Structured Encryption and Distribution-aware Leakage Suppression*

Marilyn George
MongoDB Research

Seny Kamara
Brown University, MongoDB Research

Tarik Moataz
MongoDB Research

Zachary Espiritu
MongoDB Research

Abstract. A leakage suppressor is a compiler that transforms a structured encryption (STE) scheme into a new scheme with an improved leakage profile. General-purpose suppressors for the query equality (qeq) pattern—which reveals if and when two queries are the same—were given for both static (Kamara et. al, *Crypto '18*) and dynamic (George et. al, *Eurocrypt '19*) encrypted structures. While the schemes that result from these suppressors are asymptotically efficient, they are not practical due to large constants in their query complexity.

In this work, we propose a new query equality suppressor for dictionary encryption schemes that results in practical qeq-hiding encrypted dictionaries at the cost of revealing the distribution of the queries. The resulting constructions are *distribution-aware*, in the sense that they make use of the query distribution, and *distribution-leaking* in the sense that they also reveal it. We show how to instantiate and optimize our suppressor for query distributions that are Zipf-distributed, resulting in a scheme with $O(1)$ online query complexity at the cost of a rebuild with $O(m \log^2 m / \log\log m)$ complexity, where m is the size of the input dictionary.

Keywords. structured encryption, leakage suppression, oblivious dictionaries.

1 Introduction

A structured encryption (STE) scheme encrypts a data structure in such a way that it can be privately queried. STE schemes can be used to design sub-linear

searchable symmetric encryption (SSE) schemes [15,17,21,37,40], encrypted relational databases [18,36,44], encrypted non-relational databases [45], encrypted blockchain databases [1], and end-to-end encrypted applications [23,47]; to name a few examples. Like all sub-linear encrypted search solutions, STE schemes typically achieve efficiency by leaking some information about the data and/or queries which is formally captured by a leakage profile.

Leakage suppression. In [46], Kamara, Moataz and Ohrimenko initiated the study of leakage suppression which studies general-purpose methods that suppress the leakage of STE schemes. These methods usually consist of an encoder that transforms data structures and of a compiler or suppressor that transforms an STE scheme with a particular leakage profile into a new scheme with less leakage. Note that, typically, leakage suppression focuses on *general-purpose* techniques that *completely* suppress a particular leakage pattern like the query equality (i.e., if and when a query is repeated). For example, [46] and its extension to the dynamic case by George, Kamara and Moataz [26] describe a query equality suppressor for a large class of STE schemes by abstracting and generalizing the square root ORAM construction of Goldreich and Ostrovsky [27].

While the suppressors of [46] and [26] result in schemes that are asymptotically better than black-box ORAM simulation and even comparable to some custom oblivious data structures, they are still not practical. In particular, the suppressed schemes always have high communication costs. More concretely, the communication overhead of queries can be over $15\times$ that of the pre-transformed qeq-revealing construction [26]. In addition, we know that any STE scheme that leverages ORAM as a building block incurs at least a logarithmic computation and communication overhead given existing ORAM lower bounds [27, 51]. Existing work also shows that similar bounds apply to any STE scheme that is qeq-hiding [60]. Given this state of affairs, it is natural to ask:

> *Are there practical encrypted dictionary constructions that fully suppress query equality leakage while also supporting fast queries?*

Epoch-based leakage suppression. In this work, we address this question in several respects. First, we make the case for an epoch-based leakage suppression model for the query equality leakage. Specifically, we construct encrypted dictionaries that support one epoch of query-equality-hiding queries before they have to be reset or rebuilt in some way. Epoch-based query-equality-hiding has already been used in qeq suppressors [26, 46]. Squareroot ORAM can also be viewed as an instance of an epoch-based query-equality-hiding suppressor [27].

Interestingly, in the quest for practical leakage suppression, epoch-based approaches have some clear advantages: first, purely from a theoretical perspective, if we allow for a periodic expensive rebuild, we could offload some of the (necessary) costs of leakage suppression in order to achieve fast online queries. Additionally, from a real-world perspective, this design paradigm of periodic performance degradation is well-understood in practical settings such as database backup and maintenance.

Replication-based qeq-hiding. Another useful approach for query- equality-hiding is *replication* — or simply making copies of label-value pairs in the dictionary and querying an unused copy of the label-value pair at query time. Although replication has some practical drawbacks, like increased server storage and client state, it can however enable a qeq-suppressing scheme to have the same 'online' query complexity as a qeq-leaking scheme.

While replication for qeq-hiding has appeared in prior literature, most prominently in the Pancake scheme by Grubbs, Khandelwal, Lacharité, Brown, Li, Agarwal and Ristenpart [28] and the Waffle scheme by Maiyya, Vemula, Agrawal, El Abbadi and Kerschbaum [54]; it has not been studied as a general technique for provable leakage suppression. For example, while both Pancake and Waffle have small server storage and communication overhead, the total client-side state is at least linear in the size of the dictionary. Additionally, Pancake achieves a security notion known as ROR-CDDA while Waffle achieves (α, β)-uniformity, both of which are not comparable to the standard leakage-based definitions in structured encryption.

On the other hand, our replication-based suppressor is provably qeq-hiding and allows for multiple tradeoffs. In fact, the bulk of our technical challenge lies in reducing both the server storage and client state simultaneously while preserving security. We compare our work to Pancake in more detail in the full version of the paper. We also present a high-level overview of our suppression techniques in Figure 2 and a more detailed discussion in Section 1.2.

Distribution-aware leakage suppression. One of the tradeoffs we explore towards more practical constructions is using the client's query distribution. Our resulting constructions are *distribution-aware*, in the sense that they make use of the query distribution, and *distribution-leaking* in the sense that they also reveal it. As we will see, using (and therefore leaking) the query distribution can lead to non-trivial efficiency improvements—especially with respect to server storage—but it can be quite subtle and requires some care. In some settings, however, distribution leakage might be justified. This is the case, for example, in settings where the adversary already knows the query distribution from auxiliary information. Consider a case where an encrypted structure is used to index a collection of English documents. It might be the case that the adversary already knows the language of the documents via auxiliary information so revealing it via the scheme might not add to the adversary's knowledge.

Another, perhaps more subtle scenario, is in the context of leakage attacks. Specifically, almost all leakage attacks in the literature assume knowledge of the query distribution. Therefore, if one believes that a particular leakage attack is applicable then one must believe that knowledge of the distribution is publicly available and, therefore, leaking it does not improve the attack. Note that we are not suggesting here that one should casually leak the distribution; on the contrary one should only use such a scheme if the attack *given access to the query distribution* still has very low recovery rates. What we *are* saying is that in such scenarios, the fact that the scheme reveals the distribution does not necessarily improve the attack since the adversary is already assumed to know it.

1.1 Our Contributions

In this work, we revisit the problem of query-equality-hiding encrypted dictionaries in several respects. In particular, we seek practical constructions for query-equality-hiding encrypted dictionaries. To achieve this, we propose a *replication-based* leakage suppressor for the query equality pattern. Note that by practical, here, we mean schemes with: (1) constant online query complexity, which is not achievable via existing full suppressors [26,46]; and (2) sub-linear client-side storage which is not achieved by existing replication-based constructions like Pancake [29] and Waffle [54]. We summarize our contributions below.

Shuffle-based rebuild compiler. In order to support an epoch-based qeq-hiding dictionary scheme, we construct an oblivious shuffle-based rebuild compiler SRC. At a high-level SRC obliviously shuffles and re-encrypts the encrypted data structure, thereby suppressing all correlations between queries before and after the rebuild. Concretely, we instantiate SRC using the CacheShuffle protocol from Patel, Persiano and Yeo [59] in order to leverage its tradeoffs between small client state and total communication complexity. In particular, CacheShuffle uses $O(N \log_S N)$ communication to obliviously shuffle an array of size N with $O(S)$ client state, which allows us to make this tradeoff in our final suppressor.

Distribution-aware qeq suppressor. We design a distribution-aware query equality suppressor ERS using a variety of techniques including distribution-aware replication, counter functions, sketching schemes, a client-side cache and a leakage-free rebuild (See Figure 2). For any input dictionary and query distribution, our suppressor always outputs a qeq-hiding static dictionary scheme. However, the overall storage and communication of the resulting scheme will depend on the specific instantiations of our techniques chosen for the input query distribution.

Instantiation for the Zipf distribution. Though one could in theory use a fixed instantiation of our techniques for any query distribution, custom instantiations yield better storage overheads. In particular, our techniques perform better when instantiated for skewed distributions, which are common in practice. Towards this end, we show how to instantiate our suppressor for the Zipf distribution. The Zipf distribution captures settings where a small number of labels are more likely to be queried than others, and has been shown to model publicly available query logs [43]. It has also been used to model the distribution of multi-map data structures in order to study the efficiency of various EMM constructions [5,38,46,61]. The Zipf distribution is also used to model record accesses in the Yahoo! Cloud Serving Benchmark (YCSB) for cloud data services [20].

We instantiate our suppressor in two modes: one with client state that grows as the $(s-1)$-th root of m and the other with client state polylogarithmic in m, where m is the size of the input dictionary and s is the parameter of the Zipf distribution. In Figure 3 we show how the resulting qeq-hiding schemes from both modes compare to existing encrypted dictionary schemes. In particular, we compare to standard (leaky) dictionary encryption scheme Π^{+}_{bas} from Cash et

al. [17][1], the oblivious dictionary scheme Σ_{ODS} from Wang et al. [69], and the Pancake construction from Grubbs et al. [29]. Figure 3 provides a summary of the comparison in terms of storage, communication, queue waiting time, supported operations, and security. We provide a detailed explanation of our comparison in the full version of our paper.

A note on dynamism. A limitation of our suppressor is that it can only output static dictionary encryption schemes. We note, however, that static techniques are important for at least two reasons: (1) there are many sensitive static datasets that need to be protected (e.g., genomic data, archived financial transactions, census and demographic data, archived educational records, criminal records); and (2) static techniques often lead to future work on dynamic techniques. One of the main challenges in designing dynamic suppressors is removing correlations between queries and the supported update operations (e.g., adds, deletes or edits). This challenge is particularly hard to overcome for replication-based suppressors, because updates need to be propagated to all unused replicas while maintaining security. For the special case of edit operations, one could try to adapt techniques from the Pancake system [29]. More precisely, in Pancake, the client edits the values of labels by modifying the base construction in two ways: (1) the client (or its proxy) caches the updated value until the update can be propagated to all the replicas of its label; (2) each get operation on a label is followed by a put on the same label with an encrypted value. If the label in question has a new value in the client/proxy cache, then the encryption is of the new value. On the other hand, if the label has no cached value, then the encryption is just a re-encryption of the value that was retrieved during the get. Once an updated value has been propagated to all the replicas of a label, it is removed from the cache.

One reason we cannot apply this technique directly is that our suppressor maintains only sublinear client-side state and this approach can result in linear client-side storage for some update distributions. If we further want to support the addition and deletion of labels, the problem becomes significantly more complicated. We now have to either use enough replicas at setup time to support future adds or increase and decrease the number of replicas as the underlying dictionary grows and shrinks. Both these approaches would lead to additional leakage and efficiency costs depending on the add and delete distributions. In summary, extending our techniques to the dynamic setting is a non-trivial and interesting direction for future work.

1.2 Distribution-Aware Query Equality Suppression

For a high-level overview of our techniques, see Figure 2.

[1] We note that we modify the original Π^{+}_{bas} scheme to have only $O(1)$ client state for fairness in our comparison since per-label counters are not required when the scheme is used to encrypt a dictionary data structure.

A simple replication-based suppressor. Given infinite server storage and large client state, we can easily use replication to build a query equality suppressor for encrypted dictionary schemes as follows: At setup, the client creates infinite replicas of every label-value pair and tags each replica with an incrementing counter value. Then the client encrypts the replicated label-value pairs using any efficient dictionary encryption scheme that leaks the query equality. Finally the client stores one counter per label to track used replicas. To query a label, the client uses the current counter for that label to query an unused replica and then increments the counter by 1. It is easy to see that the resulting scheme completely suppresses the query equality since the client only queries each replica once. Additionally, the suppressed scheme has the same query complexity as the query equality-leaking scheme used to encrypt the replicated dictionary.

Making server-side replication practical. In practice, however, the server cannot store infinite replicas. This creates two challenges for us: (1) how many replicas should we create given limited server storage? and (2) what do we do after the client queries all the replicas? We address (1) using distribution-aware replication to reduce storage and (2) using a leakage-free rebuild to reuse storage.

Distribution-aware replication. At setup, if we know the number of queries that the client will make for each label, we can create exactly the required number of replicas such that the client can always query an unused replica. In our suppressor, we use the query distribution to estimate how many times a label will be queried and set up the appropriate number of replicas. Intuitively, the number of replicas created for a label is directly proportional to its query probability, so labels with higher query probabilities are replicated more than labels with lower query probabilities.

Leakage-free rebuild. After setup, the total number of replicas is fixed. Each client query consumes one replica and eventually the encrypted dictionary will run out of replicas. At this point it is useful to be able to refresh and reuse the existing replicas so we can continue supporting queries without revealing the query equality. Our suppressor uses a shuffle-based rebuild protocol to re-encrypt the existing replicas at the end of every epoch so they can be reused. This shuffle-based rebuild protocol is *leakage-free* because it does not introduce any additional leakage during the rebuild of the encrypted structure.

Making client-side state sub-linear. The bigger challenge for the simple replication-based suppressor is that the client must store and update one counter for each label in the dictionary. This state is linear in the size of the dictionary, and hence defeats the purpose of outsourcing the dictionary to the server. We now have to deal with a multi-faceted challenge: how do we make the client state sub-linear while maintaining the correctness and security of the replication-based suppressor?

State as a function of time. Our main observation is that the client locally maintains the current time step, and if we can compute the replica counters for each label from that time step, we can reduce the client state. We then introduce *counter functions* which take as input the client's current time step and return a (replica) counter. A counter function can be visualized as a simple step function, shown in Figure 1.[2] Each point on the x axis is a time step and each point on the y axis is the corresponding counter. Notice the following properties of a counter function:

- counter values are monotonically increasing, past values never repeat.
- the number of steps determines the number of values the counter function takes.
- a counter value can repeat if two time steps map to the same 'step' of the counter function.

The client uses the output of these counter functions to create replicas at setup time, and to determine which replica to query at any given time t. Counter functions can be represented compactly so this helps reduce the client state. However, we still have two issues (1) if we use the same counter function for all labels, then every label has the same number of replicas and we do not make use of the query distribution; and (2) if two time steps map to the same counter for the same label, then the query equality is leaked. Our next techniques will address these issues.

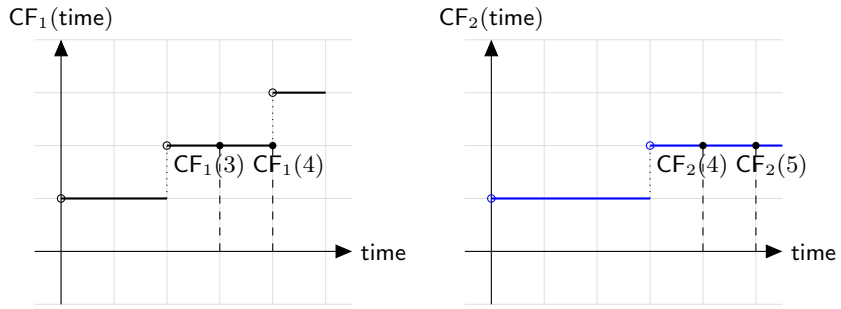

Fig. 1. Step counter functions with step lengths 2 and 3 respectively. Some query counters will map to the same replica depending on the step length. For example, $\mathsf{CF}_1(3) = \mathsf{CF}_1(4) = 2$, and $\mathsf{CF}_2(4) = \mathsf{CF}_2(5) = 2$.

State as a distribution-aware function of time. In order to use the query distribution to create and query replicas, the client must first be able to store the

[2] An initial sketch for Figure 1 was created using Generative AI.

query distribution. Since the query distribution could also be linear in the size of the data structure, we are back where we started.

We then add the following techniques to support distribution-aware replication with small client state:

- *sketching* to create a compact queryable representation of the query distribution,
- *counter choice function* to choose a counter function based on the query probability of a label. Intuitively, labels with higher query probabilities should be mapped to counter with shorter step lengths. This will ensure that they have more replicas at setup time and that there are not many time steps that map to the same replica counter.

One final challenge with distribution-aware replication is that we never know what the actual sequence of queries will be. There is always a non-zero probability that the worst case sequence of accesses could occur — in our particular case, that the counter values for a label at two different time steps collide and map to the same replica — and lead to leaking the query equality. Then we must add some last techniques to cover for the worst case.

Maintaining worst-case security. In the worst case, a label's counter function could repeat a counter value that was already queried. Since the input scheme to our suppressor leaks the query equality, this would reveal the query equality to the server and break security. Then we add the following:

- a *replica cache* to maintain correctness when the replica counter returned was already used,
- *dummy replicas* to suppress query equality when the replica counter returned was already used.

At query time, the client checks the replica cache before querying the server for a replica. To keep the cache small, we remove used replicas from the cache when the current step ends. When a replica is found in the cache the client queries an unused dummy replica on the server in order to maintain security. These dummy replicas are instantiated at setup and contribute to the total storage overhead of replication. By adding these dummy replicas to cover the worst case, notice that we have ensured that the suppressed scheme is secure regardless of the client's query sequence.

Putting it all together. Our query-equality suppressor ERS uses all the above techniques to achieve practical efficiency. We describe ERS in detail in Section 6 and show how to instantiate ERS for Zipf query distributions in Section 7.

2 Related Work

We already discussed work on general leakage suppression [26, 46, 66, 67] and replication-based dictionary constructions which mitigate query equality leakage [29, 54] so we focus here on other related work.

Technique	*Description*	*Advantages*	*Disadvantages*
Replication	creates replicas of every (ℓ, v); queries a new replica at each time step	partially qeq-hiding	uses server storage; needs additional client state to track replicas
Counter functions (CF_i)	functions mapping each time-step to an 'active' replica	tracks active replica with small client state	leaks qeq if an active replica is queried twice; needs additional work for security
Counter choice function ($\mathsf{chooseCF}$)	maps a query probability to a CF	utilizes server storage better when using CFs	needs additional client state for query distribution; needs additional work for compact client state
Sketch	represents query distribution compactly	utilizes client state better when using $\mathsf{chooseCF}$, CFs	uses additional client state
Replica cache	stores queried replicas on the client until they expire	maintains correctness for used replicas	uses additional client state
Dummy replicas	queried in lieu of real replica when valid replica is in cache	maintains security for used replicas	uses additional server storage
Leakage-free rebuild	re-encrypts all replicas after an epoch	allows reuse of replicas without leaking qeq	increases total communication complexity

Fig. 2. A summary of the techniques we use for our suppressor ERS. For a detailed description, see Section 1.2.

Scheme		Client Storage	Server Storage	Online Query	Amortized Query	Online Rounds	Amortized Rounds	Query Latency	Supports Get?	Supports Edit or Update?	Security
Π^{+}_{bas}	[17]	$O(1)$	$O(m)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$\perp$	✓	✓	Leaks dsize, oeq
Σ_{ODS}	[69]	$O(\log m) \cdot \omega(1)$	$O(m)$	$O(\log^2 m)$	$O(\log^2 m)$	$O(\log m)$	$O(\log m)$	$\perp$	✓	✓	Leaks dsize
Pancake	[28]	$\Omega(m)$	$O(m)$	$O(B)$	$O(B)$	$O(1)$	$O(1)$	$\Omega(1)^{\dagger}$	✓	✓	ROR-CDDA
Ω_{bas} ($\log^2$ state)		$O(\log^2 m)$	$O(H_{m,s} \cdot m)$	$O(1)$	$O\left(\frac{\log^2 m}{\log\log m}\right)$	$O(1)$	$O\left(\frac{\log^2 m}{\log\log m}\right)$	$\perp$	✓	✗	Leaks dsize
Ω_{bas} ($m^{\frac{1}{s-1}}$ state, $s > 2$)		$O(m^{\frac{1}{s-1}})$	$O(H_{m,s} \cdot m)$	$O(1)$	$O(\log m)$	$O(1)$	$O(\log m)$	$\perp$	✓	✗	Leaks dsize

Fig. 3. Comparison of our suppressors initialized using Π^{+}_{bas} [17] and a Zipf query distribution to existing encrypted dictionary schemes, where m is the number of label-value pairs in the input dictionary and s is the parameter of the Zipf distribution. We use † to denote an estimate based on the details provided in the paper and the artifact.

Structured encryption. Structured encryption was proposed by Chase and Kamara [19] as a generalization of searchable symmetric encryption (SSE) which was introduced by Song, Wagner and Perrig [63]. The first optimal-time constructions and the standard security definition for SSE was proposed by Curtmola, Garay, Kamara and Ostrovsky [21]. STE schemes have since been improved along various dimensions including expressiveness [15, 24, 25, 37, 58], efficiency [17, 21], dynamism [6, 39, 40], concurrency [2, 3, 13], security [5, 9, 12, 34, 62, 65, 68], and locality / page efficiency [7, 11, 16, 22].

Cryptanalysis. Leakage attacks have been designed against a variety of encrypted search algorithms, including solutions based on property preserving encryption, structured encryption and oblivious RAM. With respect to searchable and structured encryption, specifically, several leakage patterns have been studied including the query equality pattern [41, 48, 53, 57], the response identity pattern [10, 14, 32, 35, 50, 70], the volume pattern [10, 30, 31, 33, 49], or combinations of multiple patterns [55] under a variety of assumptions (see [42] for a recent survey).

Distribution-aware design. While distribution-aware design has been used previously in the literature, most notably in Pancake [29] and Waffle [54], in this work we leverage distribution-awareness, distribution leakage and replication to design a general-purpose suppressor for encrypted dictionary schemes. Distribution-aware techniques have also recently been used to improve the efficiency of Private Information Retrieval (PIR) for skewed popularity distributions [52].

Oblivious RAM and data structures. An Oblivious RAM (ORAM) scheme can be viewed as a qeq-hiding encrypted data structure, usually an array or a dictionary. For example, the original squareroot ORAM [27] can be viewed as a qeq-hiding encrypted array and Path ORAM [64] can be viewed as a qeq-hiding array or dictionary. However, the closest related work to ours is the work on oblivious data structures, which tailors tree-based ORAM techniques to create specific oblivious data structures, including oblivious dictionaries [69].

Oblivious sorts and shuffles. In this paper, we propose a shuffle-based rebuild compiler. Our compiler can be instantiated using any oblivious shuffle, such as the Melbourne Shuffle [56] or CacheShuffle [59]. In our query equality suppressor we instantiate the shuffle using CacheShuffle to leverage its tradeoffs between small client state and total communication complexity. In particular, CacheShuffle uses $O(N \log_S N)$ communication to obliviously shuffle an array of size N with $O(S)$ client state, which allows us to make this tradeoff in our final suppressor. Existing rebuild compilers in the literature such as the RBC [46] instantiate an oblivious shuffle using an oblivious sort such as the Ajtai-Komlos-Szemeredi sorting network [4] or Batcher sort [8].

3 Preliminaries

Notation. We denote the security parameter as k, and all algorithms run in time polynomial in k. The set of all binary strings of length n is denoted as $\{0,1\}^n$, and the set of all finite binary strings as $\{0,1\}^*$. $[n]$ is the set of integers $\{1,\ldots,n\}$, and $2^{[n]}$ is the corresponding power set. The output x of an algorithm $\mathcal{A}$ is denoted by $x \leftarrow \mathcal{A}$. Given a sequence $\mathbf{v}$ of n elements, we refer to its ith element as v_i or $\mathbf{v}[i]$. If S is a set then $\#S$ refers to its cardinality. If s is a string then $|s|_2$ refers to its bit length.

The word RAM. Our model of computation is the word RAM. In this model, we assume memory holds an infinite number of w-bit words and that arithmetic, logic, read and write operations can all be done in $O(1)$ time. We denote by $|x|_w$ the word-length of an item x; that is, $|x|_w = |x|_2/w$. Here, we assume that $w = \Omega(\log k)$.

Query distributions. In this paper we will use $\mathcal{Q}_{\mathsf{DS}}$ to denote a discrete probability distribution over the query space $\mathbb{Q}_{\mathsf{DS}}$ of the data structure DS. We denote by Q_{DS} a random variable distributed according to $\mathcal{Q}_{\mathsf{DS}}$ and we write $\mathrm{Q}_{\mathsf{DS}} \sim \mathcal{Q}_{\mathsf{DS}}$. In the presence of multi-dimensional random variable, we denote by $\mathbf{Q}^n_{\mathsf{DS}}$ the sequence of n random variables $(\mathrm{Q}_{\mathsf{DS},1}, \cdots, \mathrm{Q}_{\mathsf{DS},n})$, for $n \in \mathbb{N}_{\geq 1}$. When DS and n are clear from context, we drop the subscript DS and superscript n.

Probabilities. Given a discrete random variable X, we denote its probability mass function by $p_X(x)$ or $p(x)$ when X is clear. Given two discrete random variables X and Y, we denote the distribution of X conditioned on $Y = y$ for some y over the range of Y, by $p_X(x \,|\, y)$ or $p(x \,|\, y)$ when X is clear.

Abstract data types. An *abstract data type* specifies the functionality of a data structure. Examples include sets, dictionaries (also known as key-value stores or associative arrays) and graphs. The operations associated with an abstract data type fall into one of two categories: query operations, which return information about the objects; and update operations, which modify the objects. If the abstract data type supports only query operations it is *static*, otherwise it is *dynamic*.

We model a data type $\mathbf{T}$ as a collection of four spaces: the object space $\mathbb{D} = \{\mathbb{D}_k\}_{k\in\mathbb{N}}$, the query space $\mathbb{Q} = \{\mathbb{Q}_k\}_{k\in\mathbb{N}}$, the response space $\mathbb{R} = \{\mathbb{R}_k\}_{k\in\mathbb{N}}$ and the update space $\mathbb{U} = \{\mathbb{U}_k\}_{k\in\mathbb{N}}$. We also define the query map $\mathsf{qu} : \mathbb{D}\times\mathbb{Q} \to \mathbb{R}$ and the update map $\mathsf{up} : \mathbb{D} \times \mathbb{U} \to \mathbb{D}$ to represent operations associated with the dynamic data type. We refer to the query and update spaces of a data type as the operation space $\mathbb{O} = \mathbb{Q} \cup \mathbb{U}$. For a static data type that does not support updates, $\mathbb{O} = \mathbb{Q}$. When specifying a data type $\mathbf{T}$ we will often just describe its maps qu, up from which the object, query, response and update spaces can be deduced. The spaces are ensembles of finite sets of finite strings indexed by the security parameter. We assume that $\mathbb{R}$ includes a special element $\perp$ and that $\mathbb{D}$ includes an empty object d_0 such that for all $q \in \mathbb{Q}$, $\mathsf{qu}(d_0, q) = \perp$.

Data structures. A type-$\mathbf{T}$ data *structure* is a representation of data objects in $\mathbb{D}$ in some computational model (as mentioned, here it is the word RAM). Typically, we assume that there exists an efficient algorithm Query that computes the function qu. For example, the dictionary type can be represented using various data structures depending on which queries one wants to support efficiently. Hash tables support Get and Put in expected $O(1)$ time whereas balanced binary search trees support both operations in worst-case $O(\log n)$ time. We use the notation $\mathsf{DS} \equiv d$ to denote that the data structure DS instantiates the data object d. This implies that for all $q \in \mathbb{Q}$, $\mathsf{qu}(d, q) = \mathsf{Query}(\mathsf{DS}, q)$.

Data structure logs. Given a structure DS that instantiates an object d, we will use the sequence of update operations needed to create a new structure DS' that also instantiates d. We refer to this as the *data structure log* of DS and assume the existence of an efficient algorithm Log that takes as input DS and outputs a tuple of update operations $(u_1, \ldots, u_n)$ such that adding $u_1, \ldots, u_n$ to an empty structure results in some $\mathsf{DS}' \equiv d$.

Dictionaries. A dictionary structure DX of capacity n holds a collection of n label/value pairs $\{(\ell_i, v_i)\}_{i \leq n}$ and supports get and put operations. We write $v_i := \mathsf{DX}[\ell_i]$ to denote getting the value associated with label ℓ_i and $\mathsf{DX}[\ell_i] := v_i$ to denote the operation of associating the value v_i in DX with label ℓ_i. Given a query distribution $\mathcal{Q}$ over the support space $\mathbb{Q}_{\mathsf{DX}}$, we define $\mathcal{Q} \stackrel{\circ}{=} (p_\ell)_{\ell \in \mathbb{Q}_{\mathsf{DX}}}$.

Priority queues. A priority queue Q of capacity n holds n elements, where each element e_i is sorted according to its priority value p_i. A min-priority queue supports add, find minimum, and delete minimum operations. A max-priority queue supports add, find maximum, and delete maximum operations.

Cryptographic protocols. We denote by $(\mathsf{out}_A, \mathsf{out}_B) \leftarrow \Pi_{A,B}(X; Y)$ the execution of a two-party protocol Π between parties A and B, where X and Y are the inputs provided by A and B, respectively; and out_A and out_B are the outputs returned to A and B, respectively.

4 Definitions

4.1 Structured Encryption

In this work, we consider both standard and distribution-aware rebuildable STE schemes. Both types of constructions can be captured with the following slight modification to the standard STE syntax to include the query distribution $\mathcal{Q}$.

Definition 1 (Structured encryption). *A type-$\mathbf{T}$ structured encryption scheme* $\mathsf{STE} = (\mathsf{Setup}, \mathsf{Operate}_{\mathbf{C},\mathbf{S}}, \mathsf{Rebuild}_{\mathbf{C},\mathbf{S}})$ *consists of an algorithm and two two-party protocols that work as follows:*

– $(K, st, \mathsf{EDS}) \leftarrow \mathsf{Setup}(1^k, n, \mathcal{Q}, \mathsf{DS})$*: is a probabilistic polynomial-time algorithm that takes as input a security parameter* 1^k*, a query capacity* $n \geq 1$*, the description of a query distribution* $\mathcal{Q}$*, and a type-*$\mathbf{T}$ *structure* DS*. It outputs a secret key* K*, a state* st *and an encrypted structure* EDS*.*
– $\big(st', r; \mathsf{EDS}'\big) \leftarrow \mathsf{Operate}_{\mathbf{C},\mathbf{S}}\big(K, st, \mathsf{op}; \mathsf{EDS}\big)$*: is a two-party protocol executed between a client and a server where the client inputs a secret key* K*, a state* st *and an operation* op *and the server inputs an encrypted structure* EDS*. The client receives as output an updated state* st' *and a response* $r \in \mathbb{R} \cup \perp$ *while the server receives an updated encrypted structure* EDS'*.*
– $\big(K', st'; \mathsf{EDS}'\big) \leftarrow \mathsf{Rebuild}_{\mathbf{C},\mathbf{S}}\big(K, st; \mathsf{EDS}\big)$*: is a two-party protocol executed between a client and a server where the client inputs a secret key* K *and a state* st *and the server inputs an encrypted structure* EDS*. The client receives as output an updated secret key* K' *and state* st' *and the server receives an updated encrypted structure* EDS'*.*

Note that the syntax of a standard rebuildable STE scheme can be recovered by setting $\mathcal{Q} := \perp$. For a semi-dynamic STE scheme, $\mathsf{Operate}$ could be either an Add or a Query protocol, and for a static STE scheme $\mathsf{Operate}$ is always a Query protocol.

Security. We now slightly adapt the standard notion of adaptive semantic security for rebuildable STE schemes [19, 21] to the case of distribution-aware schemes. We use patt_S, patt_O, patt_R to refer to the setup, operation, and rebuild leakages respectively.

Definition 2 (Security). *Let* $\mathsf{STE} = (\mathsf{Setup}, \mathsf{Operate}_{\mathbf{C},\mathbf{S}}, \mathsf{Rebuild}_{\mathbf{C},\mathbf{S}})$ *be a structured encryption scheme and consider the following probabilistic experiments where* $\mathcal{C}$ *is a stateful challenger,* $\mathcal{A}$ *is a stateful adversary,* $\mathcal{S}$ *is a stateful simulator,* $\Lambda = (\mathsf{patt}_\mathsf{S}, \mathsf{patt}_\mathsf{O}, \mathsf{patt}_\mathsf{R})$ *is a leakage profile,* $n \geq 1$ *is a query capacity,* $\mathcal{Q}$ *is a distribution, and* $z \in \{0,1\}^*$*:*

$\mathbf{Real}_{\mathsf{STE},\mathcal{C},\mathcal{A}}(k)$*: given* z, n*, and* $\mathcal{Q}$*, the adversary* $\mathcal{A}$ *outputs a structure* DS *and receives* EDS *from the challenger, where* $(K, st, \mathsf{EDS}) \leftarrow \mathsf{Setup}(1^k, n, \mathcal{Q}, \mathsf{DS})$*.* $\mathcal{A}$ *then adaptively chooses a polynomial-size sequence of operations* $(\mathsf{op}_1, \ldots \mathsf{op}_m)$*. For all* $1 \leq i \leq m$*, the challenger and adversary do the following:*

1. *if* op_i *is a query or an update* $\mathsf{op}_i \in \mathbb{O}$*, they execute* $\mathsf{Operate}_{\mathcal{C},\mathcal{A}}\big(K, st, \mathsf{op}_i; \mathsf{EDS}\big)$*;*
2. *if* op_i *is a rebuild operation, they execute* $\mathsf{Rebuild}_{\mathcal{C},\mathcal{A}}\big(K, st; \mathsf{EDS}\big)$*.*

Finally, $\mathcal{A}$ *outputs a bit* b *that is output by the experiment.*

$\mathbf{Ideal}_{\mathsf{STE},\mathcal{A},\mathcal{S}}(k)$*: given* z, n*, and* $\mathcal{Q}$*, the adversary* $\mathcal{A}$ *outputs a structure* DS*. Given* $\mathsf{patt}_\mathsf{S}(\mathsf{DS})$*, the simulator returns an encrypted structure* EDS *to* $\mathcal{A}$*.* $\mathcal{A}$ *then adaptively chooses a polynomial-size sequence of operations* $(\mathsf{op}_1, \ldots, \mathsf{op}_m)$*. For all* $1 \leq i \leq m$*, the simulator and the adversary do the following:*

1. *if* op_i *is a query or an update* $\mathsf{op}_i \in \mathbb{O}$*, they execute* $\mathsf{Operate}_{\mathcal{S},\mathcal{A}}\big(\mathsf{patt}_\mathsf{O}(\mathsf{DS}, \mathsf{op}_1, \mathsf{op}_2, \ldots, \mathsf{op}_i); \mathsf{EDS}\big)$*;*
2. *if* op_i *is a rebuild operation, they execute* $\mathsf{Rebuild}_{\mathcal{S},\mathcal{A}}\big(\mathsf{patt}_\mathsf{R}(\mathsf{DS}); \mathsf{EDS}\big)$*.*

Finally, $\mathcal{A}$ outputs a bit b that is output by the experiment.

We say that STE *is Λ-secure if there exists a* PPT *simulator $\mathcal{S}$ such that for all* PPT *adversaries $\mathcal{A}$, for all $n \geq 1$, for all $\mathcal{Q}$ and all $z \in \{0,1\}^*$,*

$$|\Pr[\mathbf{Real}_{\mathsf{STE},\mathcal{C},\mathcal{A}}(k) = 1] - \Pr[\mathbf{Ideal}_{\mathsf{STE},\mathcal{A},\mathcal{S}}(k) = 1]| \leq \mathsf{negl}(k).$$

Leakage. We use the following leakage patterns in this work. Let $\mathbf{T} = (\mathsf{qu} : \mathbb{D} \times \mathbb{Q} \to \mathbb{R}, \mathsf{up} : \mathbb{D} \times \mathbb{U} \to \mathbb{D})$ be a dynamic data type. We assume that updates can be written as query/response pairs, i.e., $\mathbb{U} = \mathbb{Q} \times \mathbb{R}$.

- the *data size pattern* is the function family $\mathsf{dsize} = \{\mathsf{dsize}_k\}_{k\in\mathbb{N}}$ with $\mathsf{dsize}_k : \mathbb{D}_k \to \mathbb{N}$ such that $\mathsf{dsize}_k(d) = |d|_w$;
- the *query equality pattern* is the function family $\mathsf{qeq} = \{\mathsf{qeq}_{k,t}\}_{k,t\in\mathbb{N}}$ with $\mathsf{qeq}_{k,t} : \mathbb{D}_k \times \mathbb{Q}_k^t \to \{0,1\}^{t\times t}$ such that $\mathsf{qeq}_{k,t}(d, q_1, \ldots, q_t) = M$, where M is a binary $t \times t$ matrix such that for queries q_i and q_j, $M[i,j] = 1$ if $q_i = q_j$ and $M[i,j] = 0$ otherwise;
- the *operation equality pattern* is the function family $\mathsf{oeq} = \{\mathsf{oeq}_{k,t}\}_{k,t\in\mathbb{N}}$ with $\mathsf{oeq}_{k,t} : \mathbb{D}_k \times \mathbb{O}_k^t \to \{0,1\}^{t\times t}$ such that $\mathsf{oeq}_{k,t}(d, \mathsf{op}_1, \ldots, \mathsf{op}_t) = M$, where M is a binary $t \times t$ matrix such that for operations $\mathsf{op}_i = q_i$ or $u_i = (q_i, r_i)$ and $\mathsf{op}_j = q_j$ or $u_j = (q_j, r_j)$, $M[i,j] = 1$ if $q_i = q_j$ and $M[i,j] = 0$ otherwise.

Leakage-free rebuild. We say that an STE scheme STE with leakage profile $\Lambda = (\mathsf{patt}_\mathsf{S}, \mathsf{patt}_\mathsf{O}, \mathsf{patt}_\mathsf{R})$ has a leakage-free rebuild protocol if for all $d \in \mathbb{D}$, for all $\mathsf{DS} \equiv d$, $\mathsf{patt}_\mathsf{R}(\mathsf{DS}) = \perp$.

Correctness. We also modify the standard notion of correctness for rebuildable STE schemes to include the query distribution $\mathcal{Q}$. For an encrypted data structure EDS, we say that EDS instantiates the data object $d \in \mathbb{D}$ if for all $q \in \mathbb{Q}$, $\mathsf{Query}((K, st, q); \mathsf{EDS})$ outputs $\mathsf{qu}(d, q)$ to the client, where K and st are the key and the state for EDS. We denote this as $\mathsf{EDS} \equiv d$.

Definition 3 (Correctness). *Let* $\mathsf{STE} = (\mathsf{Setup}, \mathsf{Query}_{\mathbf{C},\mathbf{S}}, \mathsf{Rebuild}_{\mathbf{C},\mathbf{S}})$ *be a rebuildable structured encryption scheme. We say that* STE *is correct if it satisfies the following:*

- *(static correctness) for all $k \in \mathbb{N}$, for all $n \in \mathbb{N}$, for all $\mathcal{Q}$, for all $d \in \mathbb{D}$, for all* DS *that instantiate d, it holds that:*

$$\Pr[\mathsf{EDS} \equiv d] \geq 1 - \mathsf{negl}(k),$$

  where $(K, st, \mathsf{EDS}) \leftarrow \mathsf{Setup}(1^k, n, \mathcal{Q}, \mathsf{DS})$.
- *(dynamic correctness) for all $k \in \mathbb{N}$, for all $n \in \mathbb{N}$, for all $\mathcal{Q}$, for all $d \in \mathbb{D}$, for all* EDS *that instantiate d, for all $u \in \mathbb{U}$, it holds that:*

$$\Pr\left[\mathsf{EDS}' \equiv \mathsf{up}(d, u)\right] \geq 1 - \mathsf{negl}(k),$$

  where $\left(st'; \mathsf{EDS}'\right) \leftarrow \mathsf{Update}_{\mathbf{C},\mathbf{S}}\left(K, st, u; \mathsf{EDS}\right)$.

– *(rebuild correctness) for all $k \in \mathbb{N}$, for all $n \in \mathbb{N}$, for all $d \in \mathbb{D}$, for all* EDS *that instantiate d, it holds that:*

$$\Pr\left[\,\mathsf{EDS}' \equiv d\,\right] \geq 1 - \mathsf{negl}(k),$$

where $\left(K', st'; \mathsf{EDS}'\right) \leftarrow \mathsf{Rebuild}_{\mathbf{C},\mathbf{S}}\left(K, st; \mathsf{EDS}\right)$ *and K, st are the key and the state of* EDS.

4.2 Query Distribution Sketches

As discussed in section 1, our suppressor uses a compact representation of the query distribution on the client. This is achieved using a sketching scheme for the query distribution which we define below.

Definition 4 (Sketching scheme). *A sketching scheme* $\Delta = (\mathsf{SketchDist}, \mathsf{QueryDist})$ *consists of two probabilistic polynomial time algorithms that work as follows:*

– $\mathsf{sketch} \leftarrow \mathsf{SketchDist}(\mathcal{Q})$ *takes as input a discrete distribution $\mathcal{Q}$ over a finite set $\mathbb{Q}$ and returns a sketch of the distribution* sketch.
– $\tilde{p} \leftarrow \mathsf{QueryDist}(\mathsf{sketch}, q)$ *takes as input a sketch of the distribution $\mathcal{Q}$ and an element $q \in \mathbb{Q}$ and returns a probability $\tilde{p} \in \mathbb{R}$.*

4.3 Oblivious Shuffles

Our suppressor uses a shuffle-based rebuild protocol as a key building block for security. We define an encrypted shuffle and its security properties below.

Definition 5. *An encrypted shuffle* $\mathsf{OShuffle} = (\mathsf{Setup}, \mathsf{Shuffle}_{\mathbf{C},\mathbf{S}}, \mathsf{Dec})$ *consists of two algorithms and a protocol that work as follows:*

– $(K, \mathsf{EA}) \leftarrow \mathsf{Setup}(1^k, \mathsf{A})$*: is a probabilistic polynomial time algorithm that takes as input a security parameter and an input array* A *and outputs a secret key K and an encrypted array* EA *such that* $\mathsf{EA} \equiv \mathsf{A}$.
– $(K'; \mathsf{EA}') \leftarrow \mathsf{Shuffle}_{\mathbf{C},\mathbf{S}}(K, \pi; \mathsf{EA})$*: is a two-party protocol where the client inputs the secret key K and a permutation π and the server inputs an encrypted array* EA*. The client receives as output an updated secret key K' and the server receives an updated encrypted array* EA'
– $\mathsf{A}[i] \leftarrow \mathsf{Dec}(K, \mathsf{EA}[\mathrm{i}])$ *is a probabilistic polynomial time algorithm that takes as input the secret key and an element of the encrypted array and decrypts it to output the corresponding element of the input array.*

Security. For security, we require that the encrypted shuffle satisfy obliviousness. We now state a simulation-based security notion for an oblivious shuffle protocol.

Definition 6 (Obliviousness). *Let* $\mathsf{OShuffle} = (\mathsf{Setup}, \mathsf{Shuffle}_{\mathbf{C},\mathbf{S}}, \mathsf{Dec})$ *be an encrypted shuffle and consider the following probabilistic experiments where $\mathcal{C}$ is a stateful challenger, $\mathcal{A}$ is a stateful adversary, and $\mathcal{S}$ is a stateful simulator, and $z \in \{0,1\}^*$:*

$\mathbf{Real}_{\mathsf{OShuffle},\mathcal{C},\mathcal{A}}(k)$: *given* z*, the adversary* $\mathcal{A}$ *outputs an array* A *and receives* EA *from the challenger, where* $(K, \mathsf{EA}) \leftarrow \mathsf{Setup}(1^k, \mathsf{A})$*.* $\mathcal{A}$ *then chooses a permutation* π *over the domain* $[n]$ *where* $n = \#\mathsf{A}$*. The challenger and the adversary execute* $\mathsf{Shuffle}_{\mathbf{C},\mathbf{S}}(K, \pi; \mathsf{EA})$*. Finally,* $\mathcal{A}$ *outputs a bit* b *that is output by the experiment.*

$\mathbf{Ideal}_{\mathsf{OShuffle},\mathcal{A},\mathcal{S}}(k)$: *given* z*, the adversary* $\mathcal{A}$ *outputs an array* A*. Given* $n = \#\mathsf{A}$*, the simulator returns an encrypted structure* EA *to* $\mathcal{A}$*.* $\mathcal{A}$ *then chooses a permutation* π *over the domain* $[n]$*. The simulator and the adversary execute* $\mathsf{Shuffle}_{\mathcal{S},\mathcal{A}}(\perp; \mathsf{EA})$*; Finally,* $\mathcal{A}$ *outputs a bit* b *that is output by the experiment.*

We say that $\mathsf{OShuffle}$ *is oblivious if there exists a* PPT *simulator* $\mathcal{S}$ *such that for all* PPT *adversaries* $\mathcal{A}$*, for all* $n \geq 1$*, and all* $z \in \{0,1\}^*$*,*

$$|\Pr[\mathbf{Real}_{\mathsf{OShuffle},\mathcal{C},\mathcal{A}}(k) = 1] - \Pr[\mathbf{Ideal}_{\mathsf{OShuffle},\mathcal{A},\mathcal{S}}(k) = 1]| \leq \mathsf{negl}(k).$$

Correctness. For correctness, we require that the encrypted array EA' output by the $\mathsf{Shuffle}_{\mathbf{C},\mathbf{S}}$ protocol is such that $\mathsf{EA}' \equiv \mathsf{A}'$ where $\mathsf{A}'[\pi(i)] = \mathsf{A}[i]$.

5 An Oblivious Shuffle-Based Rebuild Compiler

In this section, we describe a rebuild compiler SRC that uses an oblivious shuffle as a building block. The compiler converts any semi-dynamic type-$\mathbf{T}$ structured encryption scheme into a static rebuildable type-$\mathbf{T}$ structured encryption scheme. We provide the pseudocode for SRC in Figure 4.

Overview. The SRC takes as input a semi-dynamic STE scheme σ_{DS} and outputs a static rebuildable STE scheme Σ_{DS}. SRC works by creating an additional encrypted array RAM containing all the update operations required to instantiate the data structure DS. The server stores this array alongside the encrypted data structure EDS. When the client wants to rebuild EDS, it executes the oblivious shuffle protocol over the entries of RAM to generate a shuffled array RAM'. Finally, the client uses the updates in RAM' to create a new EDS' that also instantiates DS.

Detailed description. During Setup, the client initializes an empty array RAM and a query counter gcnt. It then generates the sequence of updates $u_1, \ldots, u_m$ used to instantiate DS using $\mathsf{Log}(\mathsf{DS})$, encrypts each update u_i and stores it in $\mathsf{RAM}[i]$. Next, the client runs the setup of the oblivious shuffle using RAM. Finally, the client runs the setup of σ_{DS} to generate the structure EDS_0. The server stores both EDS_0 and RAM as the encrypted data structure EDS.

During query, if $\mathsf{gcnt} = n$, where n is the epoch length, the client aborts. Otherwise the client increments the query counter and runs the query protocol $\mathsf{Query}_{\mathbf{C},\mathbf{S}}$ of the scheme σ_{DS}.

During rebuild, the client samples a random permutation π over the space $[m]$ where $m = \#\mathsf{DS}$. The client and the server execute the $\mathsf{Shuffle}_{\mathbf{C},\mathbf{S}}$ protocol to shuffle RAM according to π and generate RAM'. Then the client sets up an

Let $\sigma_{\mathsf{DS}} = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Add})$ be a semi-dynamic type-$\mathbf{T}$ structured encryption scheme, let $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric-key encryption scheme, and let $\mathsf{OShuffle} = (\mathsf{Setup}, \mathsf{Shuffle}, \mathsf{Dec})$ be an oblivious shuffle. Consider the static rebuildable type-$\mathbf{T}$ structured encryption scheme $\Sigma_{\mathsf{DS}} = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Rebuild})$ defined as follows:

- $\mathsf{Setup}(1^k, n, \mathsf{DS})$
  1. initialize an empty array RAM and a counter $\mathsf{gcnt} := 0$;
  2. sample an encryption key $K_{\mathsf{RAM}} \leftarrow \mathsf{SKE.Gen}(1^k)$;
  3. generate the sequence $(u_1, u_2, \cdots, u_m) \leftarrow \mathsf{Log}(\mathsf{DS})$, where $m = \#\mathsf{DS}$.
  4. for $i \in m$, set $\mathsf{RAM}[i] = \mathsf{SKE.Enc}(K_{\mathsf{RAM}}, u_i)$;
  5. compute $(K_{\mathsf{Shuffle}}, \mathsf{RAM}) \leftarrow \mathsf{OShuffle.Setup}(1^k, \mathsf{RAM})$;
  6. compute $(K_{\mathsf{DS}}, st_{\mathsf{DS}}, \mathsf{EDS}_0) \leftarrow \sigma_{\mathsf{DS}}.\mathsf{Setup}(1^k, \mathsf{DS})$;
  7. set $K = (K_{\mathsf{RAM}}, K_{\mathsf{Shuffle}}, K_{\mathsf{DS}})$, set $st := (st_{\mathsf{DS}}, \mathsf{gcnt})$ and set $\mathsf{EDS} := (\mathsf{EDS}_0, \mathsf{RAM})$
  8. output (K, st, EDS);
- $\mathsf{Query}_{\mathbf{C},\mathbf{S}}(K, st, q; \mathsf{EDS})$
  1. $\mathbf{C}$ parses K as $(K_{\mathsf{RAM}}, K_{\mathsf{Shuffle}}, K_{\mathsf{DS}})$ and st as $(st_{\mathsf{DS}}, \mathsf{gcnt})$;
  2. if $\mathsf{gcnt} = n$, abort, else set $\mathsf{gcnt} := \mathsf{gcnt} + 1$;
  3. $\mathbf{C}$ and $\mathbf{S}$ execute $(r, st_{\mathsf{DS}}; \perp) \leftarrow \sigma_{\mathsf{DS}}.\mathsf{Query}_{\mathbf{C},\mathbf{S}}(K_{\mathsf{DS}}, st_{\mathsf{DS}}, q; \mathsf{EDS})$;
  4. $\mathbf{C}$ sets $st := (st_{\mathsf{DS}}, \mathsf{gcnt})$
  5. $\mathbf{C}$ outputs (r, st) and $\mathbf{S}$ outputs $\perp$;
- $\mathsf{Rebuild}_{\mathbf{C},\mathbf{S}}(K, st; \mathsf{EDS})$
  1. $\mathbf{C}$ parses K as $(K_{\mathsf{RAM}}, K_{\mathsf{Shuffle}}, K_{\mathsf{DS}})$ and st as $(st_{\mathsf{DS}}, \mathsf{gcnt})$;
  2. $\mathbf{S}$ parses EDS as $(\mathsf{EDS}_0, \mathsf{RAM})$;
  3. $\mathbf{C}$ samples a permutation π over $[m]$ uniformly at random;
  4. $\mathbf{C}$ and $\mathbf{S}$ execute $(K'_{\mathsf{Shuffle}}; \mathsf{RAM}') \leftarrow \mathsf{OShuffle.Shuffle}_{\mathbf{C},\mathbf{S}}(K_{\mathsf{Shuffle}}, \pi; \mathsf{RAM})$;
  5. $\mathbf{C}$ and $\mathbf{S}$ execute $(K'_{\mathsf{DS}}, st'_{\mathsf{DS}}; \mathsf{EDS}') \leftarrow \sigma_{\mathsf{DS}}.\mathsf{Setup}(1^k, \perp)$;
  6. for $i \in [m]$,
     (a) $\mathbf{S}$ sends $\mathsf{RAM}'[i]$ to $\mathbf{C}$;
     (b) $\mathbf{C}$ executes $c_i \leftarrow \mathsf{OShuffle.Dec}(K_{\mathsf{Shuffle}}, \mathsf{RAM}'[i])$;
     (c) $\mathbf{C}$ executes $u_i \leftarrow \mathsf{SKE.Dec}(K_{\mathsf{RAM}}, c_i)$;
     (d) $\mathbf{C}$ and $\mathbf{S}$ execute $(st'_{\mathsf{DS}}; \mathsf{EDS}') \leftarrow \sigma_{\mathsf{DS}}.\mathsf{Add}_{\mathbf{C},\mathbf{S}}(K'_{\mathsf{DS}}, st'_{\mathsf{DS}}, u_i; \mathsf{EDS}')$;
  7. $\mathbf{C}$ sets $st_{\mathsf{DS}} := st'_{\mathsf{DS}}$, $\mathsf{gcnt} := 0$, $K_{\mathsf{DS}} := K'_{\mathsf{DS}}$ and $K_{\mathsf{Shuffle}} := K'_{\mathsf{Shuffle}}$;
  8. $\mathbf{S}$ sets $\mathsf{EDS} = (\mathsf{EDS}', \mathsf{RAM}')$ and removes EDS_0 and RAM.

Fig. 4. The shuffle-based rebuild compiler SRC.

empty encrypted structure EDS' on the server. It downloads each entry of RAM', decrypts it, and performs the corresponding update on EDS' using the $\mathsf{Add}_{\mathbf{C},\mathbf{S}}$ protocol of σ_{DS}. After the client performs all the updates, it reinitializes its local state and the server stores $\mathsf{EDS} = (\mathsf{EDS}', \mathsf{RAM}')$

Security. We now show that if the input structured encryption scheme leaks only the total size of the structure and the operation equality pattern on every operation, then the scheme output by our compiler is a static STE scheme with a leakage-free rebuild. We state the security theorem and defer its proof to the full version of our paper.

Theorem 1. *If σ_{DS} is an $\mathcal{L}$-secure semi-dynamic STE scheme with leakage profile $\mathcal{L}_\sigma = (\mathsf{patt}_{\mathsf{S}}, \mathsf{patt}_{\mathsf{O}}) = (\mathsf{dsize}, \mathsf{oeq})$, $\mathsf{OShuffle} = (\mathsf{Setup}, \mathsf{Shuffle}, \mathsf{Dec})$ is an oblivious shuffle, and $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is a CPA-secure encryption scheme, then the scheme Σ_{DX} output by* SRC *is an $\mathcal{L}_\Sigma$-secure static rebuildable STE scheme with leakage profile $\mathcal{L}_\Sigma = (\mathsf{patt}_{\mathsf{S}}, \mathsf{patt}_{\mathsf{Q}}, \mathsf{patt}_{\mathsf{R}}) = (\mathsf{dsize}, \mathsf{qeq}, \perp)$.*

Efficiency. SRC introduces an additional server storage of $O(\#\mathsf{DS})$ to store the encrypted RAM. It also introduces communication and round complexity depending on the underlying oblivious shuffle protocol. If we instantiate $\mathsf{OShuffle}$ using $\mathsf{CacheShuffle}$ and the client has $\mathsf{state} = \omega(\log(\#\mathsf{DS}))$, then the total communication complexity is $O(\#\mathsf{DS} \cdot \log_{\mathsf{state}} \#\mathsf{DS})$, and therefore the worst case round complexity must also be $O(\#\mathsf{DS} \cdot \log_{\mathsf{state}} \#\mathsf{DS})$.

6 ERS: A Distribution-Aware Query Equality Suppressor

We now describe our distribution-aware query equality suppressor ERS. The pseudocode for our suppressor is given in Figures 5 and 6.

Overview. Our suppressor ERS takes as input a static rebuildable encrypted dictionary scheme ω_{DX} and outputs a static encrypted dictionary Ω_{DX} with no query equality leakage. The suppressor also takes as input a query distribution $\mathcal{Q}$ over the label space of the input dictionary DX, a query epoch length n, counter functions $\mathsf{CF}_1, \mathsf{CF}_2, \ldots, \mathsf{CF}_\lambda$ with ranges $\rho_1, \rho_2, \ldots, \rho_\lambda$ respectively, a counter choice function $\mathsf{chooseCF}$, a sketching scheme Δ, and an upper limit M on the size of the client-side replica cache. We now describe the setup, query, and rebuild of our suppressor in detail.

Setup. During setup, the client sets up all the replicas and initializes its state.

- *(setup replicas)* The client creates an empty dictionary $\overline{\mathsf{DX}}$ to store replicas and sets its time counter gcnt to 0. For each label ℓ, the client uses the query probability p_ℓ and the counter choice function to select a counter function CF_j. It then creates ρ_j replicas of label ℓ with counter values $1, 2, \ldots, \rho_j$. Each replicated label is assigned the value $v := \mathsf{DX}[\ell]$. After creating all the label replicas, the client creates n dummy replicas where n is the length of the query epoch. All the replicas are added to $\overline{\mathsf{DX}}$ and encrypted using the input static dictionary scheme ω_{DX}.

– *(initialize state)* Finally, the client initializes its own state by computing sketch of $\mathcal{Q}$, and initializing a dictionary $\mathsf{DX}_{\mathsf{cache}}$ and a min-priority queue $\mathsf{Q}_{\mathsf{cache}}$ for the replica cache. The client's state consists of the time counter, the sketch, the cache dictionary and queue, and any state output by the input encrypted dictionary scheme ω_{DX};

Query. During query, the client queries a replica, updates the cache, and clears expired entries from the replica cache.

– *(query replica)* If the time counter is larger than the epoch length, the client rebuilds the encrypted dictionary and continues. The client then increments the time counter gcnt and checks the cache dictionary for the label ℓ.
If the cache contains a replica for the label (or) the cache has M entries, the client queries a dummy replica with the current value of gcnt.
If the cache does not contain a replica and the cache is smaller than M, the client does the following to compute the correct replica: (1) use sketch to retrieve the query probability $\tilde{p}_\ell$, (2) evaluate the counter choice function on $\tilde{p}_\ell$ to get the index $\tilde{j}$ and the counter function $\mathsf{CF}_{\tilde{j}}$.
Finally, it evaluates $\mathsf{CF}_{\tilde{j}}(\mathsf{gcnt})$ to compute the counter idx and queries for $\ell \,\|\, \mathsf{idx}$.
– *(update cache)* When the client receives a value from the encrypted dictionary, it computes the step length $\lceil n/\rho_{\tilde{j}} \rceil$ for the counter function $\mathsf{CF}_{\tilde{j}}$ and sets the expiration time for the value to be $t = \mathsf{gcnt} + \lceil n/\rho_{\tilde{j}} \rceil - 1$. It adds the label-value pair (ℓ, r) to $\mathsf{DX}_{\mathsf{cache}}$, and adds the label ℓ to the min queue $\mathsf{Q}_{\mathsf{cache}}$ with priority t.
– *(clear cache)* The client then checks the minimum priority element of $\mathsf{Q}_{\mathsf{cache}}$. If the element expired during or before the current time, the client removes it from both $\mathsf{Q}_{\mathsf{cache}}$ and $\mathsf{DX}_{\mathsf{cache}}$.

Rebuild. After every n queries, the client executes the rebuild protocol. First, it executes the rebuild protocol of the input dictionary scheme. The client then re-initializes the time counter, the replica cache, and the state of the encrypted dictionary. The sketch of the input distribution remains unchanged.

6.1 Security

In this section we prove the security of ERS. ERS takes as input a static encrypted rebuildable dictionary scheme ω_{DX} with the leakage profile $\mathcal{L}_\omega = (\mathsf{dsize}, \mathsf{qeq}, \perp)$ and outputs a static rebuildable dictionary scheme Ω_{DX} with the query equality pattern fully suppressed.

Intuitively, the security proof goes through because ERS ensures that the client always queries a fresh replica. As long as the input dictionary scheme leaks only the query equality, the output scheme will have no query leakage since all the queries are unique. More formally, we state and sketch the proof of the following security theorem for Ω_{DX}.

Let $\mathcal{Q}$ be a probability distribution over the space $\mathbb{Q}_{\mathsf{DX}}$, epoch length $n \in \mathbb{N}_{\geq 1}$, $\mathsf{CF}_1, \ldots, \mathsf{CF}_\lambda$ be counter functions where $\mathsf{CF}_j : [n] \to [\rho_j]$ for all $j \in [\lambda]$, $\lambda \in [\#\mathbb{Q}_{\mathsf{DX}}]$, $\mathsf{chooseCF}$ be a counter choice function $\mathsf{chooseCF} : [0,1] \to [\lambda]$, $\Delta = (\mathsf{SketchDist}, \mathsf{QueryDist})$ be a sketching scheme, and cache size limit $M \in \mathbb{N}_{\geq 1}$. Let $\omega_{\mathsf{DX}} = (\mathsf{Setup}, \mathsf{Get}, \mathsf{Rebuild})$ be a static rebuildable dictionary encryption scheme and consider the scheme $\Omega_{\mathsf{DX}} = (\mathsf{Setup}, \mathsf{Get}, \mathsf{Rebuild})$ that works as follows:

- $\mathsf{Setup}(1^k, n, \mathcal{Q}, \mathsf{DX})$
  1. parse $\mathcal{Q}$ as $(p_\ell)_{\ell \in \mathbb{Q}_{\mathsf{DX}}}$;
  2. initialize empty dictionary $\overline{\mathsf{DX}}$ and a counter $\mathsf{gcnt} := 0$;
  3. for each label ℓ in $\mathbb{Q}_{\mathsf{DX}}$:
     (a) compute $j := \mathsf{chooseCF}(p_\ell)$;
     (b) let $v := \mathsf{DX}[\ell]$;
     (c) for $\mathsf{idx} \in [\rho_j]$, set $\overline{\mathsf{DX}}[\ell \,\|\, \mathsf{idx}] := v$;
  4. for $i := 1, 2, \ldots n$, set $\overline{\mathsf{DX}}[\perp \,\|\, i] := \perp$;
  5. compute $\mathsf{sketch} \leftarrow \Delta.\mathsf{SketchDist}(\mathcal{Q})$;
  6. compute $(K, st_{\mathsf{dx}}, \overline{\mathsf{EDX}}) \leftarrow \omega_{\mathsf{DX}}.\mathsf{Setup}(1^k, n, \overline{\mathsf{DX}})$;
  7. initialize empty dictionary $\mathsf{DX}_{\mathsf{cache}}$;
  8. initialize empty min-priority queue $\mathsf{Q}_{\mathsf{cache}}$;
  9. set $st := (\mathsf{gcnt}, \mathsf{sketch}, \mathsf{DX}_{\mathsf{cache}}, \mathsf{Q}_{\mathsf{cache}}, st_{\mathsf{dx}})$;
  10. set $\mathsf{EDX} := \overline{\mathsf{EDX}}$;
  11. output (K, st, EDX).
- $\mathsf{Get}_{\mathbf{C},\mathbf{S}}(K, st, \ell; \mathsf{EDX})$
  1. $\mathbf{C}$ parses st as $(\mathsf{gcnt}, \mathsf{sketch}, \mathsf{DX}_{\mathsf{cache}}, \mathsf{Q}_{\mathsf{cache}}, st_{\mathsf{dx}})$;
  2. if $\mathsf{gcnt} > n$,
     (a) $\mathbf{C}$ and $\mathbf{S}$ execute $(K', st'; \mathsf{EDX}') \leftarrow \Omega_{\mathsf{DX}}.\mathsf{Rebuild}(K, st; \mathsf{EDX})$;
     (b) $\mathbf{C}$ sets $K := K'$ and $st := st'$;
     (c) $\mathbf{C}$ parses st' as $(\mathsf{gcnt}, \mathsf{sketch}, \mathsf{DX}_{\mathsf{cache}}, \mathsf{Q}_{\mathsf{cache}}, st_{\mathsf{dx}})$;
     (d) $\mathbf{S}$ sets $\mathsf{EDX} := \mathsf{EDX}'$;
     (e) continue;
  3. $\mathbf{C}$ sets $\mathsf{gcnt} := \mathsf{gcnt} + 1$;
  4. $\mathbf{C}$ sets $r_{\mathsf{cache}} := \mathsf{DX}_{\mathsf{cache}}[\ell]$;
  5. if $r_{\mathsf{cache}} \neq \perp$,
     (a) $\mathbf{C}$ sets $r := r_{\mathsf{cache}}$;
     (b) $\mathbf{C}$ sets $\ell' := \perp \,\|\, \mathsf{gcnt}$;
     else if $r_{\mathsf{cache}} = \perp$ and $\#\mathsf{DX}_{\mathsf{cache}} \geq M$;
     (a) $\mathbf{C}$ sets $\ell' := \perp \,\|\, \mathsf{gcnt}$;
     else,
     (a) $\mathbf{C}$ computes $\tilde{p}_\ell \leftarrow \Delta.\mathsf{QueryDist}(\mathsf{sketch}, \ell)$;
     (b) $\mathbf{C}$ computes $\tilde{j} := \mathsf{chooseCF}(\tilde{p}_\ell)$;
     (c) $\mathbf{C}$ sets $\ell' := \ell \,\|\, \mathsf{idx}$ where $\mathsf{idx} := \mathsf{CF}_{\tilde{j}}(\mathsf{gcnt})$;

Fig. 5. A storage-efficient replication-based query equality suppressor ERS.

– $\mathsf{Get}_{\mathbf{C},\mathbf{S}}(K, st, \ell; \mathsf{EDX})$
  6. $\mathbf{C}$ and $\mathbf{S}$ execute $(r_{\mathsf{dx}}, st_{\mathsf{dx}}; \perp) \leftarrow \omega_{\mathsf{DX}}.\mathsf{Get}_{\mathbf{C},\mathbf{S}}(K, st_{\mathsf{dx}}, \ell'; \mathsf{EDX})$;
  7. if $r_{\mathsf{dx}} \neq \perp$,
     (a) $\mathbf{C}$ sets $r := r_{\mathsf{dx}}$;
     (b) $\mathbf{C}$ computes $\tilde{j} := \mathsf{chooseCF}(\tilde{p}_\ell)$;
     (c) $\mathbf{C}$ computes $t = \mathsf{gcnt} + \lceil \frac{n}{\rho_{\tilde{j}}} \rceil - 1$;
     (d) $\mathbf{C}$ sets $\mathsf{DX}_{\mathsf{cache}}[\ell] := r$;
     (e) $\mathbf{C}$ runs $\mathsf{Q}_{\mathsf{cache}}.\mathsf{Add}(\ell, t)$;
  8. $\mathbf{C}$ runs $(\ell', t') \leftarrow \mathsf{Q}_{\mathsf{cache}}.\mathsf{FindMin}()$
  9. while $t' \leq \mathsf{gcnt}$:
     (a) $\mathbf{C}$ runs $\mathsf{Q}_{\mathsf{cache}}.\mathsf{DelMin}()$;
     (b) $\mathbf{C}$ sets $\mathsf{DX}_{\mathsf{cache}}[\ell'] := \perp$;
     (c) $\mathbf{C}$ runs $(\ell', t') \leftarrow \mathsf{Q}_{\mathsf{cache}}.\mathsf{FindMin}()$;
  10. $\mathbf{C}$ sets $st := (\mathsf{gcnt}, \mathsf{sketch}, \mathsf{DX}_{\mathsf{cache}}, \mathsf{Q}_{\mathsf{cache}}, st_{\mathsf{dx}})$;
  11. $\mathbf{C}$ outputs (r, st) and $\mathbf{S}$ outputs $\perp$.
– $\mathsf{Rebuild}_{\mathbf{C},\mathbf{S}}(K, st; \mathsf{EDX})$
  1. $\mathbf{C}$ parses st as $(\mathsf{gcnt}, \mathsf{sketch}, \mathsf{DX}_{\mathsf{cache}}, \mathsf{Q}_{\mathsf{cache}}, st_{\mathsf{dx}})$;
  2. $\mathbf{C}$ and $\mathbf{S}$ execute $(K', st'_{\mathsf{dx}}; \mathsf{EDX}') \leftarrow \omega_{\mathsf{DX}}.\mathsf{Rebuild}_{\mathbf{C},\mathbf{S}}(K, st_{\mathsf{dx}}; \mathsf{EDX})$;
  3. $\mathbf{C}$ sets $\mathsf{gcnt} := 0$;
  4. $\mathbf{C}$ sets $\mathsf{DX}_{\mathsf{cache}}$ to an empty dictionary;
  5. $\mathbf{C}$ sets $\mathsf{Q}_{\mathsf{cache}}$ to an empty min-priority queue;
  6. $\mathbf{C}$ sets $st' := (\mathsf{gcnt}, \mathsf{sketch}, \mathsf{DX}_{\mathsf{cache}}, \mathsf{Q}_{\mathsf{cache}}, st'_{\mathsf{dx}})$
  7. $\mathbf{C}$ outputs (K', st') and $\mathbf{S}$ outputs EDX'.

Fig. 6. A storage-efficient replication-based query equality suppressor ERS (continued).

Theorem 2 (Security of Ω_{DX}). *If ω_{DX} is a static rebuildable $\mathcal{L}$-secure encrypted dictionary scheme with $\mathcal{L}_\omega = (\mathsf{patt}_{\mathsf{S}}, \mathsf{patt}_{\mathsf{Q}}, \mathsf{patt}_{\mathsf{R}}) = (\mathsf{dsize}, \mathsf{qeq}, \perp)$, then the scheme Ω_{DX} output by* ERS *is $\mathcal{L}_\Omega$-secure where $\mathcal{L}_\Omega = (\mathsf{patt}_{\mathsf{S}}, \mathsf{patt}_{\mathsf{Q}}, \mathsf{patt}_{\mathsf{R}}) = ((\mathsf{dsize}, \mathcal{Q}), \perp, \perp)$.*

Proof Sketch. We begin by noticing that ERS transforms queries such that every query in an epoch is unique. Now, let $\mathcal{S}_\omega$ be the simulator that exists by the adaptive security of ω_{EDS}. We construct a simulator $\mathcal{S}_\Omega$ for the scheme Ω_{DS} as follows:

– Simulating Setup: Given $\#\mathsf{DX} = \mathsf{dsize}(\mathsf{DX})$ and $\mathcal{Q}$, compute $\rho_1, \ldots, \rho_{\#\mathsf{DX}}$. Let $\rho := \Sigma_{i=1}^{\#\mathsf{DX}} \rho_i$ and run $\mathsf{EDX} \leftarrow \mathcal{S}_\omega(\rho + n)$. Output EDX to the adversary. Set $\mathsf{gcnt} := 0$.
– Simulating Query[3]: If $\mathsf{gcnt} > n$, simulate $\mathsf{Rebuild}$, else set $\mathsf{gcnt} = \mathsf{gcnt} + 1$. Given $\perp$, for any sequence of queries $q_1, \ldots, q_n$, use $\mathcal{S}_\omega(I_n)$ to simulate the adversary's view, where I_n is the $n \times n$ identity matrix.

[3] Note that the client (and therefore the adversary) always queries labels that exist in the support of $\mathcal{Q}$. Alternatively, both the input and output schemes leak rlen, where rlen reveals the presence/absence bit of a queried label.

– Simulating $\mathsf{Rebuild}$: Given $\perp$, run $\mathcal{S}_\omega(\perp)$ to simulate the adversary's view of rebuilding EDS. Set $\mathsf{gcnt} := 0$.

We now show that the $\mathbf{Real}_{\Omega,\mathcal{C},\mathcal{A}}$ experiment is computationally indistinguishable from the $\mathbf{Ideal}_{\Omega,\mathcal{A},\mathcal{S}}$ experiment using the following games.

– Game_0: is the same as the $\mathbf{Real}_{\Omega,\mathcal{C},\mathcal{A}}$ experiment.
– Game_1: is the same as Game_0 except the setup of ω_{DS} for DS is replaced by the simulated setup $\mathcal{S}_\omega(\rho + n)$; every query sequence $q_1, \ldots, q_n$ to ω_{DS} is replaced by the simulated query sequence $\mathcal{S}_\omega(I_n)$; and all rebuilds of ω_{DS} are replaced by simulated executions $\mathcal{S}_\omega(\perp)$.

For Game_1, notice that: (1) the simulator $\mathcal{S}_\Omega$ computes $(\rho+n)$, which is the setup leakage $\mathsf{dsize}(\mathsf{DX})$ for $\mathcal{S}_\omega$, (2) the query equality leakage $\mathsf{qeq}(q_1, \ldots, q_n) = I_n$ when only unique query operations are possible, and (3) the rebuild protocol is leakage-free and can be simulated with no leakage. Then Game_1 must be indistinguishable from Game_0 by the $(\mathsf{dsize}, \mathsf{qeq}, \perp)$-security of ω_{DS}. Finally, Game_1 is the same as the $\mathbf{Ideal}_{\Omega,\mathcal{A},\mathcal{S}}$ experiment and our proof is complete.

■

6.2 Efficiency and Correctness

We now discuss the storage and communication complexity of Ω_{DX}. We assume that the rebuild protocol of the input dictionary scheme is instantiated using the shuffle-based compiler SRC.

Server storage. For each label in the dictionary ℓ_i the number of replicas is equal to the number of steps in the counter function CF_i. Since $\mathsf{CF}_i : [n] \to [\rho_i]$, ℓ_i has at most ρ_i replicas. Additionally, ERS adds n dummy replicas, and SRC adds a RAM which has the same size as the replicated dictionary.
The server storage is then:

$$\mathsf{storage}_\Omega(\mathsf{DX}) = O\Big(\sum_{i=1}^{m} \rho_i + n\Big). \tag{1}$$

Client state. The client state is used to maintain the current time counter, the sketch of the query distribution, the replica cache, and the state for the underlying encrypted dictionary scheme. The client state is also used during the shuffle-based rebuild. The total client state is then:

$$\mathsf{state}_\Omega(\mathsf{DX}) = \Big(O(1) + \mathsf{state}_\Delta(\mathcal{Q}) + \mathsf{state}_{\mathsf{cache}}(\mathcal{Q}) + \mathsf{state}_\omega(\overline{\mathsf{DX}}) + \mathsf{state}_{\mathsf{Shuffle}}(\overline{\mathsf{DX}})\Big). \tag{2}$$

Online communication complexity. The online communication complexity of Ω is the same as that of the underlying encrypted dictionary scheme ω. If ocomm_Ω is the online communication complexity of Ω, then for any query q:

$$\mathsf{ocomm}_\Omega(q) = \mathsf{ocomm}_\omega(q). \tag{3}$$

Amortized communication complexity. When we compute the amortized communication complexity for one epoch, we account for the complexity of rebuilding the replicated dictionary $\overline{\mathsf{DX}}$. The total size of $\overline{\mathsf{DX}}$ depends on the counter functions chosen for the query distribution $\mathcal{Q}$. Then the total communication complexity for one epoch is:

$$\mathsf{comm}_\Omega(q_1, \ldots, q_n) = \sum_{i=1}^{n} \mathsf{ocomm}_\omega(q_i) + \mathsf{comm}_{\mathsf{Shuffle}}(\overline{\mathsf{DX}}). \tag{4}$$

Finally, we divide by n to get the amortized communication complexity for each query:

$$\mathsf{comm}_\Omega(q) = \frac{1}{n} \cdot \Big(\sum_{i=1}^{n} \mathsf{ocomm}_\omega(q_i) + \mathsf{comm}_{\mathsf{Shuffle}}(\overline{\mathsf{DX}}) \Big). \tag{5}$$

Amortized round complexity. The total number of communication rounds for one epoch is the totarounds for n queries using the input encrypted dictionary scheme and the rounds for the shuffle-based rebuild of the replicated dictionary. Then we have the amortized round complexity for one query:

$$\mathsf{rounds}_\Omega(q) = \frac{1}{n} \cdot \Big(\sum_{i=1}^{n} \mathsf{rounds}_\omega(q_i) + \mathsf{rounds}_{\mathsf{Shuffle}}(\overline{\mathsf{DX}}) \Big). \tag{6}$$

Correctness. Ω always returns the correct response to a queried label, except in the case where the replica cache has reached its maximum size M and the queried label is not found in the cache. We bound the probability of this event by upper bounding the size of the replica cache at every time step. More precisely, we prove the following theorem that upper bounds the size of the replica cache for a general query distribution $\mathcal{Q}$. Given this theorem, we can show that Ω_{DX} is correct with high probability if we can show that the cache size does not exceed M with high probability for the query distribution $\mathcal{Q}$ with counter functions $\mathsf{CF}_1, \ldots, \mathsf{CF}_m$. We defer the proof to the full version of our paper.

Theorem 3 (Cache size bound). *Let $p_1, p_2, \ldots, p_m$ be the p.m.f of a discrete query distribution $\mathcal{Q}$ and let $T_1, \ldots, T_m$ be the step lengths of the counter functions $\mathsf{CF}_1, \ldots, \mathsf{CF}_m$ respectively. For any epoch length $n \geq 1$, if $T_i \in [n-1]$ and $T_i \cdot p_i \ll 1$, then for any $\delta > 0$:*

$$\Pr\left[Y(t) \geq (1+\delta) \cdot \sum_{i=1}^{m} T_i \cdot p_i \right] \leq \exp\left(\frac{-\delta^2}{2 \cdot (1+\delta)} \cdot \sum_{i=1}^{m} T_i \cdot p_i \right),$$

where $Y(t)$ is a random variable representing the size of the cache at time $1 \leq t \leq n$.

When analyzing the concrete efficiency of ERS we found it useful to narrow the scope to a specific family of distributions. We chose the Zipf distribution because of its applicability in practice. In the next section, we discuss the efficiency and correctness of ERS when the query distribution $\mathcal{Q}$ is a Zipf distribution.

7 ERS: The Zipf Case

A discrete random variable X is Zipf distributed with parameter $s \in \mathbb{R}_{\geq 0}$, if for all $j \in \{1, \ldots, m\}$,

$$\Pr[\, X = j \,] = \frac{j^{-s}}{H_{m,s}},$$

where $H_{m,s} = \Sigma_{i=1}^{m} 1/j^s$ is the general form of the harmonic number. We also assume the existence of a permutation $\pi : \mathbb{Q}_{\mathsf{DX}} \to [m]$ that maps every query in the query space $\mathbb{Q}_{\mathsf{DX}}$ to a particular rank in $[m]$. We denote by $\mathcal{Z}_{m,s}$ the Zipf distribution over a query space of size m with parameter s.

Sketching scheme. Notice that under the Zipf distribution, each label in $\mathbb{Q}_{\mathsf{DX}}$ maps to a unique probability and that the bulk of the mass is located within the first few ranks. We leverage this property to build a simple scheme $\Delta_{\mathsf{sk}}^{\mathsf{zipf}} = (\mathsf{SketchDist}, \mathsf{QueryDist})$ that only keeps track of the top-κ queries, for $\kappa \in [m]$. The $\mathsf{SketchDist}$ algorithm takes as input the Zipf distribution $\mathcal{Z}_{m,s}$ and outputs

$$\mathsf{sketch} := \Big(\pi^{-1}(1), \cdots, \pi^{-1}(\kappa)\Big).$$

The $\mathsf{QueryDist}$ algorithm takes as input the label ℓ and outputs either the correct probability if the label is one of the top-κ labels or 0 otherwise. More precisely, we have

$$\mathsf{QueryDist}(\ell) = \begin{cases} \big(\pi(\ell)^s \cdot H_{m,s}\big)^{-1} & \text{if } \ell \in \mathsf{sketch} \\ 0 & \text{otherwise.} \end{cases}$$

Note that the total size of the sketch is $O(\kappa)$.

Counter functions. We instantiate $\kappa+1$ counter functions $\mathsf{CF}_1, \ldots, \mathsf{CF}_{\kappa+1}$. Each counter function CF_i has a replication factor ρ_i, chosen as a function of the Zipf distribution. More formally, for all $n \in \mathbb{N}_{\geq 1}$, $\kappa \in \mathbb{N}_{\geq 1}$, $\rho \in [n]$, we define $\mathsf{CF}_i : [n] \to [\rho_i]$ such that

$$\mathsf{CF}_i(x) = \left\lceil x \cdot \left\lceil \frac{n}{\rho_i} \right\rceil^{-1} \right\rceil$$

where the replication factor ρ_i of the ith counter function for $i \in [\kappa]$ is

$$\rho_i := \left\lceil \rho \cdot \frac{i^{-s}}{H_{m,s}} \right\rceil,$$

and for $i = \kappa + 1$, $\rho_{\kappa+1} = 1$. We also set the parameter

$$\rho = m \cdot H_{m,s}.$$

As a result, CF_1 has the highest replication factor while $\mathsf{CF}_{\kappa+1}$ has the lowest.

Choice function. For our choice function, we want to map labels with high probability to counter functions with high replication factors. Our choice function maps the top-κ labels to the largest κ replication factors, and all other labels to a replication factor of 1. More formally, the choice function $\mathsf{chooseCF} : [0,1] \to [\kappa+1]$ is defined as

$$\mathsf{chooseCF}(p) = \begin{cases} \left\lceil \exp\left(s^{-1} \cdot \log\left((p \cdot H_{m,s})^{-1}\right)\right) \right\rceil & \text{if } p \geq \left(\kappa^s \cdot H_{m,s}\right)^{-1} \\ \kappa + 1 & \text{otherwise.} \end{cases}$$

Epoch length. Finally, we set our epoch length $n = m$, where $m = \#\mathsf{DX}$.

7.1 Efficiency

Concretely, we instantiate Ω_{DX} for Zipf distributions as follows:

- We start with the semi-dynamic Π^+_{bas} construction of Cash, Jaeger, Jarecki, Jutla, Krawczyk, Rosu and Steiner [17] with leakage profile $\mathcal{L}_{\mathsf{bas}} = (\mathsf{dsize}, \mathsf{oeq})$.
- We then use our shuffle-based compiler SRC using $\mathsf{CacheShuffle}$ to compile Π^+_{bas} to a static rebuildable scheme ω_{bas}. From Theorem 1, ω_{bas} has leakage profile $\mathcal{L}_\omega = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Q}}, \mathcal{L}_{\mathsf{R}}) = (\mathsf{dsize}, \mathsf{qeq}, \bot)$.
- Finally, we suppress the query equality leakage of ω_{bas} using ERS and our instantiations for the Zipf distribution to generate the static rebuildable dictionary encryption scheme Ω_{bas} with leakage profile $\mathcal{L}_\omega = (\mathsf{dsize}, \bot, \bot)$.

Efficiency of Π^+_{bas}. When used as an encrypted dictionary, Π^+_{bas} has $O(1)$ communication and round complexity and $O(\#\mathsf{DX})$ server-side storage. Π^+_{bas} is non-interactive and has a constant client state. We refer to the rebuildable scheme output by SRC on input Π^+_{bas} as ω_{bas}.

Efficiency of ω_{bas}. After compilation, the online communication complexity and client state of ω_{bas} is $O(1)$. The online round complexity of ω_{bas} is also $O(1)$, without accounting for the round complexity introduced by the rebuild.

Rebuilding ω_{bas}. From Section 4.3 we know that SRC based on CacheShuffle has communication complexity $O(\#\mathsf{DS} \cdot \log_{\mathsf{state}} \#\mathsf{DS})$ with client state $O(\mathsf{state})$ when $\mathsf{state} = o(\log \#\mathsf{DS})$. The total round complexity of the shuffle is also $O(\#\mathsf{DS} \log_{\mathsf{state}} \#\mathsf{DS})$.

Server storage. From Equation 1, substituting for ρ_i and n, we have:

$$\mathsf{storage}_{\Omega}(\mathsf{DX}) = O\Big(\sum_{i=1}^{\kappa} \left\lceil \rho \cdot \frac{i^{-s}}{H_{m,s}} \right\rceil + \sum_{\kappa+1}^{m} 1 + m\Big) = O(\rho + m).$$

Client state. From Equation 2, knowing that client state for ω_{bas} is $O(1)$ and using a top-κ sketch, we have:

$$\mathsf{state}_{\Omega}(\mathsf{DX}) = \Big(O(\kappa) + \mathsf{state}_{\mathsf{cache}}(\mathcal{Q}) + \mathsf{state}_{\mathsf{Shuffle}}(\overline{\mathsf{DX}})\Big).$$

Then our total client state depends on the size of the replica cache and on the state used by CacheShuffle for $\overline{\mathsf{DX}}$.

Online communication complexity. From Equation 3, knowing that online communication complexity for ω_{bas} is $O(1)$, we have:

$$\mathsf{ocomm}_{\Omega}(q) = O(1).$$

The total client state available impacts the amortized communication and round complexity of Ω_{bas} due to the client state used by CacheShuffle. It also restricts the size of the sketch κ and the size of the replica cache, which affects the correctness of Ω_{bas}.

We can instantiate ERS in two modes, depending on the availability of client state. For each mode, we fix a sketch size and demonstrate that the size of the replica cache (and therefore the total client state) does not exceed this sketch size asymptotically. We also show that while Ω_{bas} has the same server storage and online communication complexity in both modes, each mode leads to different trade-offs in amortized communication complexity, round complexity, and correctness.

7.2 Mode 1: Client state grows as $m^{\frac{1}{s-1}}$

In this mode, we set:

$$\begin{aligned} \text{sketch size } \kappa &= m^{\frac{1}{s-1}}, \text{ with } s > 2 \\ \text{we also know } \#\overline{\mathsf{DX}} &= O(\rho + m), \\ \rho &= m \cdot H_{m,s}, \text{ and} \\ n &= \theta(m). \end{aligned}$$

Since $m^{\frac{1}{s-1}}$ is $\omega(\log(\rho+m))$ for $s > 2$, we can compute the communication complexity of $\mathsf{CacheShuffle}$ as:

$$\begin{aligned}
\mathsf{comm}_{\mathsf{CacheShuffle}}(\overline{\mathsf{DX}}) &= O\left((\rho+m)\cdot\frac{\log(\rho+m)}{\log m^{\frac{1}{s-1}}}\right)\\
&= O\left((m\cdot H_{m,s}+m)\cdot\frac{\log(m\cdot H_{m,s}+m)}{\log m}\right)\\
&= O\left(m\cdot H_{m,s}\cdot\frac{\log(m\cdot H_{m,s})}{\log m}\right)
\end{aligned}$$

Since for $s \geq 1$, $H_{m,s} = O(\log m)$, we have:

$$\begin{aligned}
\mathsf{comm}_{\mathsf{CacheShuffle}}(\overline{\mathsf{DX}}) &= O\Big(m\cdot(\log m+\log\log m)\Big)\\
&= O\Big(m\cdot\log m\Big).
\end{aligned}$$

Amortized communication complexity. From Equation 5, the amortized query complexity of Ω_{bas}:

$$\begin{aligned}
\mathsf{comm}_{\Omega_{\mathsf{bas}}}(q) &= \frac{1}{n}\cdot\left(\sum_{i=1}^{n}O(1)+O\Big(m\cdot\log m\Big)\right)\\
&= O(\log m),\ \text{since } n = O(m).
\end{aligned}$$

Amortized round complexity. From Equation 6, knowing that the round complexity of ω_{bas} is $O(1)$, we have the amortized round complexity of Ω_{bas}:

$$\begin{aligned}
\mathsf{rounds}_{\Omega_{\mathsf{bas}}}(q) &= \frac{1}{n}\cdot\left(\sum_{i=1}^{n}O(1)+\mathsf{rounds}_{\mathsf{Shuffle}}(\overline{\mathsf{DX}})\right),\\
&= O(1)+O(\mathsf{comm}_{\Omega_{\mathsf{bas}}}(q))\\
&= O(\log m).
\end{aligned}$$

Correctness. We state the following corollary of Theorem 3 to bound the cache size of Ω_{bas} with client state $O(m^{\frac{1}{s-1}})$ for $s > 2$, and defer the proof to the full version of our paper.

Corollary 1. *For $n, m \geq 1$, let $p_1, p_2, \ldots, p_m$ be the pmf of a Zipf distribution $\mathcal{Z}_{m,s}$ with parameter $s > 2$. If $n = m$, then for any time $1 \leq t \leq n$, for any $\delta > 0$:*

$$\Pr\Big[\,Y(t) \geq (1+\delta)\cdot(m^{\frac{1}{s-1}}+\log m)\,\Big] \leq \exp\left(\frac{-\delta^2}{2\cdot(1+\delta)}\cdot\frac{m^{\frac{1}{s-1}}}{H_{m,s}}\right),$$

where $Y(t)$ is a random variable representing the size of the replica cache for Ω_{bas} at time t.

7.3 Mode 2: Client state polylogarithmic in m

We can also restrict ERS when the client only has poly-logarithmic state available. In particular, we can set:

$$\begin{aligned} \text{sketch size } \kappa &= \log^2(\rho + m) \\ \text{we also know } \#\overline{\mathsf{DX}} &= O(\rho + m), \\ \rho &= m \cdot H_{m,s}, \text{ and} \\ n &= \theta(m). \end{aligned}$$

Since $\log^2(\rho + m)$ is $\omega(\log(\rho + m))$, we can compute the communication complexity of $\mathsf{CacheShuffle}$ as:

$$\begin{aligned} \mathsf{comm}_{\mathsf{CacheShuffle}}(\overline{\mathsf{DX}}) &= O\left((\rho + m) \cdot \frac{\log(\rho + m)}{\log(\log^2(\rho + m))}\right) \\ &= O\left((m \cdot H_{m,s} + m) \cdot \frac{\log(m \cdot H_{m,s} + m)}{\log(\log^2(m \cdot H_{m,s} + m))}\right) \\ &= O\left(m \cdot H_{m,s} \cdot \frac{\log(m \cdot H_{m,s})}{\log(\log(m \cdot H_{m,s}))}\right) \end{aligned}$$

since for $s > 1$, $H_{m,s} = O(\log m)$, we have:

$$\begin{aligned} \mathsf{comm}_{\mathsf{CacheShuffle}}(\overline{\mathsf{DX}}) &= O\left(m \cdot \log m \cdot \frac{\log m + \log\log m}{\log(\log m + \log\log m)}\right) \\ &= O\left(\frac{m \cdot \log^2 m}{\log\log m}\right). \end{aligned}$$

Amortized communication complexity. From Equation 5, the amortized query complexity of Ω_{bas}:

$$\begin{aligned} \mathsf{comm}_{\Omega_{\mathsf{bas}}}(q) &= \frac{1}{n} \cdot \left(\sum_{i=1}^{n} O(1) + O\left(\frac{m \cdot \log^2 m}{\log\log m}\right)\right) \\ &= O(1) + O\left(\frac{m \cdot \log^2 m}{n \cdot \log(\log m)}\right) \\ &= O\left(\frac{\log^2 m}{\log\log m}\right), \text{ since } n = O(m). \end{aligned}$$

Amortized round complexity. From Equation 6, knowing that the round complexity of ω_{bas} is $O(1)$, we have the amortized round complexity of Ω_{bas}:

$$\begin{aligned}\mathsf{rounds}_{\Omega_{\mathsf{bas}}}(q) &= \frac{1}{n} \cdot \left(\sum_{i=1}^{n} O(1) + \mathsf{rounds}_{\mathsf{Shuffle}}(\overline{\mathsf{DX}}) \right), \\ &= O(1) + O(\mathsf{comm}_{\Omega_{\mathsf{bas}}}(q)) \\ &= O\left(\frac{\log^2 m}{\log \log m} \right).\end{aligned}$$

Correctness. We do not have a closed-form bound on the cache size for this mode. Instead, we use simulations to compute correctness loss during the epoch for different values of s. Our experiments use a dictionary of size 2^{22} with an epoch length of 2^{22} queries. When ERS retrieves a dummy instead of the correct value for a queried label, we refer to it as a *false positive* response. In Figure 7 we show false positive rates when the replica cache size is bounded by the sketch size κ. This shows us that correctness increases when s increases. We show additional experimental results in the full version of our paper. Our artifact with reproducibility instructions for the experiments can be found at https://anonymous.4open.science/r/qeq-suppression/.

	$\kappa = M = \log m$	$\kappa = M = \log^2 m$	$\kappa = M = m^{\frac{1}{s}}$	$\kappa = M = m^{\frac{1}{s-1}}$
$s = 1.0001$	99.99	99.97	0	n/a
$s = 2.0001$	99.97	89.42	0	0
$s = 3.0001$	98.90	0	0	0
$s = 4.0001$	69.05	0	0	0

Fig. 7. False positive percentages for different combinations of Zipf parameter s, sketch size κ when the maximum cache size M is set to κ. 0 indicates all correct responses.

References

1. Adkins, D., Agarwal, A., Kamara, S., Moataz, T.: Encrypted blockchain databases. In: Proceedings of the 2nd ACM Conference on Advances in Financial Technologies. pp. 241–254 (2020)
2. Agarwal, A., Espiritu, Z.: Sequentially consistent concurrent encrypted multimaps. In: 2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P). IEEE (2025)
3. Agarwal, A., Kamara, S., Moataz, T.: Concurrent Encrypted Multimaps. In: Advances in Cryptology – ASIACRYPT 2024: 30th International Conference on the Theory and Application of Cryptology and Information Security, Kolkata, India, December 9–13, 2024, Proceedings. Part IV. pp. 169–201. Springer-Verlag, Berlin, Heidelberg (Dec 2024). https://doi.org/10.1007/978-981-96-0894-2_6, https://doi.org/10.1007/978-981-96-0894-2_6
4. Ajtai, M., Komlós, J., Szemerédi, E.: An o(n log n) sorting network. In: ACM Symposium on Theory of Computing (STOC '83). pp. 1–9 (1983)
5. Amjad, G., Kamara, S., Moataz, T.: Injection-secure structured and searchable symmetric encryption. In: Guo, J., Steinfeld, R. (eds.) Advances in Cryptology - ASIACRYPT 2023 - 29th International Conference on the Theory and Application of Cryptology and Information Security, Guangzhou, China, December 4-8, 2023, Proceedings, Part VI. Lecture Notes in Computer Science, vol. 14443, pp. 232–262. Springer (2023). https://doi.org/10.1007/978-981-99-8736-8_8, https://doi.org/10.1007/978-981-99-8736-8_8
6. Amjad, G., Patel, S., Persiano, G., Yeo, K., Yung, M.: Dynamic Volume-Hiding Encrypted Multi-Maps with Applications to Searchable Encryption. Proceedings on Privacy Enhancing Technologies (2023), https://petsymposium.org/popets/2023/popets-2023-0025.php
7. Asharov, G., Naor, M., Segev, G., Shahaf, I.: Searchable symmetric encryption: Optimal locality in linear space via two-dimensional balanced allocations. In: ACM Symposium on Theory of Computing (STOC '16). pp. 1101–1114. STOC '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2897518.2897562, http://doi.acm.org/10.1145/2897518.2897562
8. Batcher, K.: Sorting networks and their applications. In: Proceedings of the Joint Computer Conference. pp. 307–314 (1968)
9. Bienstock, A., Patel, S., Seo, J.Y., Yeo, K.: Near-optimal oblivious key-value stores for efficient psi, psu and volume-hiding multi-maps. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 301–318 (2023)
10. Blackstone, L., Kamara, S., Moataz, T.: Revisiting leakage abuse attacks. In: Network and Distributed System Security Symposium (NDSS '20) (2020)
11. Bossuat, A., Bost, R., Fouque, P.A., Minaud, B., Reichle, M.: Sse and ssd: page-efficient searchable symmetric encryption. In: Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part III 41. pp. 157–184. Springer (2021)
12. Bost, R.: Sophos - forward secure searchable encryption. In: ACM Conference on Computer and Communications Security (CCS '16) (20016)
13. Brézot, T., Hébant, C.: Findex: A Concurrent and Database-Independent Searchable Encryption Scheme (2024), https://eprint.iacr.org/2024/1541, publication info: Preprint.
14. Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: ACM Conference on Communications and Computer Security (CCS '15). pp. 668–679. ACM (2015)

15. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: Advances in Cryptology - CRYPTO '13. Springer (2013)
16. Cash, D., Tessaro, S.: The locality of searchable symmetric encryption. In: Advances in Cryptology - EUROCRYPT 2014 (2014)
17. Cash, D., Jaeger, J., Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M., Steiner, M.: Dynamic searchable encryption in very-large databases: Data structures and implementation. In: Network and Distributed System Security Symposium (NDSS '14) (2014)
18. Cash, D., Ng, R., Rivkin, A.: Improved structured encryption for sql databases via hybrid indexing. In: Applied Cryptography and Network Security: 19th International Conference, ACNS 2021, Kamakura, Japan, June 21–24, 2021, Proceedings, Part II. pp. 480–510. Springer (2021)
19. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Advances in Cryptology - ASIACRYPT '10. Lecture Notes in Computer Science, vol. 6477, pp. 577–594. Springer (2010)
20. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM symposium on Cloud computing. pp. 143–154 (2010)
21. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: Improved definitions and efficient constructions. In: ACM Conference on Computer and Communications Security (CCS '06). pp. 79–88. ACM (2006)
22. Demertzis, I., Papamanthou, C.: Fast searchable encryption with tunable locality. In: ACM International Conference on Management of Data (SIGMOD '17). pp. 1053–1067. SIGMOD '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3035918.3064057, http://doi.acm.org/10.1145/3035918.3064057
23. Espiritu, Z., George, M., Kamara, S., Qin, L.: Synq: Public Policy Analytics Over Encrypted Data. In: 2024 IEEE Symposium on Security and Privacy (SP). pp. 146–165. IEEE, San Francisco, CA, USA (May 2024). https://doi.org/10.1109/sp54263.2024.00085, https://ieeexplore.ieee.org/document/10646768/
24. Faber, S., Jarecki, S., Krawczyk, H., Nguyen, Q., Rosu, M., Steiner, M.: Rich queries on encrypted data: Beyond exact matches. In: European Symposium on Research in Computer Security (ESORICS '15). Lecture Notes in Computer Science. vol. 9327, pp. 123–145 (2015)
25. Falzon, F., Markatou, E.A., Espiritu, Z., Tamassia, R.: Range Search over Encrypted Multi-Attribute Data. Proceedings of the VLDB Endowment **16**(4), 587–600 (Dec 2022). https://doi.org/10.14778/3574245.3574247, https://dl.acm.org/doi/10.14778/3574245.3574247
26. George, M., Kamra, S., Moataz, T.: Structured encryption and dynamic leakage suppression. In: Advances in Cryptology - EUROCRYPT 2021 (2021)
27. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. Journal of the ACM **43**(3), 431–473 (1996)
28. Grubbs, P., Khandelwal, A., Lacharité, M.S., Brown, L., Li, L., Agarwal, R., Ristenpart, T.: Pancake: Frequency smoothing for encrypted data stores. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 2451–2468 (2020)
29. Grubbs, P., Khandelwal, A., Lacharité, M.S., Brown, L., Li, L., Agarwal, R., Ristenpart, T.: Pancake: Frequency smoothing for encrypted data stores. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 2451–2468 (2020)
30. Grubbs, P., Lacharité, M., Minaud, B., Paterson, K.G.: Pump up the volume: Practical database reconstruction from volume leakage on range queries. In: Lie, D.,

Mannan, M., Backes, M., Wang, X. (eds.) Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 315–331. ACM (2018). https://doi.org/10.1145/3243734.3243864, https://doi.org/10.1145/3243734.3243864

31. Grubbs, P., Lacharite, M.S., Minaud, B., Paterson, K.G.: Pump up the Volume: Practical Database Reconstruction from Volume Leakage on Range Queries. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 315–331. CCS '18, Association for Computing Machinery, New York, NY, USA (Oct 2018). https://doi.org/10.1145/3243734.3243864, https://dl.acm.org/doi/10.1145/3243734.3243864
32. Grubbs, P., Lacharité, M., Minaud, B., Paterson, K.G.: Learning to reconstruct: Statistical learning theory and encrypted database attacks. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019. pp. 1067–1083. IEEE (2019). https://doi.org/10.1109/SP.2019.00030, https://doi.org/10.1109/SP.2019.00030
33. Gui, Z., Johnson, O., Warinschi, B.: Encrypted Databases: New Volume Attacks against Range Queries. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 361–378. CCS '19, Association for Computing Machinery, New York, NY, USA (Nov 2019). https://doi.org/10.1145/3319535.3363210, https://dl.acm.org/doi/10.1145/3319535.3363210
34. Gui, Z., Paterson, K.G., Patranabis, S., Warinschi, B.: Swissse: System-wide security for searchable symmetric encryption. Proceedings on Privacy Enhancing Technologies (2024)
35. Islam, M.S., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In: Network and Distributed System Security Symposium (NDSS '12) (2012)
36. Jutla, C., Patranabis, S.: Efficient searchable symmetric encryption for join queries. In: Advances in Cryptology–ASIACRYPT 2022: 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5–9, 2022, Proceedings, Part III. pp. 304–333. Springer (2023)
37. Kamara, S., Moataz, T.: Boolean searchable symmetric encryption with worst-case sub-linear complexity. In: Advances in Cryptology - EUROCRYPT '17 (2017)
38. Kamara, S., Moataz, T.: Computationally volume-hiding structured encryption. In: Advances in Cryptology - Eurocrypt' 19 (2019)
39. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Financial Cryptography and Data Security (FC '13) (2013)
40. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: ACM Conference on Computer and Communications Security (CCS '12). ACM Press (2012)
41. Kamara, S., Kati, A., Moataz, T., DeMaria, J., Park, A., Treiber, A.: MAPLE: MArkov Process Leakage attacks on Encrypted Search. Proceedings of Privacy Enhancing Technologies **2024**(1), 430–446 (2024). https://doi.org/10.56553/popets-2024-0025
42. Kamara, S., Kati, A., Moataz, T., Schneider, T., Treiber, A., Yonli, M.: SoK: Cryptanalysis of Encrypted Search with LEAKER - A framework for LEakage AttacK Evaluation on Real-world data (2021), https://eprint.iacr.org/2021/1035, publication info: Published elsewhere. Minor revision. EuroS&P 2022
43. Kamara, S., Kati, A., Moataz, T., Schneider, T., Treiber, A., Yonli, M.: Sok: Cryptanalysis of encrypted search with leaker – a framework for leakage attack evaluation

on real-world data. In: 2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P). pp. 90–108 (2022). https://doi.org/10.1109/EuroSP53844.2022.00014

44. Kamara, S., Moataz, T.: SQL on structurally-encrypted databases. In: Advances in Cryptology–ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part I 24. pp. 149–180. Springer (2018)
45. Kamara, S., Moataz, T.: Design and analysis of a stateless encrypted document database (2023), https://www.mongodb.com/collateral/stateless-document-database-encryption-scheme
46. Kamara, S., Moataz, T., Ohrimenko, O.: Structured encryption and leakage suppression. In: Advances in Cryptology - CRYPTO '18 (2018)
47. Kamara, S., Moataz, T., Park, A., Qin, L.: A Decentralized and Encrypted National Gun Registry. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1520–1537 (May 2021). https://doi.org/10.1109/SP40001.2021.00072, https://ieeexplore.ieee.org/document/9519474, iSSN: 2375-1207
48. Kornaropoulos, E.M., Papamanthou, C., Tamassia, R.: The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1223–1240. IEEE (2020)
49. Kornaropoulos, E.M., Papamanthou, C., Tamassia, R.: Response-Hiding Encrypted Ranges: Revisiting Security via Parametrized Leakage-Abuse Attacks. In: IEEE Symp. on Security and Privacy. S&P (2021)
50. Lacharité, M., Minaud, B., Paterson, K.G.: Improved reconstruction attacks on encrypted data using range query leakage. IACR Cryptology ePrint Archive **2017**, 701 (2017), http://eprint.iacr.org/2017/701
51. Larsen, K.G., Nielsen, J.B.: Yes, there is an oblivious ram lower bound! In: Annual International Cryptology Conference. pp. 523–542. Springer (2018)
52. Lehmkuhl, R., Henzinger, A., Corrigan-Gibbs, H.: Distributional private information retrieval. In: 34th USENIX Security Symposium (USENIX Security 25). pp. 3377–3393. USENIX Association, Seattle, WA (Aug 2025), https://www.usenix.org/conference/usenixsecurity25/presentation/lehmkuhl
53. Liu, C., Zhu, L., Wang, M., Tan, Y.A.: Search pattern leakage in searchable encryption: Attacks and new construction. Inf. Sci. **265**, 176–188 (May 2014). https://doi.org/10.1016/j.ins.2013.11.021, http://dx.doi.org/10.1016/j.ins.2013.11.021
54. Maiyya, S., Vemula, S.C., Agrawal, D., El Abbadi, A., Kerschbaum, F.: Waffle: An online oblivious datastore for protecting data access patterns. Proceedings of the ACM on Management of Data **1**(4), 1–25 (2023)
55. Markatou, E.A., Falzon, F., Espiritu, Z., Tamassia, R.: Attacks on Encrypted Response-Hiding Range Search Schemes in Multiple Dimensions. Proceedings on Privacy Enhancing Technologies **2023**(4), 204–223 (Oct 2023). https://doi.org/10.56553/popets-2023-0106, https://petsymposium.org/popets/2023/popets-2023-0106.php
56. Ohrimenko, O., Goodrich, M.T., Tamassia, R., Upfal, E.: The melbourne shuffle: Improving oblivious storage in the cloud. In: Automata, Languages, and Programming: 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II 41. pp. 556–567. Springer (2014)
57. Oya, S., Kerschbaum, F.: {IHOP}: Improved statistical query recovery against searchable symmetric encryption through quadratic optimization. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 2407–2424 (2022)

58. Pappas, V., Krell, F., Vo, B., Kolesnikov, V., Malkin, T., Choi, S.G., George, W., Keromytis, A., Bellovin, S.: Blind seer: A scalable private dbms. In: Security and Privacy (SP), 2014 IEEE Symposium on. pp. 359–374. IEEE (2014)
59. Patel, S., Persiano, G., Yeo, K.: CacheShuffle: A Family of Oblivious Shuffles. In: Chatzigiannakis, I., Kaklamanis, C., Marx, D., Sannella, D. (eds.) 45th International Colloquium on Automata, Languages, and Programming (ICALP 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 107, pp. 161:1–161:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2018). https://doi.org/10.4230/LIPIcs.ICALP.2018.161, https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ICALP.2018.161
60. Patel, S., Persiano, G., Yeo, K.: Lower bounds for encrypted multi-maps and searchable encryption in the leakage cell probe model. In: Annual International Cryptology Conference. pp. 433–463. Springer (2020)
61. Patel, S., Persiano, G., Yeo, K., Yung, M.: Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019. pp. 79–93. ACM (2019). https://doi.org/10.1145/3319535.3354213, https://doi.org/10.1145/3319535.3354213
62. Persiano, G., Yeo, K.: Limits of breach-resistant and snapshot-oblivious rams. Cryptology ePrint Archive (2023)
63. Song, D., Wagner, D., Perrig, A.: Practical techniques for searching on encrypted data. In: IEEE Symposium on Research in Security and Privacy. pp. 44–55. IEEE Computer Society (2000)
64. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path oram: An extremely simple oblivious ram protocol. In: ACM Conference on Computer and Communications Security (CCS '13) (2013)
65. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: Network and Distributed System Security Symposium (NDSS '14) (2014)
66. Wang, J., Chow, S.S.: Simple storage-saving structure for volume-hiding encrypted multi-maps: (a slot in need is a slot indeed). In: IFIP Annual Conference on Data and Applications Security and Privacy. pp. 63–83. Springer (2021)
67. Wang, J., Sun, S.F., Li, T., Qi, S., Chen, X.: Practical volume-hiding encrypted multi-maps with optimal overhead and beyond. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 2825–2839 (2022)
68. Wang, J., Sun, S.F., Li, T., Qi, S., Chen, X.: Practical Volume-Hiding Encrypted Multi-Maps with Optimal Overhead and Beyond. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 2825–2839. CCS '22, Association for Computing Machinery, New York, NY, USA (Nov 2022). https://doi.org/10.1145/3548606.3559345, https://doi.org/10.1145/3548606.3559345
69. Wang, X.S., Nayak, K., Liu, C., Chan, T.H.H., Shi, E., Stefanov, E., Huang, Y.: Oblivious Data Structures. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 215–226. CCS '14, Association for Computing Machinery, New York, NY, USA (Nov 2014). https://doi.org/10.1145/2660267.2660314, https://doi.org/10.1145/2660267.2660314
70. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: USENIX Security Symposium (2016)