isotropic queries setting we have a uniform spread of ranges in our query, which matches the shape of our SLW-shuffle scheme. This explains why the violin graph for isotropic queries for SLW-shuffle displays the same shape regardless of the number of dimensions. However, the fraction of the width of each dimension covered by a single slab is higher in the datasets with 3 and 4 dimensions. This means for datasets of dimension 3 and 4 more rows fit into a single DRW-shuffle part. This makes a DRW-shuffle part be a "multi-dimensional" object and why DRW-shuffle has high performance on isotropic queries in 4 dimensions.

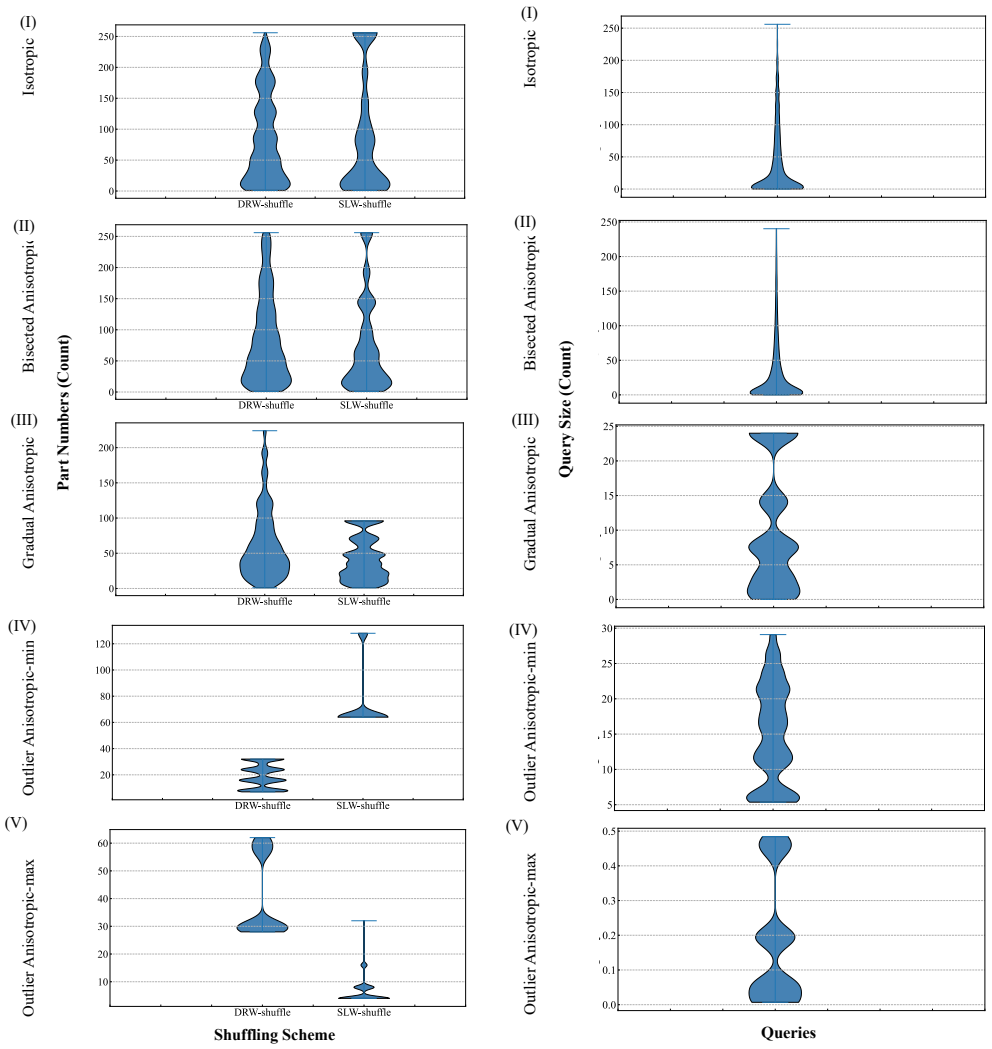
Table 5 shows how the leakage metrics change across different query shapes and shuffling techniques for a 2D and 3D datasets. The dataset details can be found in Table 3. Figure 7 and 8 illustrate the part access numbers distribution across all query shapes, as well as the distribution of the query sizes for 3D and 2D datasets respectively.

8 Conclusion

Prior study of secure multi-dimensional storage focuses on indexing structure. This work focuses on the less-charted territory of optimizing storage and retrieval steps. Our primary study is the differences in

Size	Query Type	Shuffling Technique	Parts Avg	Parts STD	Parts H	ν-Parts Avg	ν-Parts STD	ν-Parts H
$[256]^2$	**Isotropic**	RCW-shuffle	240	50	1.5	240	46	.44
	Avg = 22000	DRW-shuffle	130	74	7.8	170	91	2
	STD = 20000	SLW-shuffle	100	83	4.8	130	101	2.6
	Bisected	RCW-shuffle	250	19	1	250	14	.08
	Avg = 16000	DRW-shuffle	130	74	7.8	170	90	1.9
	STD = 14000	SLW-shuffle	79	60	6.3	110	85	2.4
	Gradual	RCW-shuffle	250	18	.74	250	14	0.07
	Avg = 16000	DRW-shuffle	130	69	7.8	170	83	1.8
	STD = 12000	SLW-shuffle	80	51	5.9	111	75	2.3
	Outlier Min	RCW-shuffle	208	34	5.6	260	0	0
	Avg = 500	DRW-shuffle	2	.8	1.6	2.3	1.2	1.6
	STD = 200	SLW-shuffle	17	3.5	.3	17	3.5	.3
	Outlier Max	RCW-shuffle	209	35	5.6	260	0	0
	Avg = 510	DRW-shuffle	254	.81	1.6	256	0	0
	STD = 210	SLW-shuffle	17	3.8	.33	17	3.8	.33
$[1024]^2$	**Isotropic**	RCW-shuffle	250	26	.45	250	24	.14
	Avg = 350000	DRW-shuffle	130	74	7.9	170	91	2
	STD = 310000	SLW-shuffle	100	83	4.8	130	100	2.6
	Bisected	RCW-shuffle	260	7.1	.12	260	4.5	.01
	Avg = 260000	DRW-shuffle	130	74	8	170	91	2
	STD = 230000	SLW-shuffle	80	61	6.4	110	85	2.5
	Gradual	RCW-shuffle	260	1.2	.02	260	0	0
	Avg = 260000	DRW-shuffle	130	62	7.7	170	74	1.6
	STD = 150000	SLW-shuffle	78	40	4.5	109	64	2.0
	Outlier Min	RCW-shuffle	260	1.6	.86	260	0	0
	Avg = 5600	DRW-shuffle	2.1	.82	1.7	2.5	1.2	1.6
	STD = 2900	SLW-shuffle	17	4	.36	17	4	.36
	Outlier Max	RCW-shuffle	260	1.6	.8	260	0	0
	Avg = 5600	DRW-shuffle	260	.73	1.5	256	0	0
	STD = 2900	SLW-shuffle	17	4.1	.37	17	4.1	.37
$[64]^3$	**Isotropic**	RCW-shuffle	460	130	2	470	120	.7
	Avg = 59000	DRW-shuffle	180	150	6.5	250	190	2.7
	STD = 67000	SLW-shuffle	170	170	4.3	203	190	3
	Bisected	RCW-shuffle	480	99	2	490	88	.4
	Avg = 44000	DRW-shuffle	170	140	7.5	230	180	2.6
	STD = 52000	SLW-shuffle	130	130	6	170	170	2.8
	Gradual	RCW-shuffle	500	70	1.2	504	54	.2
	Avg = 24000	DRW-shuffle	130	91	6.1	190	140	2.3
	STD = 17000	SLW-shuffle	86	53	4.2	123	83	2.3
	Outlier Min	RCW-shuffle	510	0.3	.4	510	0	0
	Avg = 9500	DRW-shuffle	20	9	2	22	10	1.5
	STD = 4200	SLW-shuffle	75	24	.7	75	24	.7
	Outlier Max	RCW-shuffle	260	140	6.3	340	190	1.5
	Avg = 460	DRW-shuffle	73	24	2.7	76	25	.7
	STD = 350	SLW-shuffle	11	6	1.1	11	6	1.1
$[128]^3$	**Isotropic**	RCW-shuffle	490	95	1.3	490	90	.4
	Avg = 500000	DRW-shuffle	210	160	7.4	270	200	2.5
	STD = 570000	SLW-shuffle	180	170	4.3	210	200	3
	Bisected	RCW-shuffle	500	58	.9	505	50	.2
	Avg = 360000	DRW-shuffle	200	150	8	260	180	2.5
	STD = 430000	SLW-shuffle	140	130	6	180	170	2.8
	Gradual	RCW-shuffle	510	31	.4	510	24	0
	Avg = 180000	DRW-shuffle	150	96	6.7	220	150	2.2
	STD = 130000	SLW-shuffle	86	54	4.3	120	83	2.3
	Outlier Min	RCW-shuffle	512	0	0	510	0	0
	Avg = 82000	DRW-shuffle	22	12	3.3	30	20	2.1
	STD = 44000	SLW-shuffle	80	28	0.8	80	28	0.8
	Outlier Max	RCW-shuffle	440	127	4	470	116	0.5
	Avg = 4700	DRW-shuffle	137	39	3.8	140	41	.5
	STD = 4000	SLW-shuffle	13	7.3	1.3	13	7.3	1.3

Table 5: Leakage Metrics for 2D and 3D datasets. ν set to 2, H is the entropy of the (bucketed) part distribution.



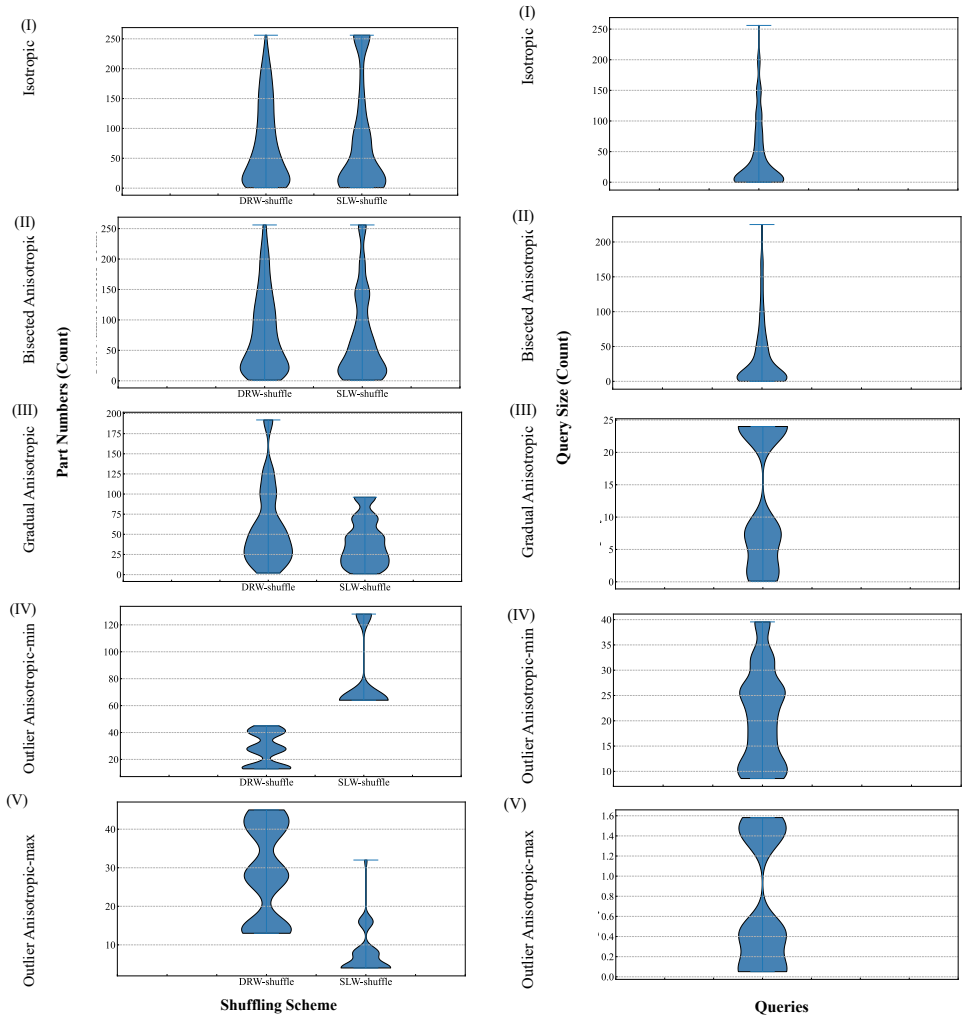
(a) Part numbers distribution for the 4D-$[32]^4$ dataset.

(b) Query Size Distribution.

Figure 5: (a)-Part numbers count per query shape for the DRW-shuffle and SLW-shuffle schemes, $[32]^4$ 4D dataset. The x-axis represents the dataset of part numbers returned as query results per shuffling method (b)-Query size distribution. The x-axis represents the queries dataset

efficiency and security of data retrieval for searchable encryption mechanisms across datasets and query shapes. In most scenarios, reducing the average and variance of returned parts improves both efficiency and security of the system. That is, efficiency and security are aligned. DRW-shuffle demonstrates superior performance only in the setting where dimensions $i > 1$ are large in each query. This is counter intuitive, we usually organize data based on the most important dimension, here the width of the non-indexed dimensions are critical.

We recommend that future research investigates the interactions between different shuffling strategies and index structures that allow for false positives, such as the single range cover [FMET22]. Such systems usually reduce the number of possible ranges that are queryable. A natural solution is to organize data according to these queryable ranges. However, unlike the organizations considered in this work, tuples usually are in more than one range cover.



(a) Part numbers distribution for the $[16]^4$ dataset

(b) Query Size Distribution

Figure 6: (a)- Part numbers count per query shape for the 4D-$[16]^4$ dataset. The x-axis represents the dataset of part numbers returned as query results per shuffling method (b)- Query Size Distribution. The x-axis represents the queries dataset

Acknowledgements

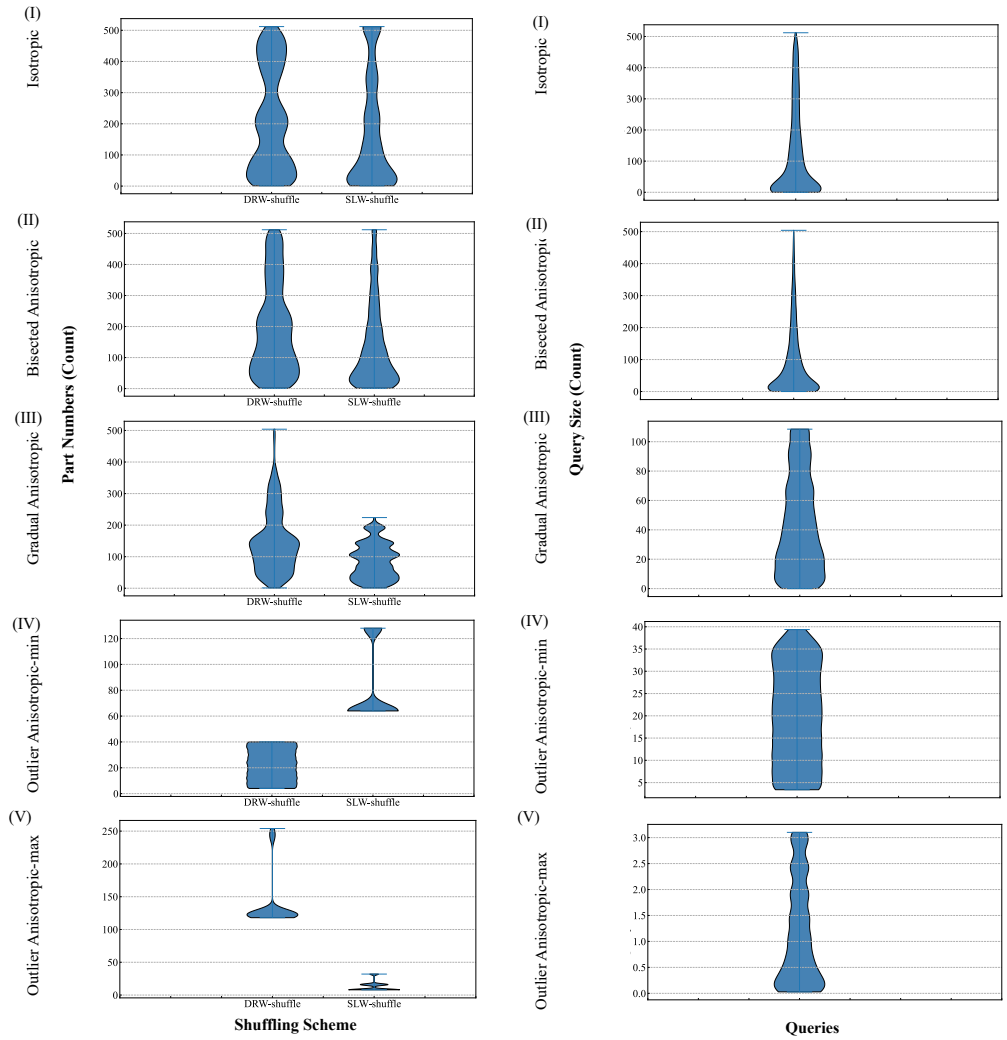
The authors are thankful to the anonymous reviewers for their help in improving the manuscript. The authors are supported by NSF Awards # 2131509, 2141033, 2146852, 2232813, 2333899, and 2341378.

References

[ACF+20] Akshima, David Cash, Francesca Falzon, Adam Rivkin, and Jesse Stern. Multidimensional database reconstruction from range query access patterns. *Cryptology ePrint Archive*, 2020.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[BFOR08] Mihir Bellare, Marc Fischlin, Adam O'Neill, and Thomas Ristenpart. Deterministic encryption: Definitional equivalences and constructions without random oracles. In *CRYPTO*, pages 360–378, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.



(a) Part numbers distribution for the 3D-$[128]^3$ dataset

(b) Query Size Distribution

Figure 7: (a)- Part numbers count for the DRW-shuffle and SLW-shuffle schemes, $[128]^3$ 3D dataset. The x-axis represents the dataset of part numbers returned as query results per shuffling method (b)- Query Size Distribution. The x-axis represents the queries dataset

[BGI+01] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im) possibility of obfuscating programs. In *CRYPTO*, pages 1–18. Springer, 2001.

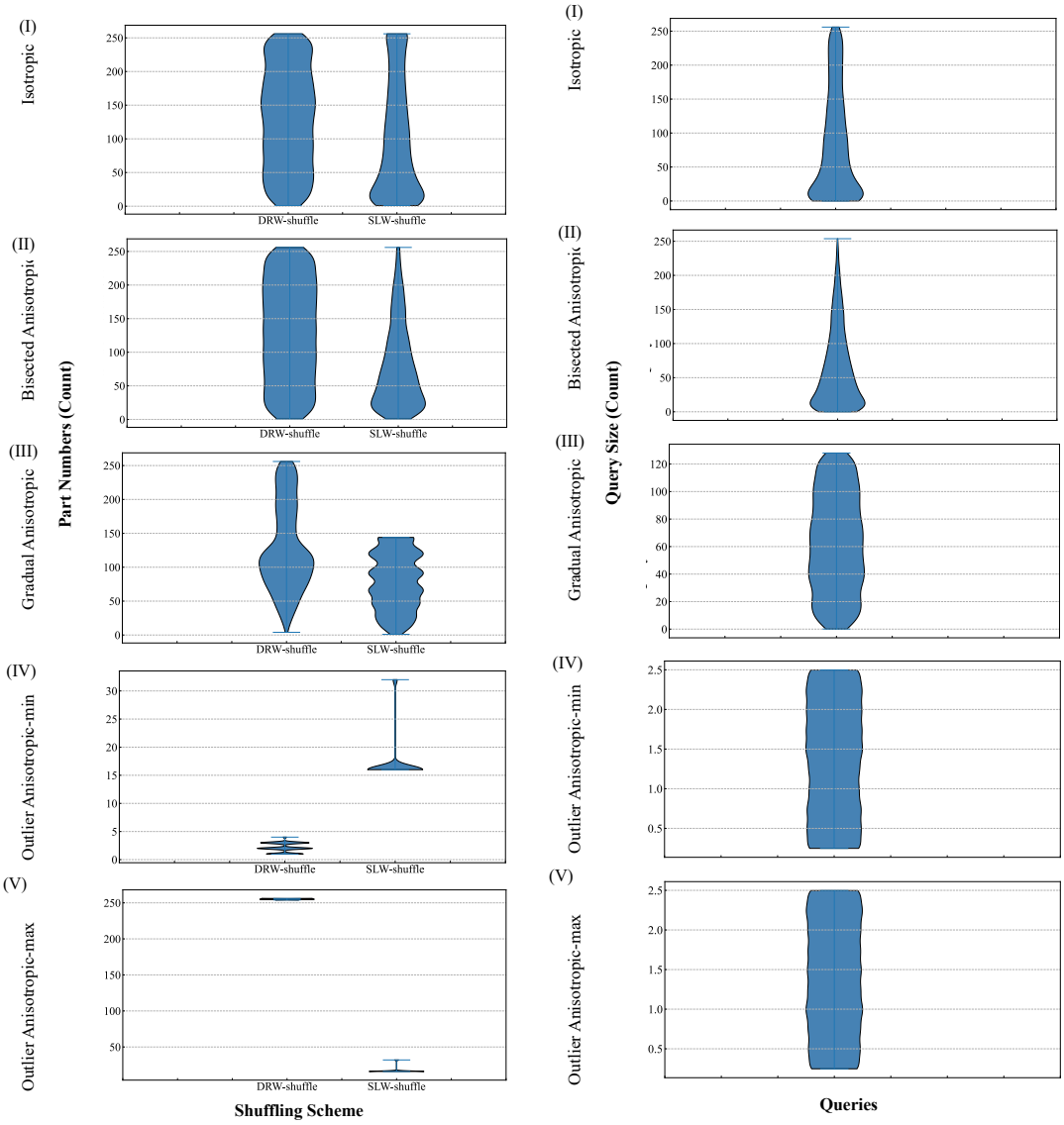
[BHJP14] Christoph Bösch, Pieter Hartel, Willem Jonker, and Andreas Peter. A survey of provably secure searchable encryption. *ACM Computing Surveys (CSUR)*, 47(2):1–51, 2014.

[BIM00] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: Pir with preprocessing. In *CRYPTO*, pages 55–73, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[BIPW17] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can we access a database both locally and privately? In *Theory of Cryptography*, pages 662–693, 2017.

[BKM19] Laura Blackstone, Seny Kamara, and Tarik Moataz. Revisiting leakage abuse attacks. *Cryptology ePrint Archive*, 2019.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*, 1988.



(a) Part numbers distribution for the 2D-$[4096]^2$ dataset

(b) Query Size Distribution

Figure 8: (a)- Part number counts per query shape for $[4096]^2$ 2D dataset. The x-axis represents the dataset of part numbers returned as query results per shuffling method (b)- Query Size Distribution. The x-axis represents the queries dataset

[BW07] Dan Boneh and Brent Waters. Conjunctive, subset, and range queries on encrypted data. In *Theory of Cryptography Conference*, pages 535–554. Springer, 2007.

[CG97] Benny Chor and Niv Gilboa. Computationally private information retrieval (extended abstract). In *STOC*, pages 304—-313, 1997.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[CGPR15] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, pages 668–679, 2015.

[CHR17] Ran Canetti, Justin Holmgren, and Silas Richelson. Towards doubly efficient private information retrieval. In *Theory of Cryptography*, pages 694–726, Cham, 2017. Springer International Publishing.

[CK10] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In *Asiacrypt*, pages 577–594. Springer, 2010.

[CKGS98] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, nov 1998.

[DPP+16] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos Garofalakis. Practical private range search revisited. In *ACM SIGMOD/PODS Conference*, 2016.

[DPPS20] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. SEAL: Attack mitigation for encrypted databases via adjustable leakage. In *USENIX Security*, pages 2433–2450, 2020.

[FMC+20] Francesca Falzon, Evangelia Anna Markatou, David Cash, Adam Rivkin, Jesse Stern, and Roberto Tamassia. Full database reconstruction in two dimensions. In *CCS*, pages 443–460, 2020.

[FMET22] Francesca Falzon, Evangelia Anna Markatou, Zachary Espiritu, and Roberto Tamassia. Range search over encrypted multi-attribute data. *Proc. VLDB Endow.*, 16(4):587–600, dec 2022.

[FMET23] Francesca Falzon, Evangelia Anna Markatou, Zachary Espiritu, and Roberto Tamassia. Range search over encrypted multi-attribute data. In *VLDB*, 2023. `https://eprint.iacr.org/2022/1076`.

[FVY+17] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K Cunningham. SoK: Cryptographically protected database search. In *IEEE Security and Privacy*, pages 172–191, 2017.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[GGH+16] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. *SIAM Journal on Computing*, 45(3):882–929, 2016.

[GKL+20] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. In *USENIX Security*, pages 2451–2468, 2020.

[GLMP18] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *CCS*, pages 315–331, 2018.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, pages 182–194, 1987.

[GPP23] Zichen Gui, Kenneth G. Paterson, and Sikhar Patranabis. Rethinking searchable symmetric encryption. In *IEEE Security and Privacy*, 2023.

[GPPW23] Zichen Gui, Kenneth G Paterson, Sikhar Patranabis, and Bogdan Warinschi. SWiSSSE: System-wide security for searchable symmetric encryption. *PoPETS*, 2023.

[GRS17] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. Why your encrypted database is not secure. In *Proceedings of the 16th workshop on hot topics in operating systems*, pages 162–168, 2017.

[KKM+22] Seny Kamara, Abdelkarim Kati, Tarik Moataz, Thomas Schneider, Amos Treiber, and Michael Yonli. Sok: Cryptanalysis of encrypted search with LEAKER - a framework for LEakage AttacK Evaluation on Real-world data. In *Euro S&P*, 2022.

[KMPP22] Evgenios M Kornaropoulos, Nathaniel Moyer, Charalampos Papamanthou, and Alexandros Psomas. Leakage inversion: Towards quantifying privacy in searchable encryption. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1829–1842, 2022.

[KO97] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *FOCS*, pages 364–373, 1997.

[LMP18] M. Lacharité, B. Minaud, and K. G. Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *IEEE Security and Privacy*, pages 297–314, 2018.

[LMW23] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. Doubly efficient private information retrieval and fully homomorphic ram computation from ring lwe. In *STOC*, page 595–608, 2023.

[MAAM20] Toufique Morshed, Md Momin Al Aziz, and Noman Mohammed. Cpu and gpu accelerated fully homomorphic encryption. In *IEEE HOST*, pages 142–153. IEEE, 2020.

[MFET23] Evangelia Anna Markatou, Francesca Falzon, Zachary Espiritu, and Roberto Tamassia. Attacks on encrypted response-hiding range search schemes in multiple dimensions. *PoPETS*, 2023.

[MFTS21] Evangelia Anna Markatou, Francesca Falzon, Roberto Tamassia, and William Schor. Reconstructing with less: Leakage abuse attacks in two dimensions. In *CCS*, pages 2243–2261, 2021.

[MGW87] Silvio Micali, Oded Goldreich, and Avi Wigderson. How to play any mental game. In *Proceedings of the Nineteenth ACM Symp. on Theory of Computing, STOC*, pages 218–229, 1987.

[MT19] Evangelia Anna Markatou and Roberto Tamassia. Full database reconstruction with access and search pattern leakage. In *International Conference on Information Security*, pages 25–43. Springer, 2019.

[MVA+23] Sujaya Maiyya, Sharath Chandra Vemula, Divyakant Agrawal, Amr El Abbadi, and Florian Kerschbaum. Waffle: An online oblivious datastore for protecting data access patterns. *Proceedings of the ACM on Management of Data*, 1(4):1–25, 2023.

[pyc23] pyca/cryptography. Python cryptography library 40.0.2. https://cryptography.io, 2023.

[Rus23] Florin Rusu. Multidimensional array data management. *Foundations and Trends® in Databases*, 12(2-3):69–220, 2023.

[SWP00] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *IEEE Security and Privacy*, pages 44–55. IEEE, 2000.

[TKMZ13] Stephen Lyle Tu, M Frans Kaashoek, Samuel R Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. In *VLDB*. Association for Computing Machinery (ACM), 2013.