Searchable Encryption with Optimal Locality: Achieving Sublogarithmic Read Efficiency

Ioannis Demertzis[1], Dimitrios Papadopoulos[1,2], and Charalampos Papamanthou[1]

[1] University of Maryland
[2] Hong Kong University of Science and Technology

Abstract. We propose the first linear-space searchable encryption scheme with constant locality and *sublogarithmic* read efficiency, strictly improving the previously best known read efficiency bound (Asharov et al., STOC 2016) from $\Theta(\log N \log\log N)$ to $O(\log N/\log\log N)$, where N is the size of the dataset. Our scheme is *size-sensitive*, meaning our bound is tight only for keyword lists whose sizes lie within the specific range $(N^{1-1/\log\log N}, N/\log^2 N]$—outside this range the read efficiency improves to $O(\log^{2/3} N)$. For our construction we develop two techniques that can be of independent interest: New probability bounds for the offline two-choice allocation problem and a new I/O-efficient oblivious RAM with $o(\sqrt{n})$ bandwidth overhead and zero failure probability.

1 Introduction

Searchable Encryption (SE), proposed by Song et al. [23] in 2000, enables a data owner to outsource a private dataset $\mathcal{D}$ to a server, so that the latter can answer keyword queries without learning too much information about the underlying dataset and the posed queries. An alternative to expensive primitives such as oblivious RAM and fully-homomorphic encryption, SE schemes are practical at the expense of formally-specified leakage. In typical SE schemes, the data owner prepares a private index which is sent to the server. To perform a query on a keyword w, the data owner engages in a protocol with the server such that by the end of the protocol the data owner retrieves the list of document identifiers $\mathcal{D}(w)$ of documents containing w. During this process, the server should learn nothing except for the (number of) retrieved document identifiers—referred to as (size of) *access pattern*—and whether the keyword search query w was repeated in the past or not—referred to as *search pattern*.

To retrieve the document identifiers $\mathcal{D}(w)$ (also referred to as *keyword list* in the rest of the paper), most SE schemes so far require the server access approximately $|\mathcal{D}(w)|$ randomly-assigned memory locations [23,18,9,17,7,24]. While this random allocation is essential for security reasons, it creates a big bottleneck when accessing large indexes stored on disk—due to expensive I/Os. Therefore the aforementioned schemes cannot scale for data stored on disk[3] due to their poor *locality*—the number of non-contiguous memory locations that must be read to retrieve the result.

[3] Demertzis and Papamanthou [10] recently showed that SE with good locality improves practical performance for in-memory data as well, due to the reduced number of server cryptographic operations required to retrieve the result.

Locality and Read Efficiency Trade-offs. One trivial way to design an SE scheme that has optimal locality $L = 1$ is to have the client download the whole encrypted index for every query w. Unfortunately, such an approach increases the bits that have to be read by a factor of $N/|\mathcal{D}(w)|$, where N is the total size of the index. Cash et al. [7] were the first to observe this trade-off: To improve the locality of SE, one should expect to read additional entries per query. The ratio of the total number of entries read over the size of the initial query result was defined as *read efficiency*. This trade-off was subsequently formalized by Cash and Tessaro [8] who proved it is impossible[4] to construct an SE scheme with linear space, optimal locality and optimal read efficiency.

In response to this impossibility result, several positive results with various trade-offs have appeared. Cash and Tessaro [8] presented a scheme with $\Theta(N \log N)$ space, $O(1)$ read efficiency and $O(\log N)$ locality, which was later improved to $O(1)$ by Asharov et al. [5]. Demertzis and Papamanthou [10] presented a scheme with bounded locality $O(N^{\epsilon})$, $O(1)$ read efficiency but linear space (where $\epsilon < 1$ is a constant). However, due to practical reasons, the most challenging question has been a study of these trade-offs in a *constant-locality* and *linear-space* regime.

SE with Constant Locality and Linear Space. Asharov et al. [5] presented two SE schemes with linear space and constant locality: The first one (A1) has very low read efficiency $\Theta(\log\log N \log^2 \log\log N)$ but is based on the assumption that no keyword list $\mathcal{D}(w)$ has size more than $N^{1-1/\log\log N}$[5]. The second one (A2) has $\Theta(\log N \log\log N)$ read efficiency and does not depend on any assumptions about the input dataset. To the best of our knowledge A2 is the best SE scheme with $O(1)$ locality and linear space known to-date for general datasets.

Our Contribution. We design the first linear-space SE scheme for general datasets with constant locality and *sublogarithmic* read efficiency, strictly improving upon the best known scheme A2 by Asharov et al. [5]. In particular the read efficiency of our scheme is $O(\frac{\log N}{\log\log N})$ as opposed to A2's $\Theta(\log N \log\log N)$. Note that we make use of $O(.)$ notation instead of $\Theta(.)$ notation: Unlike A2, our scheme achieves its worst-case $O(\log N/\log\log N)$ bound only for keyword lists whose sizes are within a small range. For all other ranges, the read efficiency is much better, in the order of either $O(\log^{2/3} N)$ or even $O(\log\log N \log^2 \log\log N)$.

1.1 Our Techniques

Our techniques (as well as previous works on low-locality SE) are using the notion of an *allocation algorithm*, whose goal is to store the dataset's keyword lists in memory such that each keyword list $\mathcal{D}(w)$ can be efficiently retrieved (both in terms of locality and read efficiency) by accessing memory locations that are *independent* of the distribution

[4] The result holds for a setting where lists $\mathcal{D}(w)$ are stored in consecutive memory locations.

[5] We tested this assumption for 4 real datasets: One containing crime records in Chicago since 2001 [1], the Enron email dataset [2], the USPS dataset [4] and the TPC-H dataset [3]. The assumption was not violated only in the Enron email dataset. For the crimes dataset, for example, the assumption was violated in 12 out of 21 attributes for 31% of the keywords on average.

of the rest of the underlying SE dataset—this is needed for security reasons, and in particular for ensuring that accessing a list $\mathcal{D}(w)$ does not reveal information about the rest of the dataset. Common techniques to achieve this is to store keyword lists using a balls-and-bins procedure [5].

Our main observation is that in order to build an SE scheme with sublogarithmic read efficiency for general datasets, we must only allocate the dataset's large keyword lists—those that have size greater than $N^{1-1/\log\log N}$. Smaller lists can be easily allocated using scheme A1 of Asharov et al. [5] as a black box which provides much better read efficiency than sublogarithmic, as we discussed before.

Offline Two-Choice Allocation. Our main allocation algorithm will be using an *offline two-choice allocation* (OTA) procedure [21] as a black box. In an OTA, there are m balls and n bins. For each ball two possible bins are chosen independently and uniformly at random. After *all choices* have been made, one can run a maximum flow algorithm to find the final assignment of balls to bins such that the maximum load is minimized. This strategy achieves an almost perfectly balanced allocation (where the maximum load is at most $\lceil m/n \rceil + 1$) with probability at least $1 - O(1/n)$ [21]—see Lemma 1.

Central Idea: One OTA Per Size and Then Merge. We use one OTA separately for every size s that falls in the range

$$(N^{1-1/\log\log N}, N/\log^2 N]$$

as follows: Let $\mathbf{A}_s$ be an array of M buckets $\mathbf{A}_s[1], \mathbf{A}_s[2], \ldots, \mathbf{A}_s[M]$, for some appropriately chosen M. One can visualize a bucket $\mathbf{A}_s[i]$ as a vertical structure of unbounded capacity. Let k_s be the number of keyword lists of size s and let $b_s = M/s$ be the number of superbuckets in $\mathbf{A}_s$, where a supebucket is a collection of s consecutive buckets in $\mathbf{A}_s$. We perform an OTA of k_s keyword lists to the b_s superbuckets. From [21], there will be at most $\lceil k_s/b_s \rceil + 1$ lists of size s in each superbucket with probability at least $1 - O(1/b_s)$, meaning the capacity of each bucket due to lists of size s will be at most $\lceil k_s/b_s \rceil + 1$ with the same probability, given there are s buckets in an superbucket.

Our final allocation involves merging the arrays $\mathbf{A}_s$ for all sizes s into an array $\mathbf{A}$ of M buckets. To bound the final load of each bucket $\mathbf{A}[i]$ in the merged array $\mathbf{A}$ one can compute

$$\sum_s (\lceil k_s/b_s \rceil + 1)$$

which turns out to be $O(M/N + \log N/\log\log N)$—see Lemma 5. By setting $M = N\log\log N/\log N$, our allocation occupies linear space and every bucket of $\mathbf{A}$ has load $O(\log N/\log\log N)$—therefore to read one list of size s, one reads the two superbuckets initially picked by the OTA and therefore the read efficiency is $O(\log N/\log\log N)$.

Handling Bucket Overflows with Additional Stashes. Our analysis above assumes the maximum load of each bucket is at most $\lceil k_s/b_s \rceil + 1$. However, there is a noticeable probability $O(1/b_s)$ of overflowing beyond this bound—this will cause our allocation algorithm to fail, leaking information about the dataset. To deal with this problem, for each size s, we place the lists of size s that overflow in a stash $\mathbf{B}_s$ (at the server) that can store up to $O(\log^2 N)$ such overflowing lists. In particular we prove that when the OTA described previously is performed for lists of size s greater than $N^{1-1/\log\log N}$ *but less than* $N/\log^2 N$, there are at most $O(\log^2 N)$ lists of size s overflowing with non-negligible probability and therefore our stashes $\mathbf{B}_s$ suffice, see Lemma 6. We stress that the condition $s \leq N/\log^2 N$ is necessary for deriving the non-negligible probability bound, justifying the pick of $N/\log^2 N$ as the endpoint of the range where we apply OTA.

New Probability Bounds for OTA. Our proof for the $O(\log^2 N)$ stash size extends the analysis of [21] in non-trivial ways, see Section 3: First, in Theorem 1 we show that in an OTA, the probability that more than τ bins overflow decreases with $(1/\tau)^\tau$. Our proof requires showing the 0/1 random variables indicating whether a bin overflows or not are negatively associated. Second, in Theorem 2 we show the probability that an OTA of m balls to n bins yields a maximum load of more than $\lceil m/n \rceil + \tau$ is at most $O(1/n)^\tau + exp(-n)$.

Accessing Stashes Obliviously. Because keyword lists of size s can now live in the stash $\mathbf{B}_s$, retrieving an arbitrary keyword list $\mathcal{D}(w)$ is a two-step process: First, access the superbuckets that were initially assigned by the OTA and then access a position x in the stash. In case $\mathcal{D}(w)$ is not in the stash (because it was not an overflowing list), x should be still assigned as any unoccupied (dummy) position of the stash, if such a position exists. If not, there will be a collision, in which case the adversary can deduce information about the distribution of the dataset, e.g., that the dataset contains at least $\log^2 N$ lists of size $|\mathcal{D}(w)|$. To avoid such leakage, the stash must be accessed obliviously.

New ORAM with $o(\sqrt{n})$ Bandwidth, $O(1)$ Locality and Zero Failure Probability. Since the stash has only $\log^2 N$ entries of size $|\mathcal{D}(w)|$ each, one can access it obliviously by reading it all. This however increases read efficiency to $\log^2 N$, which is no longer sublogarithmic. Thus we need an ORAM with $O(1)$ locality and $o(\sqrt{N})$ bandwidth. At the same time our ORAM should fail with zero probability since it will be applied on only $\log^2 N$ indices. In Section 4, we devise a new ORAM satisfying the above (with $O(n^{1/3}\log^2 n)$ bandwidth) that is based on one recursive application of Goldreich's and Ostrovsky's square-root ORAM [12]. This protocol can be of independent interest.

Handling Keyword Lists with Sizes Greater Than $N/\log^2 N$. We develop a new allocation algorithm called $\mathsf{AllocateLarge}(\mathsf{min}, \mathsf{max})$ for these lists—see Section 5.4. In fact, the algorithm can be used for any list with size in the range $(\mathsf{min}, \mathsf{max}]$ and works as follows: Let $\mathbf{A}$ be an array that has $2N$ entries decomposed in N/max buckets of capacity $2\mathsf{max}$ each. To store a list of size s in $(\mathsf{min}, \mathsf{max}]$, we pick a bucket uniformly at random, from the ones that have not been picked before for the specific size s, and store the list in this bucket. If there is no space available in the picked bucket, we store this list in a stash $\mathbf{B}_s$ (that is kept separately for each size s) that can store up to N/max lists of size s (as before, $\mathbf{B}_s$ will have to be accessed obliviously). In Theorem 8 we prove that this algorithm will never cause the stashes to overflow and will occupy linear space. So one could wonder, why cannot it be applied for the range $[1, N]$ covering all sizes? The reason is that its read efficiency is $O(\mathsf{max}/\mathsf{min})$: To read a list with the smallest size min requires reading a bucket of size $2\mathsf{max}$. Therefore to maintain sublogarithmic read efficiency, we apply it in two appropriate ranges, namely $(N/\log^2 N, N/\log^{4/3} N]$ and $(N/\log^{4/3} N, N/\log^{2/3} N]$. For the final range $(N/\log^{2/3} N, N]$, we just read the whole dataset. Overall, for all sizes $> N/\log^2 N$ we get a read efficiency $O(\log^{2/3} N)$.

1.2 Paper Structure

We present definitions of SE and ORAM in Section 2. Section 3 presents our new analysis of OTA. Section 4 presents our new ORAM scheme with $o(\sqrt{n})$ bandwidth. Section 5 presents our detailed allocation algorithms for all the different ranges. Section 6 presents our full construction, our final result stated formally (Theorem 9) and our proof of security. We conclude in Section 7.

2 Notation and Definitions

Our protocols involve a client and a server. We use the notation $\langle C', S' \rangle \leftrightarrow \Pi \langle C, S \rangle$ to indicate that a protocol Π is executed between a client with input C and a server with input S. After the execution of the protocol the client receives C' and the server receives S'. Server operations are written in light gray background. All other operations are performed by the client. The client typically interacts with the server via two operations. First, via an **`Encrypt-And-Write`** $data$ operation, with which the client encrypts $data$ locally with a CPA-secure encryption scheme and writes the encrypted data $data$ remotely to server. Second, via a **`Read-And-Decrypt`** $data$ operation, with which the client reads encrypted data $data$ from server and decrypts them locally.

In the following, $\mathcal{D}$ will denote the searchable encryption dataset (SE dataset) which is a set of keywords lists $\mathcal{D}(w_i)$. Each keyword list $\mathcal{D}(w_i)$ is a set of keyword-document pairs (w_i, id), called *elements*, where id is the document identifier containing keyword w_i. We denote with N the size of our dataset, i.e., $N = \sum_{w \in \mathbf{W}} |\mathcal{D}(w)|$, where $\mathbf{W}$ is the set of unique keywords of our dataset $\mathcal{D}$. Without loss of generality, we will assume that all keyword lists $\mathcal{D}(w_i)$ have size $|\mathcal{D}(w_i)|$ that is a power of two. This can always be enforced by padding with dummy elements, and will only increase the space at most by a factor of 2. Finally, a function $f(\kappa)$ is *negligible*, denoted $\mathsf{neg}(\kappa)$, if for sufficiently large κ it is less than $1/p(\kappa)$, for all polynomials $p(\kappa)$.

2.1 Searchable Encryption

Our new Searchable Encryption (SE) scheme uses a modification of the square-root ORAM protocol as a black box, which is a two-round protocol. Therefore to model our SE scheme we use the protocol-based definition as proposed by Stefanov et al. [24]. An SE scheme consists of protocols (SETUP, SEARCH) as defined below.

- $\langle st, \mathcal{I} \rangle \leftrightarrow \text{SETUP}\langle (1^\kappa, \mathcal{D}), 1^\kappa \rangle$: SETUP takes as input the security parameter κ and an SE dataset $\mathcal{D}$ and outputs a secret state st (for the client), and an encrypted index $\mathcal{I}$ (for the server).

```
$bit \leftarrow \textbf{Real}^{\mathsf{SE}}(\kappa)$:
1: $D_0 \leftarrow \mathsf{Adv}(1^\kappa)$;
2: $\langle st_0, \mathcal{I}_0 \rangle \leftrightarrow \textsc{Setup}\langle (1^\kappa, \mathcal{D}_0), 1^\kappa \rangle$;
3: for $k = 1$ to $q$ do
4:     $w_k \leftarrow \mathsf{Adv}(1^k, \mathcal{I}_0, M_1, \ldots, M_{k-1})$;
5:     $\langle (\mathcal{D}(w_k), st_k), \mathcal{I}_k \rangle \leftrightarrow \textsc{Search}\langle (st_{k-1}, w_k), \mathcal{I}_{k-1} \rangle$;
6:     Let $M_k$ be the messages from client to server in the SEARCH protocol above;
7: $bit \leftarrow \mathsf{Adv}(1^k, \mathcal{I}_0, M_1, M_2, \ldots, M_q)$;
8: return $bit$;
$bit \leftarrow \textbf{Ideal}^{\mathsf{SE}}_{\mathcal{L}_1,\mathcal{L}_2}(\kappa)$:
1: $D_0 \leftarrow \mathsf{Adv}(1^\kappa)$;
2: $(st_{\mathcal{S}}, \mathcal{I}_0) \leftarrow \textsc{SimSetup}(1^\kappa, \mathcal{L}_1(\mathcal{D}_0))$;
3: for $k = 1$ to $q$ do
4:     $w_k \leftarrow \mathsf{Adv}(1^k, \mathcal{I}_0, M_1, \ldots, M_{k-1})$;
5:     $(st_{\mathcal{S}}, M_k, \mathcal{I}_k) \leftarrow \textsc{SimSearch}(st_{\mathcal{S}}, \mathcal{L}_2(w_k), \mathcal{I}_{k-1})$;
6: $bit \leftarrow \mathsf{Adv}(1^k, \mathcal{I}_0, M_1, M_2, \ldots, M_q)$;
7: return $bit$;
```

Fig. 1. Real and ideal experiments for the SE scheme.

- $\langle (\mathcal{D}(w), st'), \mathcal{I}' \rangle \leftrightarrow \textsc{Search}\langle (st, w), \mathcal{I} \rangle$: SEARCH is a protocol between the client and the server, where the client's input is the secret state st, and a keyword w. Server's input is the encrypted index $\mathcal{I}$. Client's output is the the set of document identifiers $\mathcal{D}(w)$ matching the keyword w and an updated secret state st' and the server's output is an updated encrypted index $\mathcal{I}'$.

Just like in previous works [5], the goal of our SE protocols is for the client to retrieve the document identifiers (i.e., the list $\mathcal{D}(w)$) corresponding to a specific keyword w. The document themselves can be downloaded from the server in a second round, by just providing $\mathcal{D}(w)$. This part is orthogonal to our protocols and we do not consider/model it here explicitly. The correctness definition of SE is given in Appendix A. We now give the security definition.

Definition 1 (Security of SE). *An SE scheme* (SETUP, SEARCH) *is secure in the semi-honest model if for any PPT adversary* Adv*, there exists a stateful PPT simulator* (SIMSETUP, SIMSEARCH) *such that*

$$|\Pr[\textbf{Real}^{\mathsf{SE}}(\kappa) = 1] - \Pr[\textbf{Ideal}^{\mathsf{SE}}_{\mathcal{L}_1,\mathcal{L}_2}(\kappa) = 1]| \leq \mathsf{neg}(\kappa)\,,$$

where experiments $\textbf{Real}^{\mathsf{SE}}(\kappa)$ *and* $\textbf{Ideal}^{\mathsf{SE}}_{\mathcal{L}_1,\mathcal{L}_2}(\kappa)$ *are defined in Figure 1 and where the randomness is taken over the random bits used by the algorithms of the SE scheme, the algorithms of the simulator and* Adv.

Leakage Functions $\mathcal{L}_1$ and $\mathcal{L}_2$. As in previous work [5], $\mathcal{L}_1$ and $\mathcal{L}_2$ are stateful leakage functions such that $\mathcal{L}_1(\mathcal{D}_0) = |\mathcal{D}_0| = N$ and $\mathcal{L}_2(w_i)$ leaks the *size* of the access pattern

```
$bit \leftarrow \textbf{Real}^{\mathsf{ORAM}}(\kappa)$:
1: $\mathsf{M}_0 \leftarrow \mathsf{Adv}(1^\kappa)$;
2: $\langle \sigma_0, \mathsf{EM}_0 \rangle \leftrightarrow \textsc{OramInitialize}\langle (1^\kappa, \mathsf{M}_0), 1^\kappa \rangle$;
3: for $k = 1$ to $q$ do
4:     $i_k \leftarrow \mathsf{Adv}(1^\kappa, \mathsf{EM}_0, m_1, \ldots, m_{k-1})$;
5:     $\langle (v_{i_k}, \sigma_k), \mathsf{EM}_k \rangle \leftrightarrow \textsc{OramAccess}\langle (\sigma_{k-1}, i_k), \mathsf{EM}_{k-1} \rangle$;
6:     Let $m_k$ be the messages from client to server in the OramAccess protocol above;
7: $bit \leftarrow \mathsf{Adv}(1^k, \mathsf{EM}_0, m_1, m_2, \ldots, m_q)$;
8: return $bit$;
$bit \leftarrow \textbf{Ideal}^{\mathsf{ORAM}}(\kappa)$:
1: $\mathsf{M}_0 \leftarrow \mathsf{Adv}(1^\kappa)$;
2: $(st_{\mathcal{S}}, \mathsf{EM}_0) \leftarrow \textsc{SimOramInitialize}(1^\kappa, |\mathsf{M}_0|)$;
3: for $k = 1$ to $q$ do
4:     $(st_{\mathcal{S}}, \mathsf{EM}_k, m_k) \leftarrow \textsc{SimOramAccess}(st_{\mathcal{S}}, \mathsf{EM}_{k-1})$;
5: $bit \leftarrow \mathsf{Adv}(1^k, \mathsf{EM}_0, m_1, m_2, \ldots, m_q)$;
6: return $bit$;
```

Fig. 2. Real and ideal experiments for the ORAM scheme.

$|\mathcal{D}(w_i)|$ and the *search pattern* of w_i. Formally for any keyword w_i that is searched at time i, $\mathcal{L}_2(w_i)$ is defined as

$$\mathcal{L}_2(w_i) = \begin{cases} (|\mathcal{D}(w_i)|, j) & \text{if } w_i \text{ was searched at time } j < i \\ (|\mathcal{D}(w_i)|, \perp) & \text{if } w_i \text{ was never searched before} \end{cases} . \tag{1}$$

2.2 Oblivious RAM

We recall *Oblivious RAM* (ORAM), a notion introduced and first studied by Goldreich and Ostrovsky [12]. ORAM can be thought of as a compiler that encodes the memory into a special format such that accesses on the compiled memory do not reveal the underlying access patterns on the original memory. An ORAM scheme consists of protocols $(\textsc{OramInitialize}, \textsc{OramAccess})$. We give the definition for an ORAM that supports only reads since this is what we need in our application—the definitions naturally extends for writes as well.

- $\langle \sigma, \mathsf{EM} \rangle \leftrightarrow \textsc{OramInitialize}\langle (1^\kappa, \mathsf{M}), 1^\kappa \rangle$: OramInitialize takes as input the security parameter κ and a memory array M of n indexed values $(1, v_1), \ldots, (n, v_n)$ of λ bits each and outputs a secret state σ (for the client), and an encrypted memory EM (for the server).
- $\langle (v_i, \sigma'), \mathsf{EM}' \rangle \leftrightarrow \textsc{OramAccess}\langle (\sigma, i), \mathsf{EM} \rangle$: OramAccess is a protocol between the client and the server, where the client's input is the secret state σ and an index i. Server's input is the encrypted memory EM. Client's output is the value v_i that was assigned to i and an updated secret state σ' and the server's output is an updated encrypted memory EM'.

```
(chosen, alternative) ← OfflineTwoChoiceAllocation($m, n$)
1: Let $\{1,\ldots,m\}$ be a set of balls and $\{1,\ldots,n\}$ be a set of bins;
2: Initialize A and B to be empty arrays of $m$ entries;
3: for $i = 1,\ldots,m$ do
4:     Pick two bins $a_i$ and $b_i$ from $\{1,\ldots,n\}$ independently and uniformly at random;
5:     A[$i$] $= a_i$;
6:     B[$i$] $= b_i$;
7: (chosen, alternative) ← MaxFlowSchedule($m, n$, A, B);
8: return (chosen, alternative);
```

Fig. 3. Offline two-choice allocation of m balls to n bins.

Definition 2 (Security of ORAM). *Assume* (ORAMINITIALIZE, ORAMACCESS) *is an ORAM scheme. The ORAM scheme is secure if for any PPT adversary* Adv*, there exists a stateful PPT simulator* (SIMORAMINITIALIZE, SIMORAMACCESS) *such that*

$$|\Pr[\mathbf{Real}^{\mathsf{ORAM}}(\kappa) = 1] - \Pr[\mathbf{Ideal}^{\mathsf{ORAM}}(\kappa) = 1]| \leq \mathsf{neg}(\kappa)\,,$$

where experiments $\mathbf{Real}^{\mathsf{ORAM}}(\kappa)$ *and* $\mathbf{Ideal}^{\mathsf{ORAM}}(\kappa)$ *are defined in Figure 2 and where the randomness is taken over the random bits used by the algorithms of the ORAM scheme, the algorithms of the simulator and* Adv.

3 New Bounds for Offline Two-Choice Allocation

As mentioned in the introduction, our central allocation algorithm uses a variation of the classic balls-in-bins problem, known as *offline two-choice allocation*—see Figure 3. Assume m balls and n bins. In the selection phase, for the i-th ball, two bins a_i and b_i are chosen independently and uniformly at random. After selection, in a post-processing phase, the i-th ball is mapped to either bin a_i or b_i such that the maximum load is minimized. This assignment is achieved by a maximum flow algorithm [21] (for completeness we also give this algorithm in Figure 13 in the Appendix). The bin that ball i is finally mapped to is stored in an array $\mathsf{chosen}[i]$ whereas the other bin that was chosen for ball i is stored in an array $\mathsf{alternative}[i]$. Let L^*_{max} denote the maximum load across all bins after this allocation process completes. Sanders et al. [21] proved the following:

Lemma 1 (Sanders et al. [21]). *Algorithm* OfflineTwoChoiceAllocation *in Figure 3 outputs an allocation* chosen *of m balls to n bins such that $L^*_{\mathsf{max}} > \lceil \frac{m}{n} \rceil + 1$ with probability at most $O(1/n)$*[6]*. Moreover, the allocation can be performed in time $O(n^3)$.*

For our purposes, the bounds derived by Sanders et al. [21] do not suffice. In the following we derive new bounds. In particular:

1. We derive probability bounds on the *number of overflowing bins*, namely the bins that contain more than $\lceil \frac{m}{n} \rceil + 1$ balls after OfflineTwoChoiceAllocation returns—see Section 3.1;

[6] Sanders et al. [21] gave a better bound $O(1/n)^{\lceil \frac{m}{n} \rceil + 1}$ which is $O(1/n)$ since $\lceil m/n \rceil \geq 0$. Our analysis is simplified when we take this looser bound $O(1/n)$.

2. We derive probability bounds on the *overflow size*, namely the number of balls beyond $\lceil \frac{m}{n} \rceil + 1$ that the overflowing bins contain—see Section 3.2.

Then we combine the above to derive bounds on the total number of overflowing balls—see Section 3.3.

3.1 Bounding the Number of Overflowing Bins

For every bin $\ell \in [n]$, let us define a random 0-1 variable Z_ℓ such that Z_ℓ is 1 if bin ℓ contains more than $\lceil \frac{m}{n} \rceil + 1$ balls after $\mathsf{OfflineTwoChoiceAllocation}$ returns and 0 otherwise. What we want to bound is the random variable

$$Z = \sum_{\ell=1}^{n} Z_i \,,$$

which represents total number of overflowing bins. Unfortunately we cannot use a Chernoff bound directly, since (i) the variables Z_i are not independent; (ii) we do not know the exact expectation $\mathbb{E}[Z]$ of Z. Fortunately, we observe that if we show that the variables Z_i are *negatively associated* (at a high level negative association indicates that for a given set of variables, whenever some of them increase the rest tend to decrease) and if we can derive an *upper bound* on the expectation of Z we can use a Chernoff-like bound that we prove in Lemma 9 in the Appendix. We first recall the definition of negative association:

Definition 3 (Dubhashi and Ranjan [11]). *A set of random variables $\{X_1, \ldots, X_n\}$ is negatively associated if for every two disjoint index sets $I \in [n]$ and $J \subseteq [n]$ it is*

$$\mathbb{E}[f(X_i, i \in I) g(X_j, j \in J)] \leq \mathbb{E}[f(X_i, i \in I)]\mathbb{E}[g(X_j, j \in J)]$$

for all $f : \mathbb{R}^{|I|} \to \mathbb{R}$, $g : \mathbb{R}^{|J|} \to \mathbb{R}$ that are both non-increasing or non-decreasing[7].

We now prove the following.

Lemma 2. *The set of random variables $Z_1, Z_2, \ldots, Z_n$ are negatively associated.*

Proof. Let X_{ijk} for all $i \in [n]$, $j \in [n]$ and $k \in [m]$ be the random variable defined as

$$X_{ijk} = \begin{cases} 1 & \text{if } \mathsf{OfflineTwoChoiceAllocation} \text{ chose the two bins } i \text{ and } j \text{ for ball } k \\ 0 & \text{otherwise} \end{cases} .$$

For each k it holds that $\sum_{i,j} X_{ijk} = 1$, since only one pair of bins are chosen for ball k. Therefore, by [11, Proposition 11], it follows that each set

$$\mathbf{X}_k = \{X_{ijk}\}_{i \in [n], j \in [n]}$$

[7] A function $h : \mathbb{R}^k \to \mathbb{R}$ is non-decreasing when $h(\mathbf{x}) \leq h(\mathbf{y})$ whenever $\mathbf{x} \leq \mathbf{y}$ in the component-wise ordering on $\mathbb{R}^k$.

is negatively associated. Moreover, since the sets $\mathbf{X}_k, \mathbf{X}_{k'}$ for $k \neq k'$ consist of mutually independent variables (as the selection of the two bins is made independently for each ball), it follows from [11, Proposition 7.1] that the entire set

$$\mathbf{X} = \{X_{ijk}\}_{i\in[n], j\in[n], k\in[m]}$$

is negatively associated. Now consider the disjoint sets U_ℓ for $\ell \in [n]$ where U_ℓ is defined as

$$U_\ell = \{X_{ijk} \mid \mathsf{chosen}[k] = \ell \ \wedge \ (\ell = i \vee \ell = j)\} \,,$$

where chosen is the array output by $\mathsf{OfflineTwoChoiceAllocation}$. Let now

$$h_\ell(X_{ijk}, X_{ijk} \in U_\ell) = \sum_{X_{ijk}\in U_\ell} X_{ijk}$$

for $\ell \in [n]$. Clearly each h_ℓ is a non-decreasing function and therefore by [11, Proposition 7.2] it follows that the set of random variables $\mathbf{Y} = \{Y_\ell\}_{\ell\in[n]}$ where $Y_\ell = h_\ell$ is also negatively associated. We can finally define Z_ℓ for $\ell = 1, \ldots, n$ as

$$Z_\ell = f(Y_\ell) = \begin{cases} 0 & \text{if } Y_\ell \leq \lceil m/n \rceil + 1 \\ 1 & \text{otherwise} \end{cases} .$$

Since f is also a non-decreasing function (as whenever Y_ℓ grows, $Z_\ell = f(Y_\ell)$ may only increase) therefore, again by [11, Proposition 7.2], it follows that the set of random variables $Z_1, Z_2, \ldots, Z_n$ is also negatively associated. □

Lemma 3. *The expected number of overflowing bins* $\mathbb{E}[Z]$ *is* $O(1)$.

Proof. For all bins $\ell \in [n]$, it is

$$\mathbb{E}[Z_\ell] = \Pr[L^*_{\mathsf{max}} > \lceil m/n \rceil + 1] \leq \Pr[Y_q > \lceil m/n \rceil + 1] = O(1/n) \,,$$

by Lemma 1. By linearity of expectation and since $Z = \sum Z_i$, it is $\mathbb{E}[Z] = O(1)$. □

Theorem 1. *Assume* $\mathsf{OfflineTwoChoiceAllocation}$ *from Figure 3 is used to allocate* m *balls into* n *bins. Let* Z *be the number of bins that receive more than* $\lceil m/n \rceil + 1$ *balls. Then there exists a fixed positive constant* c *such that for sufficiently large* n[8] *and for any* $\tau > 1$ *it is*

$$\Pr[Z \geq c \cdot \tau] \leq \left(\frac{e}{\tau}\right)^{c\cdot\tau} .$$

Proof. By Lemma 3 we have that there exists a fixed constant c such that $\mathbb{E}[Z] \leq c$ for sufficiently large n. Therefore, by Lemma 2 and Lemma 9 in the Appendix (where we set $\mu_H = c$ since $\mathbb{E}[Z] \leq c$) we have that for any $\delta > 0$

$$\Pr[Z \geq (1+\delta)\cdot c] \leq \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^c \leq \left(\frac{e^{1+\delta}}{(1+\delta)^{(1+\delta)}}\right)^c .$$

Setting $\delta = \tau - 1$ which is > 0 for $\tau > 1$, we get the desired result. □

[8] This means that there exists a fixed constant n_0 such that for $n \geq n_0$ the statement holds—we provide a quick estimate of the constants c and n_0 in Appendix E—$c = 36$ and $n_0 = 655$—but we believe the constants can be improved with a more thorough analysis.

3.2 Bounding the Overflow Size

Next, we turn our attention to the number of balls Y_ℓ that can be assigned to bin ℓ. In particular we want to derive a probability bound $\Pr[Y_\ell > \lceil m/n \rceil + \tau]$ defined in general for parameter $\tau \geq 2$—Sanders et al. [21] studied only the case where $\tau = 1$. To do that, we will bound the probability that after OfflineTwoChoiceAllocation returns the maximum load L^*_{max} is larger than $\lceil m/n \rceil + \tau$ for $\tau \geq 2$. We now prove the following result.

Theorem 2. *Assume* OfflineTwoChoiceAllocation *from Figure 3 is used to allocate m balls into n bins. Let L^*_{max} be the maximum load across all bins. Then for any $\tau \geq 2$*

$$\Pr\left[L^*_{\mathsf{max}} \geq \left\lceil \frac{m}{n} \right\rceil + \tau\right] \leq O(1/n)^\tau + O(\sqrt{n} \cdot 0.9^n)\,.$$

Proof. Our analysis here closely follows the one of [21]. Without loss of generality, we assume the number of balls m to be a multiple of the number of bins n[9] and we will set $b = m/n$. Let now (a_i, b_i) be the two random choices that OfflineTwoChoiceAllocation makes for ball i where $i = 1, \ldots, m$. For a subset $U \subseteq \{1, \ldots, n\}$ of bins we define the random variables $X_1^U, \ldots, X_m^U$ such that

$$X_i^U = \begin{cases} 1 & \text{if } a_i \in U \text{ and } b_i \in U \\ 0 & \text{otherwise} \end{cases},$$

i.e., X_i^U is 1 if both selections for the i-th ball are from subset U, which unavoidably leads to this ball being assigned to a bin within subset U. The random variable $L_U = \sum_{i=1}^m X_i^U$ is called the *unavoidable load* of U. Also, for a set U and a parameter τ, let $P_U = \Pr[L_U \geq (b+\tau)|U| + 1]$. Finally let L^*_{max} be the *optimal load*, namely the minimum maximum load that can be derived by considering all possible allocations *given* the random choices $(a_1, b_1), \ldots, (a_m, b_m)$. Since MaxFlowSchedule computes an allocation with the optimal load, we must compute the probability $\Pr[L^*_{\mathsf{max}} > b+\tau]$, where $\tau \geq 2$. From [22, Lemma 5] we have that

$$L^*_{\mathsf{max}} = \max_{\emptyset \neq U \subseteq \{1,\ldots,n\}} \left\{ \frac{L_U}{|U|} \right\}.$$

Thus we can write

$$\begin{aligned} \Pr[L^*_{\mathsf{max}} > b+\tau] &= \Pr[\exists U \subseteq [n] : L_U/|U| > b+\tau] \\ &= \Pr[\exists U \subseteq [n] : L_U \geq (b+\tau)|U| + 1] \\ &\leq \textstyle\sum_{\emptyset \neq U \subseteq [n]} \Pr[L_U \geq (b+\tau)|U| + 1] = \sum_{|U|=1}^n \binom{n}{|U|} P_U\,, \end{aligned}$$

[9] If not, we pad to $m = n\lceil m'/n \rceil$ balls, where m' is the original number of balls. Then, to get an allocation for the m' balls, we get an allocation for the m balls and we remove the balls that we do not need. Clearly, if L^* is the optimal maximum load for the m' balls, it is the case that $L^* \leq L^*_{\mathsf{max}}$ (if $L^* > L^*_{\mathsf{max}}$ you can get a better allocation for the m' balls through the allocation of the m balls, a contradiction) and therefore whatever probability bounds we derive for L^*_{max} hold for L^*.

where the inequality follows from a simple union bound and for the last step we used the fact that P_U is the same for all sets U of the same cardinality. This is because for all sets U_1 and U_2 with $|U_1| = |U_2|$ we have that $\Pr[L_{U_1} \geq (b+\tau)|U_1| + 1] = \Pr[L_{U_2} \geq (b+\tau)|U_2| + 1]$ since U_1 and U_2 are identically distributed.

Next, we need to bound the sum $\sum_{|U|=1}^{n} \binom{n}{|U|} P_U$. For this we will split the sum into three separate summands

$$T_1 = \sum_{1 \leq |U| \leq \frac{n}{8}} \binom{n}{|U|} P_U,\ T_2 = \sum_{\frac{n}{8} < |U| < \frac{nb}{b+\tau}} \binom{n}{|U|} P_U \text{ and } T_3 = \sum_{\frac{nb}{b+\tau} \leq |U| \leq n} \binom{n}{|U|} P_U.$$

We begin with the simple observation that $T_3 = 0$. To see why, note that for $|U| \geq nb/(b+\tau)$ it holds that $P_U = \Pr[L_U \geq (b+\tau)|U| + 1] = \Pr[L_U \geq (b+\tau)nb/(b+\tau) + 1] = \Pr[L_U \geq m+1] = 0$ as m is a natural upper bound for L_U (i.e., if both selections fall within U for all balls). Regarding T_2, we argue as follows. First, from [21, Lemma 9] we have that

$$\sum_{\frac{n}{8} < |U| < \frac{nb}{b+1}} \binom{n}{|U|} P_U^* = O(\sqrt{n} \cdot 0.9^n)\,,$$

where $P_U^* = \Pr[L_U \geq (b+1)|U| + 1]$. Clearly, for all U, $P_U \leq P_U^*$. Moreover, $\sum_{\frac{n}{8} < |U| < \frac{nb}{b+\tau}} P_U^* \leq \sum_{\frac{n}{8} < |U| < \frac{nb}{b+1}} P_U^*$ for all $\tau \geq 2$. Putting it all together it follows that

$$T_2 \leq \sum_{\frac{n}{8} < |U| < \frac{nb}{b+1}} \binom{n}{|U|} P_U^* = O(\sqrt{n} \cdot 0.9^n).$$

By Lemma 10 in the Appendix, we get $T_1 = O(1/n)^{b+\tau} = O(1/n)^{b+\tau}$ for all $\tau \geq 2$ and therefore for all $\tau \geq 2$ it is $\sum_{|U|=1}^{n} \binom{n}{|U|} P_U = O(1/n)^{\tau}$ since $b \geq 0$. This completes the proof. □

3.3 Bounding the Total Number of Overflowing Balls

Let $T > 0$ be the number of overflowing balls, i.e., $T = \sum_{i=1}^{\ell} Z_i(Y_i - \lceil m/n \rceil - 1)$. We now have the following:

Theorem 3. *Assume* OfflineTwoChoiceAllocation *from Figure 3 is used to allocate m balls into n bins. Let T be the number of overflowing balls as defined above. Then there exist fixed positive constants c, c_1, c_2 such that for sufficiently large n and for any $\tau \geq 2$ it is*

$$\Pr[T > c \cdot \tau^2] \leq \left(\frac{e}{\tau}\right)^{c\cdot\tau} + \left(\frac{c_1}{n}\right)^{\tau} + c_2\sqrt{n} \cdot 0.9^n\,.$$

Proof. Define the events $E : T > c \cdot \tau^2$, $E_1 : Z > \tau$ and $E_2 : L^*_{\mathsf{max}} > \lceil m/n \rceil + \tau$, for some $\tau \geq 2$. By the law of total probability

$$\begin{aligned}\Pr[E] &= \Pr[E|E_1 \wedge E_2]\Pr[E_1 \wedge E_2] + \Pr[E|E_1 \wedge E_2']\Pr[E_1 \wedge E_2'] \\ &\quad \Pr[E|E_1' \wedge E_2]\Pr[E_1' \wedge E_2] + \Pr[E|E_1' \wedge E_2']\Pr[E_1' \wedge E_2'] \\ &\leq \Pr[E_1]\Pr[E_2] + \Pr[E_1]\Pr[E_2'] + \Pr[E_1']\Pr[E_2] + 0\,,\end{aligned}$$

where $\Pr[E|E_1' \wedge E_2'] \Pr[E_1' \wedge E_2'] = 0$ since the probability $\Pr[E|E_1' \wedge E_2'] = 0$. This is because there is no way there can be more than τ^2 overflowing balls given both the number of overflowing bins and the maximum overflow per bin is at most τ. Therefore by Theorem 1 and Theorem 2 we have

$$\Pr[E] \leq \Pr[E_1] + 2\Pr[E_2] \leq \left(\frac{e}{\tau}\right)^{c\cdot\tau} + O(1/n)^{\tau} + O(\sqrt{n} \cdot 0.9^n)\,,$$

which completes the proof by taking c_1 and c_2 to be the constants in the $O()$ notations $O(1/n)$ and $O(\sqrt{n} \cdot 0.9^n)$ respectively. □

4 New Oblivious RAM with $O(1)$ Locality and $o(\sqrt{n})$ Bandwidth

Our constant-locality SE construction requires the use of an Oblivious RAM (ORAM) scheme as a black box. As we explained in the introduction, the ORAM scheme that will be used must have the following properties:

1. It needs to have constant locality, meaning that for each oblivious access it should only read $O(1)$ non-contiguous locations in the encrypted memory. Existing ORAM constructions with polylogarithmic bandwidth have *logarithmic* locality. For example, a path ORAM access [26] traverses $\log n$ binary tree nodes stored in non-contiguous memory locations—therefore we cannot use it here.
   This property is required as our underlying SE scheme must have $O(1)$ locality;
2. It needs to have bandwidth cost $o(\sqrt{n} \cdot \lambda)$. Note that this is less than the bandwidth of the square-root ORAM by Goldreich and Ostrovsky [12] (which also has $O(1)$ locality if an I/O-efficient oblivious sorting algorithm [14] for the reshuffling is used) but we will use the square-root ORAM as our starting point.
   This property is required because we would be applying the ORAM scheme on an array of $O(\log^2 N)$ entries, yielding ovelall bandwidth equal to $o(\log N \cdot \lambda)$, which would imply sublogarithmic read efficiency for the underlying SE scheme.

We note here that an existing scheme that seems to be satisfying both properties above is the ORAM construction based on a randomized shuffling algorithm from [20, Theorem 7] by Ohrimenko et al. (where we set $c = 3$). This ORAM has $O(1)$ locality and $O(n^{1/3} \log n \cdot \lambda)$ bandwidth. However we cannot apply it here due to its failure probability which is $\mathsf{neg}(n)$, where n is the size of the memory array. Unfortunately, since our array has $O(\log^2 N)$ entries (where N is the size of the SE dataset), this would give a probability of failure $\mathsf{neg}(\log^2 N)$ which is not necessarily $\mathsf{neg}(N)$.

4.1 Achieving $O(n^{1/3} \log^2 n \cdot \lambda)$ amortized bandwidth and $O(1)$ amortized locality

In the following we summarize our amortized construction first—see Figure 4, and then we present our final de-amortized ORAM construction in Figure 16 in the Appendix that is derived by using standard techniques by Goodrich et al. [15]. To simplify the exposition, we consider just read-only ORAM since this is what we need for our SE scheme. Our ORAM algorithms can be easily extended for writes as well.

ORAM Setup. Given memory M storing n index-value pairs $(1, v_1), (2, v_2), \ldots, (n, v_n)$ we allocate three main arrays for storage: A of size $n_a = n + n^{2/3}$, B of size $n_b = n^{2/3} + n^{1/3}$, and C of size $n_c = n^{1/3}$. Initially A stores all elements encrypted with a CPA-secure encryption scheme and permuted with a pseudorandom permutation[10] $\pi_a : [n_a] \rightarrow [n_a]$ and B and C are empty, containing encryptions of dummy values.

We also initialize another pseudorandom permutation $\pi_b : [n_b] \rightarrow [n_b]$ used for accessing elements from array B. In particular, if an element $x \in [n]$ is stored in array B, it is located at position $\pi_b[\mathsf{Tab}[x]]$ of B, where Tab is a hash table that is stored locally and maps an element $x \in [n]$ to an index $\mathsf{Tab}[x] \in [n_b]$. Note that the hash table is necessary to index elements in B since $n_b < n$.

ORAM Access. To access element x, the algorithm always downloads, decrypts and sequentially scans array C. Similarly to the square-root ORAM, we consider two cases:

1. *Element x is in C.* In this case the requested element has been found and the algorithm performs two additional dummy accesses for security reasons: it accesses a random[11] position in array A and a random position in array B.
2. *Element x is not in C.* In this case we distinguish the following subcases.
   - Element x is not in B.[12] In this case x can be retrieved by accessing the random position $\pi_a[x]$ of array A. Like previously, the algorithm also accesses a random position in array B.
   - Element x is in B. In this case x can be retrieved by accessing the random position $\pi_b[\mathsf{Tab}[x]]$ of array B. Like previously, the algorithm also accesses a random position in array A.

In the end of the access, the retrieved element x is being written in the next available position of C, the algorithm computes a fresh encryption of C and writes C encrypted back to the server. Also, element x is being recorded as an accessed element in array $\mathsf{SCRATCH}$—$\mathsf{SCRATCH}$ will be the input to a small reshuffling, see below.

Reshuffling, epochs and superepochs. Our algorithm for obliviously accessing an element x described above proceeds in *epochs* and *superepochs*. An epoch is defined as a sequence of $n^{1/3}$ accesses. A superepoch is defined as a sequence of $n^{2/3}$ accesses.

At the end of every epoch C becomes full, and all elements in C along with the ones that have been accessed in the current superepoch (and are now stored in B) are obliviously reshuffled into B using a fresh pseudorandom permutation π_b. In our implementation in Figure 4, we store all the elements that must be reshuffled in an array $\mathsf{SCRATCH}$. After the reshuffling C can be emptied (denoted with $\perp$ Line 30) so that it can be used again in the future.

At the end of every superepoch all the elements of the dataset are obliviously reshuffled into array A using a fresh pseudorandom permutation π_a and arrays B, C and $\mathsf{SCRATCH}$ are emptied.

[10] In practice (and since we will be applying our ORAM on a "small domain" of $O(\log^2 N)$ elements), π_a will be implemented with efficient small-domain pseudorandom permutations (e.g., [16,25,19]).

[11] As in the square-root ORAM, the position chosen is not entirely random—it is chosen from the ones that have not been chosen so far.

[12] This can be decided by checking whether $\mathsf{Tab}[x]$ is null or not.

```
Protocol $\langle \sigma, \mathsf{EM} \rangle \leftrightarrow$ ORAMINITIALIZE$\langle (1^\kappa, \mathsf{M}), \perp \rangle$:
 1: Parse $\mathsf{M}$ as $(1, v_1), (2, v_2), \ldots, (n, v_n)$ where $|i, v_i| = \lambda$ (the values are $\lambda$ bits long);
 2: Let $n_a \leftarrow n + n^{2/3}$, $n_b \leftarrow n^{2/3} + n^{1/3}$, $n_c \leftarrow n^{1/3}$;
 3: Let $A$, $B$ and $C$ be arrays of size $n_a$, $n_b$ and $n_c$ respectively. Initialize them with 0 entries;
 4: Let SCRATCH be an array of size $n_b$. Initialize it with 0 entries;
 5: Let $\pi_a : [n_a] \rightarrow [n_a]$ and $\pi_b : [n_b] \rightarrow [n_b]$ be pseudorandom permutations;
 6: For $i = 1, \ldots, n$, store $(i, v_i)$ at location $\pi_a[i]$ in $A$;
 7: Encrypt-And-Write arrays $A$, $B$, $C$ and SCRATCH and add them to EM ;
 8: Let $\mathsf{count}_a \leftarrow 0$ and $\mathsf{count}_b \leftarrow 0$;
 9: Let Tab be an empty hash table;
10: Set $\sigma = (\pi_a, \pi_b, \mathsf{Tab}, \mathsf{count}_a, \mathsf{count}_b)$;
11: return $\langle \sigma, \mathsf{EM} \rangle$;
Protocol $\langle (v_i, \sigma'), \mathsf{EM}' \rangle \leftrightarrow$ ORAMACCESS$\langle (\sigma, i), \mathsf{EM} \rangle$:
 1: Parse $\sigma$ as $(\pi_a, \pi_b, \mathsf{Tab}, \mathsf{count}_a, \mathsf{count}_b)$ and EM as $(A, B, C, \mathsf{SCRATCH})$;
 2: Increment $\mathsf{count}_a$ and $\mathsf{count}_b$;
 3: Read-And-Decrypt array $C$;
 4: if $(i, v_i) \in C$ then                    ▷ $(i, v_i)$ was accessed before and is stored in $C$
 5:     $\mathsf{index}_a \leftarrow \pi_a[n + \mathsf{count}_a]$;
 6:     $\mathsf{index}_b \leftarrow \pi_b[n^{2/3} + \mathsf{count}_b]$;
 7: else
 8:     if $\mathsf{Tab}[i] \neq \mathsf{null}$ then                    ▷ $(i, v_i)$ is stored in $B[\mathsf{index}_b]$
 9:         $\mathsf{index}_a \leftarrow \pi_a[n + \mathsf{count}_a]$;
10:         $\mathsf{index}_b \leftarrow \pi_b[\mathsf{Tab}[i]]$;
11:     else                    ▷ $(i, v_i)$ is stored in $A[\mathsf{index}_a]$
12:         $\mathsf{index}_a \leftarrow \pi_a[i]$;
13:         $\mathsf{index}_b \leftarrow \pi_b[n^{2/3} + \mathsf{count}_b]$;
14: Read-And-Decrypt $A[\mathsf{index}_a]$;
15: Read-And-Decrypt $B[\mathsf{index}_b]$;
16: Retrieve $(i, v_i)$ from either $A[\mathsf{index}_a]$ or $B[\mathsf{index}_b]$ or $C$;
17: $C[\mathsf{count}_b] \leftarrow (i, v_i)$;
18: Encrypt-And-Write array $C$;
19: $\mathsf{Tab}[i] \leftarrow \mathsf{count}_a$;
20: Encrypt-And-Write element $(\mathsf{Tab}[i], v_i)$ at position $\mathsf{count}_a$ of array SCRATCH;
21: if $\mathsf{count}_a > n^{2/3}$ then                    ▷ Transition to a new superepoch
22:     Let $\pi_a$ and $\pi_b$ be new pseudorandom permutations;
23:     $\mathsf{count}_a \leftarrow 0$ and $\mathsf{count}_b \leftarrow 0$;
24:     $\langle \perp, A \rangle \leftrightarrow$ OBLIVIOUSSORTING$\langle (\pi_a, n_a, n^{1/3} \log^2 n), A \rangle$;                    ▷ large rebuild
25:     Set $B \leftarrow \perp$; $C \leftarrow \perp$; $\mathsf{SCRATCH} \leftarrow \perp$; Set $\mathsf{Tab} \leftarrow \perp$;
26: if $\mathsf{count}_b > n^{1/3}$ then                    ▷ Transition to a new epoch
27:     Let $\pi_b$ be new pseudorandom permutation;
28:     $\mathsf{count}_b \leftarrow 0$;
29:     $\langle \perp, B \rangle \leftrightarrow$ OBLIVIOUSSORTING$\langle (\pi_b, n_b, n^{1/3} \log^2 n), \mathsf{SCRATCH} \rangle$;                    ▷ small rebuild
30:     Set $C \leftarrow \perp$;
31: return $\langle (v_i, (\pi_a, \pi_b, \mathsf{Tab}, \mathsf{count}_a, \mathsf{count}_b)), (A, B, C, \mathsf{SCRATCH}) \rangle$;
```

Fig. 4. Read-only ORAM construction with $O(n^{1/3} \log^2 n \cdot \lambda)$ amortized bandwidth and $O(1)$ amortized locality.

Oblivious Sorting With Good Locality. As in previous works, our reshuffling in the ORAM protocol is performed using an oblivious sorting protocol. Since we are using the ORAM scheme in an SE scheme that must have good locality, we must ensure that the oblivious sorting protocol used has good locality as well, i.e., it does not access too many non-contiguous locations. One way to achieve that is to download the whole encrypted array, decrypt it, sort it and encrypt it back. This has excellent locality $L = 1$ but requires linear client space. A standard oblivious sorting protocol such as Batcher's odd-even mergesort [6] does not work either since its locality can be linear.

Fortunately, Goodrich and Mitchenmacher [14] developed an oblivious sorting protocol for an external memory setting that is a perfect fit for our application—see the pseudocode of the protocol in Figure 15 in the Appendix. The client interacts with the server only by reading and writing b consecutive blocks of memory. We call each b-block access (either for read or write) an *I/O operation*. The performance of their protocol is characterized in the following theorem.

Theorem 4 (Goodrich and Mitchenmacher [14], Goodrich [13]). *Given an array X containing n comparable blocks, we can sort X with a data-oblivious external-memory protocol that uses $O((n/b)\log^2(n/b)$ I/O operations and local memory of $4b$ blocks, where an I/O operation is defined as the read/write of b consecutive blocks of X.*

Note that in the above oblivious sorting protocol one can parametrize the block size, affecting the local space accordingly. In our case, we set the block size to be equal to $n^{1/3}\log^2 n$—see Lines 24 and 29 in Figure 4. This will be enough for achieving constant (amortized) locality in our SE scheme.

The correctness and security proofs of our ORAM scheme can be found in Appendix C.

Lemma 4. *The ORAM scheme of Figure 4 has $O(n^{1/3}\log^2 n \cdot \lambda)$ amortized bandwidth per access and $O(1)$ amortized locality per access and the client space is $O(n^{2/3}\log n + n^{1/3}\log^2 n \cdot \lambda)$.*

Proof. Over the course of n accesses, each access $1 \le i \le n$ incurs the following:

- $O(n^{1/3}\cdot\lambda)$ bandwidth and $O(1)$ locality due to access of A, B, C and $\mathsf{SCRATCH}$;
- $O(n^{2/3}\log^2 n \cdot \lambda)$ bandwidth and $O(n^{1/3})$ locality due to the small rebuilding which happens only when $i \bmod n^{1/3} = 0$ (i.e., $n^{2/3}$ times);
- $O(n\log^2 n\cdot\lambda)$ bandwidth and $O(n^{2/3})$ locality due to the large rebuilding which happens only when $i \bmod n^{2/3} = 0$ (i.e., $n^{1/3}$ times).

Note that in order to derive the locality of the rebuilding above, we used Theorem 4 for $b = n^{1/3}\log^2 n$. Now, the amortized bandwidth is

$$\lambda \cdot \frac{n\cdot O(n^{1/3}) + n^{2/3}\cdot O(n^{2/3}\log^2 n) + n^{1/3}\cdot O(n\log^2 n)}{n} = O(n^{1/3}\log^2 n\cdot\lambda)$$

and the amortized locality is

$$\frac{n\cdot O(1) + n^{2/3}\cdot O(n^{1/3}) + n^{1/3}\cdot O(n^{2/3})}{n} = O(1)\,.$$

Finally, the client must store Tab locally, that consists of $n^{2/3}$ entries of $\log n$ bits each and also needs to have $O(n^{1/3} \log^2 n \cdot \lambda)$ space locally for the oblivious sorting—see Theorem 4. □

4.2 From Amortized to Worst-Case Bandwidth and Locality

To turn the amortized version of our scheme into worst-case we must perform some reshuffling work (in particular $c \cdot n^{1/3} \log^2 n$ work where c is an appropriate constant) during every regular ORAM access so that by the time $n^{1/3}$ or $n^{2/3}$ accesses have been performed, the small or large reshuffling respectively is complete and therefore there is no need for an expensive massive reshuffling. That idea was described for the square root ORAM by Goodrich et al. [15]. The details are given in Appendix D. We now have the following results.

Theorem 5. *Let n is the size of the memory array and λ be the size of the block. The ORAM scheme of Figure 16 (i) is correct according to Definition 5; (ii) is secure according to Definition 2 and assuming pseudorandom permutations and CPA-secure encryption; (iii) has $O(n^{1/3} \log^2 n \cdot \lambda)$ worst-case bandwidth and $O(1)$ worst-case locality per access and requires client space $O(n^{2/3} \log n + n^{1/3} \log^2 n \cdot \lambda)$.*

Further Reducing the Local Space for Big Block Sizes. We now observe that for big block sizes that are $\Omega(n^{1/3})$ bits (as is the case for our application), the space required to store the hash table ($n^{2/3} \log n$) becomes asymptotically less or equal to the local space required to do the oblivious sorting ($n^{1/3} \log^2 n \cdot \lambda$), and therefore the total local space becomes $O(n^{1/3} \log^2 n \cdot \lambda)$. Therefore we have the following.

Corollary 1. *Let $\lambda = \Omega(n^{1/3})$ bits be the block size. Then the ORAM scheme of Figure 16 has $O(n^{1/3} \log^2 n \cdot \lambda)$ worst-case bandwidth per access, $O(1)$ worst-case locality per access and $O(n^{1/3} \log^2 n \cdot \lambda)$ client space.*

5 Allocation Algorithms

As we mentioned in the introduction, to construct our final SE scheme we are going to use a series of *allocation algorithms*. The goal of an allocation algorithm for an SE dataset $\mathcal{D}$ consisting of q keyword lists $\mathcal{D}(w_1), \mathcal{D}(w_2), \ldots, \mathcal{D}(w_q)$ is to store/allocate the elements of all lists into an array $\mathbf{A}$ (or multiple arrays).

Retrieval Instructions. To be useful, an allocation algorithm should also output a hash table Tab such that $\mathsf{Tab}[w]$ contains "instructions" on how to correctly retrieve a keyword list $\mathcal{D}(w)$ after the list is stored. For example, for a keyword list $\mathcal{D}(w)$ that contains four elements stored at positions $5, 16, 26, 27$ of $\mathbf{A}$ by the allocation algorithm, some valid alternatives for the instructions $\mathsf{Tab}[w]$ are: (i) *"access positions $5, 16, 26, 27$ of array $\mathbf{A}$"*; (ii) *"access all positions from 3 to 28 of array $\mathbf{A}$"*; (iii) *"access the whole array $\mathbf{A}$"*. Clearly, there are different tradeoffs among the above instructions, as we discuss in the following.

Independence Property. For security purposes, and in particular for being able to simulate the search procedure of the SE scheme, it is important the instructions $\mathsf{Tab}[w]$ output by an allocation algorithm for a specific keyword list $\mathcal{D}(w)$ be *independent* of the distribution of the rest of the underlying SE dataset—intuitively this implies that accessing $\mathcal{D}(w)$ as instructed by the allocation algorithm does not reveal any information about the rest of the dataset.

Two Examples. The aforementioned independence property is easy to achieve, for example, with the following naive allocation "read-all" algorithm: Allocate $\mathcal{D}(w_1)$ in the first $|\mathcal{D}(w_1)|$ positions of array $\mathbf{A}$, $\mathcal{D}(w_2)$ in the next $|\mathcal{D}(w_2)|$ positions of array $\mathbf{A}$ and so on. Then, for all keywords w in $\mathcal{D}$, output $\mathsf{Tab}[w] \leftarrow$ *"access the whole array* $\mathbf{A}$*"* as the instruction. Clearly $\mathsf{Tab}[w]$ is independent of the distribution of the dataset. However, even accessing a small list (e.g., consisting of just one element) requires reading the whole array $\mathbf{A}$ (N elements) which is very inefficient. Note that our final construction uses this algorithm for keyword list that are very large, for which reading the whole array is not an issue.

We can improve the above allocation algorithm by applying a random permutation π in the array $\mathbf{A}$, storing element $\mathbf{A}[i]$ in position $\mathbf{A}[\pi[i]]$—this is actually the allocation algorithm used by most existing SE schemes, e.g., [9]. In that case, the allocation algorithm sets $\mathsf{Tab}[w]$ to contain the positions assigned by the random permutation π to the elements of $\mathcal{D}(w)$. Due to the random permutation π, instructions $\mathsf{Tab}[w]$ are independent from the distribution of the rest of the dataset (any allocation is equally likely) and at the same time retrieving any list requires accessing exactly $|\mathcal{D}(w)|$ locations. While this "permute" algorithm seems ideal, it has a major shortcoming: it requires $|\mathcal{D}(w)|$ random jumps in the memory—this can lead to impractical protocols when array $\mathbf{A}$ is so large that needs to be stored on disk.

Our Approach. In this section, we wish to develop allocation algorithms that satisfy the independence property that are only required to jump to a constant number of locations to retrieve the result (just like "read-all" did), while minimizing the additional entries that are being retrieved (just like "permute" did). As we said before, our final allocation algorithm that will be used by our proposed SE scheme consists of four different allocation algorithms that work for specific size ranges of keyword lists. First we begin with some terminology.

5.1 Buckets and Superbuckets

Following terminology from [5], our allocation algorithms use fixed-capacity *buckets* for storage. A bucket with capacity C can store up to C *elements*—in our case an *element* is a keyword-document pair (w, id). To simplify notation, we represent a set of B buckets $A_1, A_2, \ldots, A_B$ as an array $\mathbf{A}$ of B buckets, referring to bucket A_i as $\mathbf{A}[i]$. Additionally, a *superbucket* $\mathbf{A}\{k, s\}$ is a set of the following s consecutive buckets

$$\mathbf{A}[(k-1)s+1], \mathbf{A}[(k-1)s+2], \ldots, \mathbf{A}[ks]\,.$$

```
Algorithm (A, Tab) ← AllocateSmall(D, N): (taken from [5])
1: Let max ← $N^{1-1/\log\log N}$, $C = c_s \cdot \log\log N \log^2 \log\log N$[a] and $B \leftarrow N/C$;
2: Let $\mathbf{A}$ be an array of $B$ buckets—each bucket has capacity $C$;
3: Initialize an empty hash table Tab;
4: for sizes $s = \mathsf{max}, \mathsf{max}/2, \mathsf{max}/4, \ldots, 1$ do
5:     for each keyword $w$ such that $|\mathcal{D}(w)| = s$ do
6:         Pick $\alpha$ and $\beta$ from $\{1, \ldots, \frac{B}{s}\}$ independently and uniformly at random;
7:         Let $\mathbf{A}\{\alpha, s\}$ and $\mathbf{A}\{\beta, s\}$ be two superbuckets;
8:         Let $x \in \{\alpha, \beta\}$ correspond to the superbucket with the minimum load;
9:         Store $\mathcal{D}(w)$ horizontally into superbucket $\mathbf{A}\{x, s\}$;
10:        $\mathsf{Tab}[w] = (s, \alpha, \beta, \perp)$;
11: if there is a bucket $\mathbf{A}[i]$ that overflows then
12:     return FAIL;
13: else
14:     Pad every bucket $\mathbf{A}[i]$ to $C$ elements using dummy values;
15:     return (A, Tab);
```

[a] Constant c_s can be appropriately chosen in [5].

Fig. 5. Allocation algorithm for small sizes from Asharov et al. [5].

We say that we store a keyword list $\mathcal{D}(w) = \{(w, id_1), (w, id_2), \ldots, (w, id_s)\}$ *horizontally* into superbucket $\mathbf{A}\{k, s\}$ when each element (w, id_i) is stored in a separate bucket of the superbucket.[13]

Finally, the *load* of a bucket or a superbucket is the number of elements stored in each bucket or superbucket.

5.2 Allocating Small Keyword Lists with Two-Dimensional Balanced Allocation

We refer to keyword lists that have size at most $N^{1-1/\log\log N}$ as *small*. For those keyword lists we use the two-dimensional allocation algorithm of Asharov et al. [5]. For completeness we provide the algorithm in Figure 5, which we call AllocateSmall. Let $C = c_s \cdot \log\log N \log^2 \log\log N$, for some appropriately chosen constant c_s [5]. The algorithm uses $B = N/C$ buckets of capacity C each. It then considers all small keyword lists starting from the largest to the smallest, and depending on the list's size s, it picks two superbuckets from $\{1, 2 \ldots, B/s\}$ uniformly at random, horizontally placing the keyword list into the superbucket with the minimum load. The algorithm records the chosen superbucket, as well as the other chosen superbucket as instructions in a hash table Tab. If, during this allocation process some bucket overflows, then the algorithm outputs FAIL. We now have the following result.

[13] For example consider an array $\mathbf{A}$ consisting of 20 buckets $\mathbf{A}[1], \mathbf{A}[2] \ldots, \mathbf{A}[20]$ where each bucket $\mathbf{A}[i]$ has capacity $C = 5$. Superbucket $\mathbf{A}\{3, 4\}$ contains the buckets $\mathbf{A}[9], \ldots, \mathbf{A}[12]$. Horizontally storing $\{a_1, a_2, \ldots, a_4\}$ into $\mathbf{A}\{3, 4\}$ means storing a_1 into $\mathbf{A}[9]$, a_2 into $\mathbf{A}[10]$, and so on, reducing the capacity of $\mathbf{A}[9], \ldots, \mathbf{A}[12]$ from 5 to 4, and increasing their load from 0 to 1.

Theorem 6 (Asharov et al. [5]). *Algorithm* AllocateSmall *in Figure 5 outputs* FAIL *with probability* $\mathsf{neg}(N)$*. Moreover the output array of buckets* $\mathbf{A}$ *occupies space* $O(N)$*.*

Note that a list of size s can be read by accessing s consecutive buckets (i.e., a superbucket), therefore the read efficiency for these lists is $O(\log\log N \log^2 \log\log N)$.

5.3 Allocating Medium Keyword Lists with Offline Two-Choice Allocation

We refer to keyword lists with sizes greater than $\mathsf{min} = N^{1-1/\log\log N}$ and at most $\mathsf{max} = N/\log^2 N$ as *medium*. The allocation algorithm for these lists is shown in Figure 6. The algorithm uses an array $\mathbf{A}$ of $B = N\log\log N/\log N$ buckets, where each bucket has capacity $C = 3 \cdot \log N/\log\log N$. Just like AllocateSmall, the allocation algorithm for medium sizes stores a list $\mathcal{D}(w)$ of size s horizontally into one of the superbuckets

$$\mathbf{A}\{1,s\}, \mathbf{A}\{2,s\}, \ldots, \mathbf{A}\{B/s,s\}\,.$$

However, unlike AllocateSmall, the supebucket that is finally chosen to store $\mathcal{D}(w)$ depends only on keyword lists of the *same* size with $\mathcal{D}(w)$ that have already been allocated and not on all other keyword lists encountered so far. In particular, let k_s be the number of keyword lists that have size s. Let also $b_s = B/s$ be the number of superbuckets with respect to size s. To figure out which superbucket to pick for horizontally storing a particular keyword list of size s, the algorithm views the k_s keyword lists as *balls* and the b_s superbuckets as *bins* and performs an offline two-choice allocation of k_s keyword lists (balls) into b_s superbuckets (bins), as described in Section 3. When, during this process some superbucket contains

$$\left\lceil \frac{k_s}{b_s} \right\rceil + 1$$

keyword lists of size s, any subsequent keyword list of size s meant for this superbucket is instead placed into a stash $\mathbf{B}_s$ that contains exactly $c \cdot \log^2 N$ buckets of size s each for some fixed constant c derived in Theorem 1. Our algorithm will fail, if

– Some bucket $\mathbf{A}[i]$ overflows (i.e., the total number of elements that are eventually stored into $\mathbf{A}[i]$ exceeds its capacity C), which as we show in Lemma 5 never happens; or
– More than $c \cdot \log^2 N$ keyword lists of size s must be stored at some stash $\mathbf{B}_s$, which as we show in Lemma 6 happens with negligible probability.

All the choices that the algorithm makes, such as the two superbuckets originally chosen for every list during the offline two-choice allocation as well as the position in the stash (in case the list was an overflowing one) are recorded in Tab as retrieval instructions. We now prove the following lemma.

Lemma 5. *During the execution of algorithm* AllocateMedium *in Figure 6, no bucket* $\mathbf{A}[i]$ *(for all* $i = 1, \ldots, B$*) will ever overflow.*

```
Algorithm $(\mathbf{A}, \mathcal{B}, \mathsf{Tab}) \leftarrow \mathsf{AllocateMedium}(\mathcal{D}, N)$:
 1: Let $\mathsf{min} \leftarrow N^{1-\frac{1}{\log\log N}}$, $\mathsf{max} \leftarrow \frac{N}{\log^2 N}$, $C \leftarrow 3\cdot\frac{\log N}{\log\log N}$, $B \leftarrow N/C$ and $\ell = c\cdot\log^2 N$;[a]
 2: Let $\mathbf{A}$ be an array of $B$ buckets—each bucket has capacity $C$;
 3: Initialize an empty hash table $\mathsf{Tab}$;
 4: for sizes $s = 2\mathsf{min}, 4\mathsf{min}, \ldots, \mathsf{max}$ do
 5:     Let $\mathbf{B}_s$ be an array of $\ell$ buckets—each bucket has capacity $s$;     ▷ This is the stash
 6:     $i \leftarrow 0$;
 7:     Let $k_s$ be the number of keywords in $\mathcal{D}$ with $|\mathcal{D}(w)| = s$;
 8:     Let $b_s \leftarrow B/s$ be the number of superbuckets with respect to size $s$;
 9:     Let $\mathsf{inStash}_s \leftarrow 0$;
10:     $(\mathsf{chosen}, \mathsf{alternative}) \leftarrow \mathsf{OfflineTwoChoiceAllocation}(k_s, b_s)$;
11:     for each keyword $w$ such that $|\mathcal{D}(w)| = s$ do
12:         Increment $i$;
13:         Set $\alpha \leftarrow \mathsf{chosen}[i]$;
14:         Set $\beta \leftarrow \mathsf{alternative}[i]$;
15:         if superbucket $\mathbf{A}\{\alpha, s\}$ contains $\leq \lceil \frac{k_s}{b_s} \rceil$ keyword lists of size $s$ then
16:             Store $\mathcal{D}(w)$ horizontally into superbucket $\mathbf{A}\{\alpha, s\}$;
17:             $\mathsf{Tab}[w] = (s, \alpha, \beta, 1)$;
18:         else                                                  ▷ Move to stash
19:             Increment $\mathsf{inStash}_s$;
20:             if $\mathsf{inStash}_s > \ell$ then                     ▷ Stash overflows
21:                 return FAIL;
22:             Store $\mathcal{D}(w)$ in the bucket $\mathbf{B}_s[\mathsf{inStash}_s]$;
23:             $\mathsf{Tab}[w] = (s, \alpha, \beta, \mathsf{inStash}_s)$;
24: if there is a bucket $\mathbf{A}[i]$ that has overflown then
25:     return FAIL;
26: else
27:     Pad every bucket $\mathbf{A}[i]$ to $C$ elements using dummy values;
28: return $(\mathbf{A}, (\mathbf{B}_{2\mathsf{min}}, \mathbf{B}_{4\mathsf{min}}, \mathbf{B}_{8\mathsf{min}}, \mathbf{B}_{16\mathsf{min}} \ldots, \mathbf{B}_{\mathsf{max}}), \mathsf{Tab})$;
```

[a] Constant c is derived by Theorem 1.

Fig. 6. Allocation algorithm for medium sizes.

Proof. For each size $s = 2\mathsf{min}, 4\mathsf{min}, \ldots, \mathsf{max}$, Line 15 of AllocateMedium allows at most $\lceil k_s/b_s \rceil + 1$ keyword lists of size s to be stored in any superbucket $\mathbf{A}\{i, s\}$. Since every keyword list of size s is stored horizontally in a superbucket $\mathbf{A}\{i, s\}$, it follows that every bucket $\mathbf{A}[i]$ within every superbucket $\mathbf{A}\{i, s\}$ will have load, due to keywords lists of size s, at most $s \cdot (\lceil k_s/b_s \rceil + 1)/s = \lceil k_s/b_s \rceil + 1$. Therefore the total load of a bucket $\mathsf{A}[i]$ due to all sizes $s = 2\mathsf{min}, 4\mathsf{min}, \ldots, \mathsf{max}$ is at most

$$\sum_s \left(\left\lceil \frac{k_s}{b_s} \right\rceil + 1 \right) \leq \sum_s \frac{k_s}{b_s} + \sum_s 2\,.$$

We now bound the above sums separately. Since $b_s = B/s$, $\sum_s k_s \cdot s \le N$ and $B = N \log\log N / \log N$ it is

$$\sum_s \frac{k_s}{b_s} = \frac{1}{B} \sum_s k_s \cdot s \le \frac{N}{B} = \frac{\log N}{\log\log N}\,.$$

Now since $\mathsf{min} = 2 \cdot N^{1-1/\log\log N} = 2^{\log N - \log N/\log\log N + 1}$, $\mathsf{max} = N/\log^2 N = 2^{\log N - 2\log\log N}$ and size s takes only powers of 2, there are $\frac{\log N}{\log\log N} - 2\log\log N$ terms in the sum $\sum_s 2$ and therefore

$$\sum_s \left(\left\lceil \frac{k_s}{b_s} \right\rceil + 1 \right) \le 3 \cdot \frac{\log N}{\log\log N} - 4 \cdot \log\log N \le 3 \cdot \frac{\log N}{\log\log N}\,,$$

which is equal to the capacity C of the bucket in $\mathsf{AllocateMedium}$. Therefore no bucket will ever overflow. $\square$

Lemma 6. *During the execution of algorithm* $\mathsf{AllocateMedium}$ *in Figure 6, no stash* $\mathbf{B}_s$ *(for* $s = 2\mathsf{min}, 4\mathsf{min}, \ldots, \mathsf{max}$*) will ever overflow, except with probability* $\mathsf{neg}(N)$.

Proof. Recall that for each $s = 2\mathsf{min}, 4\mathsf{min}, \ldots, \mathsf{max}$, placing the k_s keyword lists of size s into the b_s superbuckets of size s is performed via an offline two-choice allocation of k_s balls into b_s bins. Also recall that the lists that end up in the stash $\mathbf{B}_s$ (that has capacity $\log^2 N$) are originally placed by the allocation algorithm in superbuckets containing more than $\lceil k_s/b_s \rceil + 1$ keyword lists of size s, thus they are *overflowing*. Let T_s be the number of these lists. By Theorem 9, where we set $T = T_s$ and $n = b_s$ and $\tau = \log N$, we have that for large b_s and for fixed constants c, c_1 and c_2

$$\Pr[T_s > c \cdot \log^2 N] \le \left(\frac{e}{\log N}\right)^{c \cdot \log N} + \left(\frac{c_1}{b_s}\right)^{\log N} + c_2 \sqrt{b_s} \cdot 0.9^{b_s} = \mathsf{neg}(N)\,,$$

since $b_s = B/s = N \log\log N / s \log N \ge \log N \log\log N$ as $s \le \mathsf{max} = N/\log^2 N$. $\square$

Theorem 7. *Algorithm* $\mathsf{AllocateMedium}$ *in Figure 6 outputs* FAIL *with probability* $\mathsf{neg}(N)$. *Moreover, the size of the output array* $\mathbf{A}$ *and the stashes* $\mathcal{B}$ *is* $O(N)$.

Proof. $\mathsf{AllocateMedium}$ can fail either because a bucket $\mathbf{A}[i]$ overflows, which by Lemma 5 happens with probability 0, or because some stash $\mathbf{B}_s$ ends up having to store more than $\log^2 N$ elements for some $s = 2\mathsf{min}, 4\mathsf{min}, \ldots, \mathsf{max}$, which by Lemma 6 happens with probability $\mathsf{neg}(N)$.

For the space complexity, since no bucket $\mathbf{A}[i]$ overflows, array $\mathbf{A}$ occupies space $O(N)$. Also each stash B_s contains $\log^2 N$ buckets of size s each so the total size required by the stashes is

$$c \cdot \log^2 N(\mathsf{min} + 2\mathsf{min} + 4\mathsf{min} + \ldots + \mathsf{max}) \le 2 \cdot c \cdot \log^2 N \cdot \mathsf{max} = O(N)\,,$$

since $\mathsf{max} = N/\log^2 N$. $\square$

```
Algorithm $(A, \mathcal{B}, \mathsf{Tab}) \leftarrow \mathsf{AllocateLarge}(\mathcal{D}, N, \mathsf{min}, \mathsf{max})$:
1: Initialize an empty hash table $\mathsf{Tab}$;
2: Let $\mathbf{A}$ be an array of $t = N/\mathsf{max}$ buckets—each bucket has capacity $2\mathsf{max}$;
3: for sizes $s = 2\mathsf{min}, 4\mathsf{min}, 8\mathsf{min}, 16\mathsf{min} \ldots, \mathsf{max}$ do
4:     Let $\mathbf{B}_s$ be and array of $N/\mathsf{max}$ buckets—each bucket has capacity $s$;
5:     $\mathsf{inStash}_s \leftarrow 0$;
6:     $\mathsf{available}_s \leftarrow \{1, 2, \ldots, t\}$;
7: for each keyword $w$ such that $\mathsf{min} < |\mathcal{D}(w)| \leq \mathsf{max}$ do
8:     $s \leftarrow |\mathcal{D}(w)|$;
9:     Pick $k \in \mathsf{available}_s$ uniformly at random;
10:    Set $\mathsf{available}_s \leftarrow \mathsf{available}_s - \{k\}$;
11:    if bucket $\mathbf{A}[k]$ has at least $s$ available space then
12:        Store $\mathcal{D}(w)$ in bucket $\mathbf{A}[k]$;
13:        $\mathsf{Tab}[w] \leftarrow (s, k, \perp, 1)$;
14:    else
15:        Increment $\mathsf{inStash}_s$;
16:        if $\mathsf{inStash} \geq N/\mathsf{max}$ then
17:            return $\mathsf{FAIL}$;
18:        Store $\mathcal{D}(w)$ in bucket $B_s[\mathsf{inStash}_s]$;
19:        $\mathsf{Tab}[w] \leftarrow (s, k, \perp, \mathsf{inStash}_s)$;
20: return $(A, (\mathbf{B}_{2\mathsf{min}}, \mathbf{B}_{4\mathsf{min}}, \mathbf{B}_{8\mathsf{min}}, \mathbf{B}_{16\mathsf{min}} \ldots, \mathbf{B}_{\mathsf{max}}), \mathsf{Tab})$;
```

Fig. 7. Allocation algorithm for large sizes.

5.4 Allocating Large Keyword Lists

We call a keyword list large, if its size is in the range $N/\log^2 N$ and $N/\log^{2/3} N$. Algorithm $\mathsf{AllocateLarge}$ in Figure 7 is used to allocate lists whose size falls within a specific subrange $(\mathsf{min}, \mathsf{max}]$ of the above range. Our actual construction (see Figure 9) will use two such consecutive subranges—that is why we pass min and max as arguments in $\mathsf{AllocateLarge}$.

For a given subrange $(\mathsf{min}, \mathsf{max}]$, $\mathsf{AllocateLarge}$ stores all keyword lists in either an array $\mathbf{A}$ of $t = N/\mathsf{max}$ buckets of capacity $2\mathsf{max}$ each or in some stash $\mathbf{B}_s$ that contains N/max buckets of capacity s each and which is kept for each size s. In particular, for a large keyword list $\mathcal{D}(w)$ of size s, the algorithm picks a bucket $\mathbf{A}[i]$ uniformly at random and tries to place the list in this bucket. If there is no space in $\mathbf{A}[i]$, it places the list in the next available position in $\mathbf{B}_s$. The algorithm records both the chosen bucket for keyword w as well as the stash position, irrespective of whether $\mathcal{D}(w)$ is placed in the bucket $\mathbf{A}[i]$ or not (as we will see later, for security purposes, both of these will be accessed when w is searched for). Now, when another keyword of size s must be allocated, the algorithm picks from a set of buckets $\mathsf{available}_s$ that does not contain i (and in general all previous choices made for lists of size s), and repeats the same procedure. The formal description of the algorithm is shown in Figure 7. We now prove that if we follow this procedure, our stashes will never overflow.

Theorem 8. *Algorithm* $\mathsf{AllocateLarge}$ *in Figure 7 never outputs* FAIL*. Moreover, the size of the output array* $\mathbf{A}$ *and the stashes* $\mathcal{B}$ *is* $O(N)$*.*

```
Algorithm $(A, \mathsf{Tab}) \leftarrow \mathsf{AllocateHuge}(\mathcal{D}, N)$:
1: Let $\mathsf{min} \leftarrow N/\log^{2/3} N$;
2: Initialize an empty hash table $\mathsf{Tab}$;
3: Let $\mathbf{A}$ be an array of $N$ entries;
4: $\mathsf{count} \leftarrow 1$;
5: for all keywords $w$ such that $|\mathcal{D}(w)| > \mathsf{min}$ do
6:     Store $\mathcal{D}(w)$ in positions $\mathsf{count}, \mathsf{count}+1, \ldots, \mathsf{count}+|\mathcal{D}(w)|-1$ of array $\mathbf{A}$;
7:     $\mathsf{count} \leftarrow \mathsf{count} + |\mathcal{D}(w)|$;
8:     $\mathsf{Tab}[w] \leftarrow (|\mathcal{D}(w)|, \perp, \perp, \perp)$;
9: return $(\mathbf{A}, \mathsf{Tab})$;
```

Fig. 8. Allocation algorithm for huge sizes.

Proof. Assume $\mathsf{AllocateLarge}$ fails. Then for some $s = \mathsf{max}, \mathsf{max}/2, \ldots, \mathsf{min}$ it has to be the case that N/max pieces of size s are placed in the stash $\mathbf{B}_s$. A piece of size s is placed in the stash because a bucket of size $2\mathsf{max}$ in array $\mathbf{A}$ has occupancy at least $2\mathsf{max} - s + 1$. Therefore, by the time the overflow happens N/max different buckets in array $\mathbf{A}$ must have occupancy at least $2\mathsf{max} - s + 1$ (this is because $\mathsf{AllocateLarge}$ does not probe buckets that have been probed before for pieces of the same size). Therefore the number of entries that should have been considered up to that point is

$$\frac{N}{\mathsf{max}}(2\mathsf{max} - s + 1) \geq \frac{N}{\mathsf{max}}(\mathsf{max} + 1) \geq N + \frac{N}{\mathsf{max}} \geq N + \log^{2/3} N\,,$$

since $s \leq \mathsf{max} \leq N/\log^{2/3} N$. This is a contradiction, however, since the number of entries of our dataset is exactly N. For the space complexity, since the output array $\mathbf{A}$ has at most N/max buckets of size $2\mathsf{max}$ each, it follows that $\mathbf{A}$ has size $O(N)$. Finally, since each stash B_s has N/max buckets of size s each, the total size is

$$\frac{N}{\mathsf{max}}(\mathsf{max} + \mathsf{max}/2 + \mathsf{max}/4 + \ldots + \mathsf{min}) = O(N)\,.$$

□

5.5 Allocating Huge Keyword Lists with a Read-All Algorithm

Keyword lists that have size from $N/\log^{2/3} N$ up to N are stored directly in an array $\mathbf{A}$ of N entries, one after the other—see Figure 8. To read a huge list in our actual construction, one would have to read the whole array $\mathbf{A}$—however, due to the huge size of the list, the read efficiency would still be small.

6 Our SE Construction

We now present our main construction that uses the ORAM scheme presented in Section 4 and the allocation algorithms presented in Section 5 as black boxes. Our formal protocols are shown in Figure 9 and Figure 10.

```
Protocol $\langle st, \mathcal{I}\rangle \leftrightarrow \textsc{Setup}\langle(1^\kappa, \mathcal{D}), \perp\rangle$:
1: Let $N \leftarrow \sum_{w\in\mathbf{W}} |\mathcal{D}(w)|$;
2: Let $\mathsf{Tab}$ be an empty hash table of capacity $N$;
3: $(\mathbf{S}, \mathsf{Tab}_S) \leftarrow \mathsf{AllocateSmall}(\mathcal{D}, N)$;
4: for all buckets $\mathbf{S}[i] \in \mathbf{S}$ do
5:     Encrypt-And-Write bucket $\mathbf{S}[i]$ and add encrypted $\mathbf{S}[i]$ to server index $\mathcal{I}$;
6: $(\mathbf{M}, \mathbf{B}_M, \mathsf{Tab}_M) \leftarrow \mathsf{AllocateMedium}(\mathcal{D}, N)$;
7: for all buckets $\mathbf{M}[i] \in \mathbf{M}$ do
8:     Encrypt-And-Write bucket $\mathbf{M}[i]$ and add encrypted $\mathbf{M}[i]$ to server index $\mathcal{I}$;
9: $(L, \mathbf{B}_L, \mathsf{Tab}_L) \leftarrow \mathsf{AllocateLarge}(\mathcal{D}, N, N/\log^2 N, N/\log^{4/3} N)$;
10: for all buckets $L[i] \in L$ do
11:     Encrypt-And-Write bucket $L[i]$ and add encrypted $L[i]$ to server index $\mathcal{I}$;
12: $(\mathcal{L}, \mathbf{B}_\mathcal{L}, \mathsf{Tab}_\mathcal{L}) \leftarrow \mathsf{AllocateLarge}(\mathcal{D}, N, N/\log^{4/3} N, N/\log^{2/3} N)$;
13: for all buckets $\mathcal{L}[i] \in \mathcal{L}$ do
14:     Encrypt-And-Write bucket $\mathcal{L}[i]$ and add encrypted $\mathcal{L}[i]$ to server index $\mathcal{I}$;
15: $(\mathbf{H}, \mathsf{Tab}_H) \leftarrow \mathsf{AllocateHuge}(\mathcal{D}, N)$;
16: Encrypt-And-Write array $\mathbf{H}$ and add encrypted $\mathbf{H}$ to server index $\mathcal{I}$;
17: Set $\mathsf{Tab} \leftarrow \mathsf{Tab}_S \cup \mathsf{Tab}_M \cup \mathsf{Tab}_L \cup \mathsf{Tab}_\mathcal{L} \cup \mathsf{Tab}_H$;
18: $st \leftarrow \mathsf{Tab}$;
19: for every stash $\mathbf{B}_s \in \mathbf{B}_M \cup \mathbf{B}_L \cup \mathbf{B}_\mathcal{L}$ corresponding to size $s$ do
20:     $\langle \sigma_s, \mathsf{EM}_s\rangle \leftrightarrow \textsc{OramInitialize}\langle(1^\kappa, \mathbf{B}_s), \perp\rangle$;
21:     Encrypt-And-Write $\sigma_s$ and add $\sigma_s$ and $\mathsf{EM}_s$ to server index $\mathcal{I}$;
22: if AllocateSmall or AllocateMedium or AllocateLarge called above output FAIL then
23:     return FAIL;
24: return $\langle st, \mathcal{I}\rangle$;
```

Fig. 9. The setup protocol of our SE construction.

6.1 Setup Protocol of SE scheme

Our setup algorithm allocates keyword lists depending on whether they are small, medium, large or huge, as defined in Section 5. We describe the details below.

Small Keyword Lists. These are allocated to superbuckets using $\mathsf{AllocateSmall}$ from Section 5.2. The allocation algorithm outputs an array of buckets $\mathbf{S}$ storing the small keyword lists and the instructions hash table Tab_S storing, for each small keyword list $\mathcal{D}(w)$, its size s and the superbuckets α and β assigned for this keyword list by the allocation algorithm. The setup protocol of the SE scheme finally encrypts and writes bucket array $\mathbf{S}$ and stores it remotely—see Line 5 in Figure 9. It stores Tab_S locally.

Medium Keyword Lists. These are allocated to superbuckets using $\mathsf{AllocateMedium}$ from Section 5.3. $\mathsf{AllocateMedium}$ outputs (i) an array of buckets $\mathbf{M}$; (b) the set of stashes $\{\mathbf{B}_s\}_s$ that handle the overflows, for all sizes s in the range; (iii) the instructions hash table Tab_M storing, for each keyword list $\mathcal{D}(w)$ that falls into this range, its size s, the superbuckets α and β assigned for this keyword list and a stash position x in the stash $\mathbf{B}_s$ where the specific keyword list could have been potentially stored, had it

```
Protocol $\langle(\mathcal{D}(w), st'), \mathcal{I}'\rangle \leftrightarrow$ SEARCH$\langle(st, w), \mathcal{I}\rangle$:
 1: Parse $st$ as Tab and $\mathcal{I}$ as $(\mathbf{S}, \mathbf{M}, L, \mathcal{L}, \mathbf{H}, \{\sigma_s, \mathsf{EM}_s\})$;
 2: Let $(s, \alpha, \beta, x) \leftarrow \mathsf{Tab}[w]$;
 3: if $s > N/\log^{2/3} N$ then                                  ▷ Huge sizes
 4:     Read-And-Decrypt array $\mathbf{H}$;
 5:     Retrieve $\mathcal{D}(w)$ from $\mathbf{H}$;
 6: if $s \le N^{1-1/\log\log N}$ then                            ▷ Small sizes
 7:     Read-And-Decrypt superbuckets $\mathbf{S}\{\alpha, s\}$ and $\mathbf{S}\{\beta, s\}$;
 8:     Retrieve $\mathcal{D}(w)$ from $\mathbf{S}\{\alpha, s\}$ and $\mathbf{S}\{\beta, s\}$;
 9: else
10:     Read-And-Decrypt $\sigma_s$;
11:     $\langle(v_x, \sigma_s), \mathsf{EM}_s\rangle \leftrightarrow$ ORAMACCESS$\langle(\sigma_s, x), \mathsf{EM}_s\rangle$;
12:     Encrypt-And-Write $\sigma_s$;
13:     if $N^{1-1/\log\log N} < s \le N/\log^2 N$ then                ▷ Medium sizes
14:         Read-And-Decrypt superbuckets $\mathbf{M}\{\alpha, s\}$ and $\mathbf{M}\{\beta, s\}$;
15:         Retrieve $\mathcal{D}(w)$ from $\mathbf{M}\{\alpha, s\}$ and $\mathbf{M}\{\beta, s\}$ or $v_x$;
16:     if $N/\log^2 N < s \le N/\log^{4/3} N$ then             ▷ Large sizes (scale 1)
17:         Read-And-Decrypt bucket $L[\alpha]$;
18:         Retrieve $\mathcal{D}(w)$ from $L[\alpha]$ or $v_x$;
19:     if $N/\log^{4/3} N < s \le N/\log^{2/3}$ then             ▷ Large sizes (scale 2)
20:         Read-And-Decrypt bucket $\mathcal{L}[\alpha]$;
21:         Retrieve $\mathcal{D}(w)$ from $\mathcal{L}[\alpha]$ or $v_x$;
22: return $\langle(\mathcal{D}(w), st), \mathcal{I}\rangle$;
```

Fig. 10. The search protocol of our SE construction.

caused an overflow (otherwise a dummy position is stored). The setup protocol finally encrypts and writes array $\mathbf{M}$ and stores it remotely—see Line 8 in Figure 9. It also builds an ORAM per stash $\mathbf{B}_s$—see Line 23 in Figure 9. Finally it stores Tab_M locally.

Large Keyword Lists. These are allocated to buckets (and not superbuckets) using AllocateLarge from Section 5.4. To keep read efficiency small, we run AllocateLarge for three distinct subranges, as we detailed in Section 5. Similarly to AllocateMedium, each execution of AllocateLarge outputs an array of buckets (either L or $\mathcal{L}$ depending on the subrange) a hash table and a set of stashes $\{\mathbf{B}_s\}_s$ that handle overflows for the sizes s in each subrange. The setup protocol finally encrypts and writes arrays L or $\mathcal{L}$ and stores them remotely and also builds an ORAM per stash B_s. Finally it stores Tab_L and $\mathsf{Tab}_{\mathcal{L}}$ locally.

Huge Keyword Lists. For these lists, we use AllocateHuge from Section 5.5. This algorithm outputs an array $\mathbf{H}$ and a hash table Tab_H. Our setup protocol encrypts and writes $\mathbf{H}$ remotely and stores Tab_H locally.

Local State and Using Tokens. For the sake of simplicity and readability of Figure 9, we assume that the client keeps locally the hash table Tab—see Line 20. This occupies linear space $O(N)$ but can be securely outsourced using standard SE techniques [24], and without affecting the efficiency (read efficiency and locality) of our scheme: For every hash table entry $w \to [s, \alpha, \beta, x]$, store at the server the "encrypted" hash table entry

$$t_w \to \mathsf{ENC}_{k_w}(s||\alpha||\beta||x)\,,$$

where t_w and k_w comprise the *tokens* for keyword w (these are the outputs of a PRF applied on w with two different secret keys that the client stores) and ENC is a CPA-secure encryption scheme. To search for keyword w, the client just needs to send to the server the tokens t_w and k_w and the server can then search the encrypted hash table and retrieve the information $s||\alpha||\beta||x$ by decrypting.

Handling ORAM State and Failures. Our setup protocol does not store locally the ORAM states σ_s of the stashes $\mathbf{B}_s$ for which we build an ORAM. Instead, it encrypts and writes them remotely and downloads them when needed—see Line 21 in Figure 9. Also, note that our setup algorithm can fail, whenever any of the allocation algorithms fail. By Theorems 6, 7 and 8 we have the following:

Lemma 7. *Protocol* SETUP *in Figure 9 fails with probability* $\mathsf{neg}(N)$.

Lemma 8. *Protocol* SETUP *in Figure 9 outputs an encrypted index $\mathcal{I}$ that has $O(N)$ size and runs in $O(N)$ time.*

Proof. The space complexity follows from Theorems 6, 7, 8, by the fact that array $\mathbf{H}$ output by $\mathsf{AllocateLarge}$ has size $O(N)$ and by the fact that the ORAM states σ_s, being asymptotically less than the ORAM themselves, clearly occupy at most linear space.

For the running time, note that $\mathsf{AllocateSmall}$, $\mathsf{AllocateLarge}$, $\mathsf{AllocateHuge}$ run in linear time and the ORAM setup algorithms also run in linear time (same analysis with the space can be made). By Lemma 1, $\mathsf{AllocateMedium}$ must perform a costly $O(n^3)$ offline allocation (a maximum flow computation) where n is the number of superbuckets defined for every size s in the range. The maximum number of superbuckets M is achieved for the smallest size handled by $\mathsf{AllocateMedium}$ and is equal to

$$M = \frac{N \log\log N}{N^{1-1/\log\log N} \cdot \log N} = N^{1/\log\log N} \log\log N / \log N\,.$$

Recall that there are at most $\log N / \log\log N$ sizes handled by $\mathsf{AllocateMedium}$ and therefore the time required to do the offline allocation is at most

$$O\left(\frac{\log N}{\log\log N} \cdot M^3\right) = O\left(N^{3/\log\log N} \log^3 \log N / \log^3 N\right) = O(N)\,.$$

Therefore the total running time is $O(N)$. □

6.2 Search Protocol of SE scheme

Given a keyword w, the client first retrieves information (s, α, β, x) from $\mathsf{Tab}[w]$. Depending on the size s of $\mathcal{D}(w)$ the client takes the following actions (see Figure 10):

- If the list $\mathcal{D}(w)$ is *small*, the client reads two superbuckets $\mathbf{S}\{\alpha, s\}$ and $\mathbf{S}\{\beta, s\}$ and decrypts them. Since the size of the buckets $\mathbf{S}[i]$ is $\log\log N \log^2 \log\log N$ and each superbucket contains s of them, it follows that the read efficiency for small sizes is $O(\log\log N \log^2 \log\log N)$. Note also that since only two superbuckets are read, the locality for small lists is $O(1)$.
- If the list $\mathcal{D}(w)$ is *medium*, the client reads two superbuckets $\mathbf{M}\{\alpha, s\}$ and $\mathbf{M}\{\beta, s\}$ and decrypts them. Also he performs an ORAM access in the stash $\mathbf{B}_s$ for location x. Since the size of the buckets $\mathbf{M}[i]$ is $O(\log N/\log\log N)$ and each superbucket has s of them, it follows that the read efficiency for medium sizes due to accessing array $\mathbf{M}$ is $O(\log N/\log\log N)$.
  For the ORAM access, note that in our case it is $n = c\cdot\log^2 N$. Therefore by Corollary 1, and since our block size is at least $N^{1-1/\log\log N}$ which is $\Omega(\log^{2/3} N)$, the bandwidth required is
  $$O(n^{1/3}\log^2 n \cdot s) = O(\log^{2/3} N \log^2 \log N \cdot s)$$
  and therefore the read efficiency due to the ORAM access is
  $$O(\log^{2/3} N \log^2 \log N) = o(\log N/\log\log N)\,.$$
  Therefore the overall read efficiency for medium sizes is $O(\log N/\log\log N)$. Again, since only two superbuckets are read and since the ORAM locality is $O(1)$ (Corollary 1), it follows that the locality for medium lists is $O(1)$.
- If the list $\mathcal{D}(w)$ is *large*, the client reads bucket α (of size $2\cdot\mathsf{max}$) from either arrays L, or $\mathcal{L}$ depending on the exact subrange (therefore the locality is $O(1)$), and also does an ORAM access for position x in the stash B_s. The read efficiency due to reading from arrays L or $\mathcal{L}$ is at most
  $$\frac{2\cdot\mathsf{max}}{\mathsf{min}} = O(\log^{2/3} N) = o(\log N/\log\log N)\,,$$
  independently of the subrange we are considering. Concerning the ORAM access, the ORAM is applied on an array of $N/\mathsf{max} \leq \log^{4/3} N$ buckets of size s each (since $\mathsf{max} \geq N/\log^{4/3} N$ for large sizes), and therefore the same analysis for the ORAM that we did above holds here as well.
- For huge sizes, the read efficiency is at most $O(\log^{2/3} N) = o(\log N/\log\log N)$ since the locality is constant since the whole array $\mathbf{H}$ is read.

Therefore, overall, the locality is $O(1)$, the read efficiency is $O(\log N/\log\log N)$ and the space required at the server is $O(N)$.

6.3 Security of our Construction

We now prove the security of our construction. For this, we build a simulator SIMSETUP and SIMSEARCH in Figures 11 and 12 respectively.

```
Algorithm $(st_{\mathcal{S}}, \mathcal{I}_0) \leftarrow \textsc{SimSetup}(1^\kappa, \mathcal{L}_1(\mathcal{D}_0))$:
1: Parse $\mathcal{L}_1(\mathcal{D}_0)$ as $N$;
2: Let $\mathbf{S}$ to be an array that contains $N$ dummy elements; Encrypt-And-Write $\mathbf{S}$;
3: Let $\mathbf{M}$ to be an array of $N$ dummy elements; Encrypt-And-Write $\mathbf{M}$;
4: Let $L$, $\mathsf{L}$ and $\mathcal{L}$ be arrays of $2 \cdot N$ dummy elements; Encrypt-And-Write $L$, $\mathsf{L}$ and $\mathcal{L}$;
5: Let $\mathbf{H}$ be an array of $N$ dummy elements; Encrypt-And-Write $\mathbf{H}$;
6: Let $\mathsf{min} = N^{1-1/\log\log N}$ and $\mathsf{max} = N/\log^2 N$;
7: for $s = 2\mathsf{min}, 4\mathsf{min}, 8\mathsf{min}, \ldots, \mathsf{max}$ do
8:     Set $\mathbf{B}_s$ to be an array of $c \cdot \log^2 N$ entries of $s$ dummy elements each;
9:     $(st_{\mathcal{S}}^s, \mathsf{EM}_s) \leftarrow \textsc{SimOramInitialize}(1^\kappa, |\mathbf{B}_s|)$;
10:    Parse $st_{\mathcal{S}}^s$ as $\sigma_s$; Encrypt-And-Write $\sigma_s$;
11: for $\mathsf{exp} = 2, 4/3$ do
12:    Let $\mathsf{min} = N/\log^{\mathsf{exp}} N$ and $\mathsf{max} = N/\log^{\mathsf{exp}-2/3} N$;
13:    for $s = 2\mathsf{min}, 4\mathsf{min}, 8\mathsf{min}, \ldots, \mathsf{max}$ do
14:        Set $\mathbf{B}_s$ to be an array of $N/\mathsf{max}$ entries of $s$ dummy elements each;
15:        Set $\mathsf{available}_s \leftarrow \{1, 2, \ldots, N/\mathsf{max}\}$;
16:        $(st_{\mathcal{S}}^s, \mathsf{EM}_s) \leftarrow \textsc{SimOramInitialize}(1^\kappa, |\mathbf{B}_s|)$;
17:        Parse $st_{\mathcal{S}}^s$ as $\sigma_s$; Encrypt-And-Write $\sigma_s$;
18: Let $\mathsf{messages}$ be an empty hash table;
19: Set $\mathcal{I}_0 = (\mathbf{S}, \mathbf{M}, L, \mathcal{L}, \mathbf{H}, \{\sigma_s, \mathsf{EM}_s\})$;
20: return $((N, \mathsf{Choices}, \{st_{\mathcal{S}}^s\}, \{\mathsf{available}_s\}), \mathcal{I}_0)$;
```

Fig. 11. The simulator of the setup protocol of our SE scheme.

Simulation of the Setup Protocol. To simulate the setup protocol, our simulator must output $\mathcal{I}_0$ by just using the leakage $\mathcal{L}_1(D_0) = N$. Our SIMSETUP algorithm outputs $\mathcal{I}_0$ as CPA-secure encryptions of arrays $(\mathbf{S}, \mathbf{M}, L, \mathcal{L}, \mathbf{H})$ that contain dummy values and have the same dimensions with the arrays of the actual setup algorithm. Also, it calls the ORAM simulator from Figure 14 and also outputs $\{\sigma_s, \mathsf{EM}_s\}$). Due to the security of the underlying ORAM scheme and the CPA-security of the underlying encryption scheme, the adversary cannot distinguish between the two outputs.

One potential problem, however, is the fact that SIMSETUP always succeeds while there is a chance that the setup algorithm can fail, which will enable the adversary to distinguish between the two. However, by Lemma 7, this happens with probability $\mathsf{neg}(N) = \mathsf{neg}(\kappa)$, as required by our security definition, Definition 1.

Simulation of the Search Protocol. The simulator of the SEARCH protocol is shown in Figure 12. For a keyword query w_k, the simulator takes as input the leakage $\mathcal{L}_2(w_k) = (s, b)$, as defined in Relation 1.

If the query on w_k was performed before (thus $b \neq \perp$), the simulator just outputs the previous messages M_b plus the messages that were output by the ORAM simulator.

If the query on w_k was not performed before, then the simulator can easily generate the messages M_k depending on the size s of the keyword list $\mathcal{D}(w_k)$. In particular note that all the accesses on the arrays $(\mathbf{S}, \mathbf{M}, L, \mathcal{L}, \mathbf{H})$ are independent of the dataset that is being considered and therefore can be easily simulated by repeating the same process with the real execution. We can now state our final theorem.

Theorem 9. *There exists a correct and secure SE scheme with linear space, constant locality and sublogarithmic read efficiency.*

7 Conclusions and Observations

First, as we showed, our construction is using ORAM as a black box and therefore one could wonder why not use ORAM from the very beginning and on the whole dataset. While ORAM can theoretically solve the SE problem with much better security guarantees, it is not a good fit when one must minimize the read efficiency. For example, to the best of our knowledge, there is no ORAM that we could have used on the whole dataset that would yield sublogarithmic read efficiency (irrespective of the locality).

Second, we note that Proposition 4.6 by Asharov et al. [5] states that one could not expect to construct an allocation algorithm where the square of the locality $\times$ the read efficiency is $O(\log N/ \log\log N)$. This is the case with our construction! The reason this proposition does not apply to our approach is because our allocation algorithm is using multiple structures for storage, e.g., stashes and multiple arrays, and therefore does not fall into the model used to prove the negative result.

Finally, we note that our read efficiency for large sizes can be improved, by increasing the space. For example, by splitting the range in three subranges

$$(N/\log^2 N, N/\log^{1.5} N], (N/\log^{1.5} N, N/\log N], (N/\log^{1.5} N, N/\log^{0.5} N]$$

(which means keeping more stashes and one more array), we can improve the read efficiency to $O(N/\log^{0.5} N)$ only for large sizes. Similarly, the read efficiency can be further reduced by introducing more subranges. A similar observation was made by Demertzis and Papamanthou [10] where the read efficiency and locality of their scheme can be tuned by increasing or decreasing the space.

References

1. Crimes 2001 to present (city of chicago). https://data.cityofchicago.org/ public-safety/crimes-2001-to-present/ijzp-q8t2.
2. Enron email dataset. https://www.cs.cmu.edu/ ./enron/.
3. Tpc-h dataset.http://www.tpc.org/tpch/.
4. Usps dataset.http://www.app.com.
5. Gilad Asharov, Moni Naor, Gil Segev, and Ido Shahaf. Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 1101–1114, 2016.
6. Kenneth E. Batcher. Sorting networks and their applications. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference, Atlantic City, NJ, USA, 30 April - 2 May 1968*, pages 307–314, 1968.
7. David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, M Rosu, and Michael Steiner. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *NDSS*, 2014.

```
Algorithm $(st_\mathcal{S}, M_k, \mathcal{I}_k) \leftarrow$ SimSearch$(st_\mathcal{S}, \mathcal{L}_2(w_k), \mathcal{I}_{k-1})$:
1: Parse $st_\mathcal{S}$ as $(N, \mathsf{messages}, \{st^s_\mathcal{S}\}, \{\mathsf{available}_s\})$;
2: Parse $\mathcal{I}_{k-1}$ as $\mathbf{S}, \mathbf{M}, L, \mathcal{L}, \mathbf{H}, \{\sigma_s, \mathsf{EM}_s\}$;
3: Parse $\mathcal{L}_2(w_k)$ as $(s, b)$;
4: Set $m_k = null$;
5: if $b \neq \perp$ then                                        ▷ Query has been asked before
6:     if $N^{1-1/\log\log N} < s \leq N/\log^{2/3} N$ then
7:         $(st^s_\mathcal{S}, \mathsf{EM}_s, m_k) \leftarrow$ SimOramAccess$(st^s_\mathcal{S}, \mathsf{EM}_s)$;
8:     Set $st_\mathcal{S} \leftarrow (N, \mathsf{messages}, \{st^s_\mathcal{S}\}, \{\mathsf{available}_s\})$;
9:     Set $M_k \leftarrow (\mathsf{messages}[b], m_k)$;
10:    Set $\mathcal{I}_k \leftarrow (\mathbf{S}, \mathbf{M}, L, \mathsf{L}, \mathcal{L}, \mathbf{H}, \{\sigma_s, \mathsf{EM}_s\})$;
11:    return $(st_\mathcal{S}, M_k, \mathcal{I}_k)$;
12: if $s > N/\log^{2/3} N$ then                                 ▷ Huge sizes
13:    Read-And-Decrypt array $\mathbf{H}$;
14:    Add the above message to $\mathsf{messages}[k]$;
15: if $s \leq N^{1-1/\log\log N}$ then                           ▷ Small sizes
16:    Set $C \leftarrow c \cdot \log\log N \log^2 \log\log N$[a] and $B \leftarrow N/C$;
17:    Pick $\alpha$ and $\beta$ independently and uniformly at random from $\{1, 2, \ldots, \frac{B}{s}\}$;
18:    Read-And-Decrypt superbuckets $\mathbf{S}\{\alpha, s\}$ and $\mathbf{S}\{\beta, s\}$;
19:    Add the above message to $\mathsf{messages}[k]$;
20: else
21:    Read-And-Decrypt $\sigma_s$;
22:    Add the above message to $\mathsf{messages}[k]$;
23:    $(st^s_\mathcal{S}, \mathsf{EM}_s, m_k) \leftarrow$ SimOramAccess$(st^s_\mathcal{S}, \mathsf{EM}_s)$;
24:    Encrypt-And-Write $\sigma_s$;
25:    Add the above message to $\mathsf{messages}[k]$;
26:    if $N^{1-1/\log\log N} < s \leq N/\log^2 N$ then            ▷ Medium sizes
27:        Set $C = 3 \cdot \log N/\log\log N$ and $B \leftarrow N/C$;
28:        Pick $\alpha$ and $\beta$ independently and uniformly at random from $\{1, 2, \ldots, \frac{B}{s}\}$;
29:        Read-And-Decrypt suberbuckets $\mathbf{M}\{\alpha, s\}$ and $\mathbf{M}\{\beta, s\}$;
30:        Add the above message to $\mathsf{messages}[k]$;
31:    else
32:        Pick $\alpha$ uniformly at random from $\mathsf{available}_s$;
33:        $\mathsf{available}_s \leftarrow \mathsf{available}_s - \{\alpha\}$;
34:        if $N/\log^2 N < s \leq N/\log^{4/3} N$ then            ▷ Large sizes (scale 1)
35:            Read-And-Decrypt bucket $L[\alpha]$;
36:            Add the above message to $\mathsf{messages}[k]$;
37:        if $N/\log^{4/3} N < s \leq N/\log^{2/3} N$ then        ▷ Large sizes (scale 2)
38:            Read-And-Decrypt bucket $\mathcal{L}[\alpha]$;
39:            Add the above message to $\mathsf{messages}[k]$;
40: Set $st_\mathcal{S} \leftarrow (N, \mathsf{messages}, \{st^s_\mathcal{S}\}, \{\mathsf{available}_s\})$;
41: Set $M_k \leftarrow (\mathsf{messages}[k], m_k)$;
42: Set $\mathcal{I}_k \leftarrow (\mathbf{S}, \mathbf{M}, L, \mathcal{L}, \mathbf{H}, \{\sigma_s, \mathsf{EM}_s\})$;
43: return $(st_\mathcal{S}, M_k, \mathcal{I}_k)$;
```

[a] Constant c is appropriately chosen in [5].

Fig. 12. The simulator of the search protocol of our SE scheme.

8. David Cash and Stefano Tessaro. The locality of searchable symmetric encryption. In *EUROCRYPT, 2014*.
9. Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, 2011.
10. Ioannis Demertzis and Charalampos Papamanthou. Fast searchable encryption with tunable locality. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1053–1067, 2017.
11. Devdatt P. Dubhashi and Desh Ranjan. Balls and bins: A study in negative dependence. *Random Struct. Algorithms*, 13(2):99–124, 1998.
12. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.
13. Michael T. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*, pages 379–388, 2011.
14. Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*, pages 576–587, 2011.
15. Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011, Chicago, IL, USA, October 21, 2011*, pages 95–100, 2011.
16. Louis Granboulan and Thomas Pornin. Perfect block ciphers with small blocks. In *Fast Software Encryption, 14th International Workshop, FSE 2007, Luxembourg, Luxembourg, March 26-28, 2007, Revised Selected Papers*, pages 452–465, 2007.
17. Seny Kamara and Charalampos Papamanthou. Parallel and Dynamic Searchable Symmetric Encryption. In *Financial Cryptography*, 2013.
18. Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic Searchable Symmetric Encryption. In *CCS*, 2012.
19. Ben Morris and Phillip Rogaway. Sometimes-recurse shuffle - almost-random permutations in logarithmic expected time. In *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, pages 311–326, 2014.
20. Olga Ohrimenko, Michael T. Goodrich, Roberto Tamassia, and Eli Upfal. 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*, pages 556–567, 2014.
21. Peter Sanders, Sebastian Egner, and Jan H. M. Korst. Fast concurrent access to parallel disks. *Algorithmica*, 35(1):21–55, 2003.
22. L. A.M. Schoenmakers. A new algorithm for the recognition of series parallel graphs. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.
23. Dawn Xiaoding Song, David Wagner, and Adrian Perrig. Practical Techniques for Searches on Encrypted Data. In *SP*, 2000.
24. Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.
25. Emil Stefanov and Elaine Shi. Fastprp: Fast pseudo-random permutations for small domains. *IACR Cryptology ePrint Archive*, 2012:254, 2012.

26. Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 299–310, 2013.

Appendix

A Correctness Definitions of SE and ORAM

Definition 4 (Correctness of SE). *Let* $(\textsc{Setup}, \textsc{Search})$ *be an SE scheme. Let now* $\langle st_0, \mathcal{I}_0\rangle \leftrightarrow \textsc{Setup}\langle(1^\kappa, \mathcal{D}_0), 1^\kappa\rangle)$ *for some initial SE dataset* $\mathcal{D}_0$*. Consider* q *keyword queries* $w_1, \ldots, w_q$*. An SE scheme is correct if* $\langle(\mathcal{D}(w_k), st_k), \mathsf{EM}_k\rangle$ *are the final outputs of the protocol* $\textsc{Search}\langle(st_{k-1}, w_k), \mathcal{D}_{k-1}\rangle$ *for any* $1 \leq k \leq q$*, where* $\mathcal{D}_k$*,* $\mathcal{I}_k$*,* st_k *are the SE dataset, the encrypted index and the secret state, respectively, after the* k*-th query, and* $\textsc{Search}$ *is run between an honest client and an honest server.*

Definition 5 (Correctness of ORAM). *Let* $(\textsc{OramInitialize}, \textsc{OramAccess})$ *be an ORAM scheme. Let* $\langle \sigma_0, \mathsf{EM}_0\rangle \leftrightarrow \textsc{OramInitialize}\langle(1^\kappa, \mathsf{M}_0), 1^\kappa\rangle)$ *for some initial memory* M_0 *of* n *indexed values* $(1, v_1), (2, v_2), \ldots, (n, v_n)$*. Consider* q *arbitrary requests* $i_1, \ldots, i_q$*. We say that the ORAM scheme is correct if* $\langle(v_{i_k}, \sigma_k), \mathsf{EM}_k\rangle$ *are the final outputs of the protocol* $\textsc{OramAccess}\langle(\sigma_{k-1}, i_k), \mathsf{EM}_{k-1}\rangle$ *for any* $1 \leq k \leq q$*, where* M_k*,* EM_k*,* σ_k *are the memory array, the encrypted memory array and the secret state, respectively, after the* k*-th access operation, and* $\textsc{OramAccess}$ *is run between an honest client and server.*

B Various Useful Lemmata

Lemma 9. *Let* $\{X_1, \ldots, X_n\}$ *be negatively associated* 0-1 *random variables and* X *be their sum. Let* $\mu = \mathbb{E}[X]$ *and* μ_H *be an upper bound on* μ*, i.e.,* $\mu \leq \mu_H$*. Then, for any* $\delta > 0$*, the following version of the Chernoff bound holds*

$$\Pr[X \geq (1+\delta)\mu_H] \leq \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\mu_H} .$$

Proof. For $i = 1, \ldots, n$, let $p_i = \Pr[X_i = 1]$. By linearity of expectation, we have that $\mu = \sum_{i=1}^n p_i$. Since variables X_i are negatively associated, from [11, Lemma 2], we have that for $t > 0$,

$$\mathbb{E}[e^{tX}] = \mathbb{E}[e^{tX_1} \cdot e^{tX_2} \ldots e^{tX_n}] \leq \prod_{i=1}^n \mathbb{E}[e^{tX_i}] .$$

For each $\mathbb{E}[e^{tX_i}]$, it holds that $\mathbb{E}[e^{tX_i}] = p_i e^t + (1-p_i) = 1 + p_i(e^t - 1) \leq e^{p_i(e^t-1)}$, where we used the fact that for any k, it holds that $1 + k \leq e^k$. Replacing above we get that

$$\mathbb{E}[e^{tX}] \leq \prod_{i=1}^n \mathbb{E}[e^{tX_i}] \leq \prod_{i=1}^n e^{p_i(e^t-1)} = e^{\sum_{i=1}^n p_i(e^t-1)} = e^{\mu(e^t-1)} \leq e^{\mu_H(e^t-1)} .$$

```
(chosen, alternative) ← MaxFlowSchedule($m, n$, A, B) (Section 3.3 of [21])
1:  Let $G$ be a graph that has $n$ nodes and the following $m$ unit-capacity directed edges
        $\{(\mathsf{A}[1], \mathsf{B}[1]), (\mathsf{A}[2], \mathsf{B}[2]) \ldots, (\mathsf{A}[m], \mathsf{B}[m])\}$ ;
2:  Let $s$ and $t$ be two new nodes added to $G$ serving as the source and the sink;
3:  For all $v \in G$ such that $indeg(v) > \lceil m/n \rceil + 1$, add a directed edge $(s, v)$ of capacity
        $indeg(v) - (\lceil m/n \rceil + 1)$ ;
4:  For all $v \in G$ such that $indeg(v) < \lceil m/n \rceil + 1$, add a directed edge $(v, t)$ of capacity
        $(\lceil m/n \rceil + 1) - indeg(v)$ ;
5:  Compute the maximum flow in $G$ from $s$ to $t$;
6:  if the maximum flow in $G$ from $s$ to $t$ saturates all the edges having $s$ as origin then
7:      Change the direction of all edges $(\mathsf{A}[i], \mathsf{B}[i])$ by calling swap$(\mathsf{A}[i], \mathsf{B}[i])$ that carry flow;
8:  Let chosen and alternative be emtpy arrays of $m$ entries;
9:  for $i = 1$ to $m$ do
10:     Set chosen$[i]$ ← B$[i]$;
11:     Set alternative$[i]$ ← A$[i]$;
12: return (chosen, alternative);
```

Fig. 13. Maximum flow algorithm for finding allocation.

Finally, applying Markov's inequality, for any $t > 0$ we get that

$$\Pr[X \geq (1+\delta)\mu_H] = \Pr[e^{tX} \geq e^{t(1+\delta)\mu_H}] \leq \frac{\mathbb{E}[e^{tX}]}{e^{t(1+\delta)\mu_H}} \leq \frac{e^{\mu_H(e^t-1)}}{e^{t(1+\delta)\mu_H}} .$$

By now setting $t = \ln(1+\delta)$ (since $\delta > 0$) we get that

$$\Pr[X \geq (1+\delta)\mu_H] \leq \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\mu_H} .$$

□

Lemma 10. *For any set $U \subseteq \{1, \ldots, n\}$ and for any $\tau \geq 2$ it holds that*

$$\sum_{1 \leq |U| \leq \frac{n}{8}} \binom{n}{|U|} P_U \leq \left(\frac{|U|}{n}\right)^{(b+\tau-1)|U|+1} \cdot e^{(b+1)|U|+1} = O(1/n)^{b+\tau},$$

where $P_U = \Pr[L_U \geq (b+\tau)|U| + 1]$ and L_U is the unavoidable load of a subset of bins U, where the unavoidable load L_U is defined in Section 3.2.

Proof. By the Stirling approximation, we have that

$$\binom{n}{|U|} \leq \left(\frac{ne}{|U|}\right)^{|U|} = \left(\frac{n}{|U|}\right)^{|U|} e^{|U|}.$$

To bound $P_U = \Pr[L_U \geq (b+\tau)|U|+1]$ we note that the variables X_i^U are distributed independently from each other, and thus for a set U the variable L_U follows the binomial distribution with success probability p^2 where $p = |U|/n$. By applying the strong version of Chernoff bound, it follows from [21, Lemma 7] that for any $x \geq \mathbb{E}[L_U]$

$$\Pr[L_U \geq x] \leq \left(\frac{mp^2}{x}\right)^x \left(\frac{1-p^2}{1-x/m}\right)^{m-x}. \tag{2}$$

By Equation 2, setting $x = (b+\tau)|U|+1 \geq \mathbb{E}(L_U)$ (since $|U| \leq n$) we get that

$$\Pr[L_U \geq (b+\tau)|U|+1] \leq \left(\frac{b|U|^2/n}{(b+\tau)|U|+1}\right)^{(b+\tau)|U|+1} \left(\frac{1-|U|^2/n^2}{1-((b+\tau)|U|+1)/bn}\right)^{bn-(b+\tau)|U|-1}.$$

Let $f = \left(\frac{b|U|^2/n}{(b+\tau)|U|+1}\right)^{(b+\tau)|U|+1}$ and $g = \left(\frac{1-|U|^2/n^2}{1-((b+\tau)|U|+1)/bn}\right)^{bn-(b+\tau)|U|-1}$, i.e., $\Pr[L_U \geq (b+\tau)|U|+1] \leq f \cdot g$. We will proceed to bound these two values independently. For f we have that

$$\begin{aligned}
f = \left(\frac{b|U|^2/n}{(b+\tau)|U|+1}\right)^{(b+\tau)|U|+1} &= \left(\frac{b|U|^2}{n(b+\tau)|U|+n}\right)^{(b+\tau)|U|+1} \\
&\leq \left(\frac{b|U|^2}{n(b+\tau)|U|}\right)^{(b+\tau)|U|+1} \\
&= \left(\frac{b|U|}{n(b+\tau)}\right)^{(b+\tau)|U|+1} \\
&\leq \left(\frac{|U|}{n}\right)^{(b+\tau)|U|+1} \left(\frac{b}{b+\tau}\right)^{(b+\tau)|U|} \\
&= \left(\frac{|U|}{n}\right)^{(b+\tau)|U|+1} \left(1-\frac{\tau}{b+\tau}\right)^{(b+\tau)|U|} \\
&\leq \left(\frac{|U|}{n}\right)^{(b+\tau)|U|+1} \cdot e^{-\tau|U|}.
\end{aligned}$$

Regarding g, first note that $1-|U|^2/n^2 = (1+|U|/n)(1-|U|/n)$. We now split g into two functions g_1, g_2 with $g = g_1 g_2$ such that

$$\begin{aligned}
g_1 &= \left(1+\frac{|U|}{n}\right)^{bn-(b+\tau)|U|-1} \leq \left(1+\frac{|U|}{n}\right)^{bn} \leq e^{b|U|} \\
g_2 &= \left(\frac{1-|U|/n}{1-((b+\tau)|U|+1)/bn}\right)^{bn-(b+\tau)|U|-1} = \left(\frac{bn-b|U|}{bn-(b+\tau)|U|-1}\right)^{bn-(b+\tau)|U|-1} \\
&= \left(1+\frac{(b+\tau)|U|-b|U|+1}{bn-(b+\tau)|U|-1}\right)^{bn-(b+\tau)|U|-1} \leq e^{(b+\tau)|U|-b|U|+1} = e^{\tau|U|+1}.
\end{aligned}$$

By combining all the bounds, we get

$$\begin{aligned}\binom{n}{|U|}P_U &\le \left(\frac{n}{|U|}\right)^{|U|} e^{|U|} \cdot \left(\frac{|U|}{n}\right)^{(b+\tau)|U|+1} \cdot e^{-\tau|U|} \cdot e^{b|U|} \cdot e^{\tau|U|+1} \\ &= \left(\frac{|U|}{n}\right)^{(b+\tau-1)|U|+1} \cdot e^{(b+1)|U|+1}.\end{aligned}$$

Let us view $\left(\frac{|U|}{n}\right)^{(b+\tau-1)|U|+1} \cdot e^{(b+1)|U|+1}$ as function $h(|U|)$. Its second derivative can written in the form $A\ln^2(|U|/n) + B\ln(|U|/n) + C$ where

$$A = \left(\tau^2 + (2b-2)\,\tau + b^2 - 2b + 1\right)|U|\,,$$

$$B = \left(2\tau^2 + (6b-2)\,\tau + 4b^2 - 4b\right)|U| + 2\tau + 2b - 2\,,$$

and

$$C = \left(\tau^2 + 4b\tau + 4b^2\right)|U| + 3\tau + 5b - 1\,.$$

Its discriminant is $-4\,(\tau+b-1)^2\,((\tau+b-1)\,|U| - 1)$ which is ≤ 0 when $\tau \ge 2$ and $|U| > 0$, as it is in our scenario. Since also for $\tau \ge 2$ it is $A \ge 0$ it follows that for positive $|U|$ the second derivative is always non-negative. Therefore, for any $[x,y]$ interval with $x, y > 0, x \le y$, h gets it maximum value as $h(x)$ or $h(y)$. Therefore, we can write

$$T_1 = \sum_{1\le|U|\le\frac{n}{8}} \binom{n}{|U|}P_U \le h(1) + (n/8)\max\{h(2), h(n/8)\}.$$

Moreover it holds that

$$(\mathbf{1})\quad h(1) = n^{-(b+\tau)} \cdot e^{b+2} = n^{-(b+\tau)} \cdot e^{b+\tau}e^{2-\tau} \le e^2(e/n)^{b+\tau} = O(1/n)^{b+\tau}$$

$$(\mathbf{2})\quad (n/8)h(2) = (n/8)(2/n)^{2b+2\tau-1} \cdot e^{2b+3} \le (1/16\cdot e^3)(2e/n)^{2(b+\tau)} = O(1/n)^{2(b+\tau)}.$$

Finally, it also holds that

$$\begin{aligned}(\mathbf{3})\quad (n/8)h(n/8) &= (n/8)\left(\frac{1}{8}\right)^{(b+\tau-1)(n/8)+1} \cdot e^{(b+1)(n/8)+1} \\ &= (n/8)\cdot e^{(\ln\frac{1}{8})\left((b+\tau-1)(n/8)+1\right)+(b+1)(n/8)+1} \\ &= (n/8)\cdot e^{(n/8)\left((\ln\frac{1}{8})(b+\tau-1)+(b+1)\right)+1+\ln\frac{1}{8}} \\ &< O(n)e^{-(n/8)(b+2\tau-3)} = e^{-\Omega(n)}\,,\end{aligned}$$

where we used the fact that $\ln(1/8) < -2$ and $\tau \ge 2$. Combining these three bounds we get that $T_1 = O(1/n)^{b+\tau}$ (which is the weakest one). □

C ORAM Correctness and Security Proofs

Lemma 11. *The ORAM scheme in Figure 4 is correct according to Definition 5.*

Proof. It is enough to prove that for all indices i, (i, v_i) will be always stored either in C or in $A[\pi_a[i]]$ or in $B[\pi_b[\mathsf{Tab}[i]]]$—these are the values from which we retrieve v_i in Line 16 of our construction in Figure 4. We consider the following disjoint cases.

1. **(i has been accessed since the last reshuffle)** In this case, (i, v_i) can be found in C since it was stored there during the last access to it and C has not been emptied in the meantime.
2. **(i has not been accessed since the last large reshuffle)** In this case, (i, v_i) can be found only in $A[\pi[i]]$ since during a large reshuffle all the elements of the dataset are reshuffled into A (and stay there if not accessed afterwards) and all the other arrays are emptied.
3. **(i has been accessed since the last large reshuffle but not since the last small reshuffle)** In this case, the element can be found in $B[\pi_b[\mathsf{Tab}[i]]]$. This is because, after its first access that occurred after the large reshuffle element i moved to C and after the small reshuffle element i moved to B with a new index $\mathsf{Tab}[i]$ in B and it was stored at location $\pi_b[\mathsf{Tab}[i]]$ during the small reshuffle. Since it was never accessed after the small reshuffle, it remained in B.

□

```
Algorithm $(st_\mathcal{S}, \mathsf{EM}_0) \leftarrow$ SIMORAMINITIALIZE$(1^\kappa, |\mathsf{M}_0|)$:
1: Let $(n, \lambda) = |\mathsf{M}_0|$;                ▷ Recall $\lambda$ is the size of the ORAM block
2: for $i = 1$ to $n$ do
3:     Set $v_i = \mathbf{0}^\lambda$;
4:     $\mathsf{M}_0[i] = (i, v_i)$;
5: $\langle \sigma_0, \mathsf{EM}_0 \rangle \leftrightarrow$ ORAMINITIALIZE$\langle (1^\kappa, \mathsf{M}_0), \perp \rangle$;
6: return $(\sigma_0, \mathsf{EM}_0)$;
Algorithm $(st_\mathcal{S}, \mathsf{EM}_k, m_k) \leftarrow$ SIMORAMACCESS$(st_\mathcal{S}, \mathsf{EM}_{k-1})$:
  Parse $st_\mathcal{S}$ as $\sigma_{k-1}$;
  Choose $i_k \in [n]$;
  $\langle (v_{i_k}, \sigma_k), \mathsf{EM}_k \rangle \leftrightarrow$ ORAMACCESS$\langle (\sigma_{k-1}, i_k), \mathsf{EM}_{k-1} \rangle$;
  Let $m_k$ be the messages sent from client to server during the above ORAMACCESS protocol;
  return $(\sigma_k, \mathsf{EM}_k, m_k)$;
```

Fig. 14. The simulator for the ORAM scheme of Figure 4.

Lemma 12. *The ORAM scheme in Figure 4 is secure according to Definition 2 and assuming pseudorandom permutations and CPA-secure encryption.*

Proof. Our simulator is shown in Figure 14. Note that all EM_i are trivially indistinguishable from the EM_i output by the real game due to the CPA-security of the encryption scheme that is used—recall that whatever is being written on the server by our protocols is always freshly encrypted.

We now argue that the messages $m_1, m_2, \ldots, m_q$ in the real game are indistinguishable from the messages $m_1, m_2, \ldots, m_q$ output by the simulator. This is because for each $1 \leq k \leq q$, the set of message m_k is entirely independent of the queried value i_k had we used truly random permutations for π_a and π_b. This follows from the following facts:

– When accessing i_k, array C, stored at a fixed memory location is accessed in its entirety. Also $(\mathsf{Tab}[i_k], v_{i_k})$ is uploaded encrypted at a fixed position count_a in $\mathsf{SCRATCH}$ (see Line 20). So both memory accesses are independent of the index i_k.
– When accessing i_k within a specific superepoch, a location $x = \pi_a[y]$ from array A is accessed for the first and last time within the specific superepoch. Since x is the output of a truly random permutation and is accessed only once within the specific superepoch, x is independent of i_k. The same argument applies for the accesses made to array B. Now if we replace the truly random permutation with the pseudorandom permutation of our construction, the adversary can gain a negligible advantage which is acceptable.
– When accessing i_k at the end of the current superepoch, an oblivious sorting is executed whose memory accesses do not depend on the actual data that are being sorted, but only on the size of the array that is being sorted. Same argument appies for the case when i_k is accessed at the end of an epoch.

□

D ORAM Construction with Worst-Case Complexities

Our ORAM construction with worst-case complexities is in Figure 16. We now describe the main idea.

Large Reshuffling. Along with the first ORAM access, we can immediately perform $c \cdot n^{1/3} \log^2 n$ work (or one I/O) of the large reshuffling, under another pseudorandom permutation π'_a. This is because we know ahead of time that what needs to be reshuffled is the dataset we started with—this is because no writes are supported and therefore data will never change.[14] By continuing in this way, after $n^{2/3}$ accesses, the large reshuffling would be complete. To store the output of the new reshuffling of A under π'_a we use another array $\mathcal{A}$ of $n + n^{2/3}$ entries.

Small Reshuffling. Unlike the large one, the small reshuffling cannot start right along with the first ORAM access. This is because when the first ORAM access is performed, it is not known in advance which elements are going to be accessed in this epoch (this information will only be available by the end of this epoch). To address this problem, we introduce some artificial delay in the reshuffling. More formally, during the $n^{1/3}$ accesses of an epoch i, what is being reshuffled are the elements that were accessed during epochs $1, 2, \ldots, i-1$ of the same superepoch—and we store these elements in

[14] We can support writes as well, but the de-amortization of the large reshuffling would be much more complicated and we do not describe it here.

an array SCRATCH. Note that this implies that during the very first epoch ($i = 1$), we are reshuffling an array that contains dummy elements. We now give some more details concerning the implementation of the above.

As in the amortized version, while an epoch i proceeds, we move elements accessed in epoch i into array C. However, we also maintain an other array $\mathcal{C}$ of size n_c. When epoch i finishes we perform the following steps:

1. We append the contents of array $\mathcal{C}$ to SCRATCH;
2. We set our new array $\mathcal{C}$ to be array C;
3. We empty array C.

At that point, all elements of epochs $1, 2, \ldots, i-1$ are stored in SCRATCH and they are ready for reshuffling.

E Computing the Constants in the Asymptotics

In this section, we compute the hidden constants for c, n_0 in Theorem 1. In order to do this, we first provide the closed formula for an upper bound on $\Pr[L^*_{\mathsf{max}} > \lceil m/n \rceil + 1]$ from [21], which is the following:

$$\Pr[L^*_{\mathsf{max}} > \lceil m/n \rceil + 1] \leq e(e/n)^{b+1} + B(n) + \sum_{\frac{n}{8} < |U| < \frac{nb}{b+1}} \binom{n}{|U|} P^*_U$$

where $B(n) \overset{\text{def}}{=} \max\{(n/8)(2/n)^{2b+1}e^{2b+3}, (n/8)(1/8)^{(1/8)nb+1}e^{(n/8)(b+1)+1}\}$ and $P^*_U = \Pr[L_U \geq (b+1)|U| + 1]$.

Regarding the first term, note that $\forall n > 2$ it is upper bounded by e^2/n. In order to bound $B(n)$, we handle the two values separately. For the first value we have that

$$(n/8)(2/n)^{2b+1}e^{2b+3} = (e^3/4)(2e/n)^{2b}$$

which $\forall n > 5$ is upper bounded by $e^4/2n$. For the second value we have that

$$\begin{aligned}(n/8)(1/8)^{(1/8)nb+1}e^{(n/8)(b+1)+1} &= (e/64) \cdot n \cdot e^{\ln(1/8)(nb/8)+(nb/8)+n/8} \\ &= (e/64) \cdot n \cdot e^{(n/8)(b(\ln(1/8)+1)+1)} \\ &< (e/64) \cdot n \cdot e^{(n/8)(\ln(1/8)+2)}\end{aligned}$$

where we used the fact that $\ln(1/8) + 1 < -1$. For $n > 655$, the above is also upper bounded by $e^4/2n$.

Lastly, in order to bound the sum we proceed as follows. First, recall that we defined $p = |U|/n$. Then, from [21, Lemma 7] we have that

$$P^*_U \leq \left(\left(\frac{bp}{b+1} \right)^{p(b+1)} \left(\frac{1-p^2}{1-p-p/b} \right)^{b-p(b+1)} \right)^n$$

.

```
Protocol $\langle \perp, Y \rangle \leftrightarrow$ ObliviousSorting$\langle (\pi, n, b), X \rangle$:
$\triangleright$ Assume $n$ and $b$ are powers of 2 $\triangleright$ Also assume that $X[i]$ also stores the respective index $i$, so
that comparisons using $\pi$ are possible while elements are being moved around
 1: if $n \leq b$ then
 2:     Read-And-Decrypt array $X$. Set $Y$ to be the sorted version of $X$[a];
 3: else
 4:     $\langle \perp, Y_1 \rangle \leftrightarrow$ ObliviousSorting$\langle (\pi, n/2, b), X[1, \ldots, n/2] \rangle$;
 5:     $\langle \perp, Y_2 \rangle \leftrightarrow$ ObliviousSorting$\langle (\pi, n/2, b), X[n/2+1, \ldots, n] \rangle$;
 6:     $\langle \perp, Y \rangle \leftrightarrow$ ObliviousMerge$\langle (\pi, n, b), (Y_1, Y_2) \rangle$;
 7: Encrypt-And-Write array $Y$;
 8: return $\langle \perp, Y \rangle$;
Protocol $\langle \perp, Y \rangle \leftrightarrow$ ObliviousMerge$\langle (\pi, n, b), (Y_1, Y_2) \rangle$:          $\triangleright$ $Y_1, Y_2$ must be sorted
 1: if $n \leq b$ then
 2:     Read-And-Decrypt array $Y_1$;
 3:     Read-And-Decrypt array $Y_2$;
 4:     Set $Y$ to be the merged array of $Y_1$ and $Y_2$;
 5: else
 6:     Let $D$ be a $2 \times n/2$ matrix and $Y$ be a length $n$ array stored at the server;
 7:     $j = 0$;
 8:     for $i = 1, 2b+1, 4b+1, \ldots, n/2 - 2b + 1$ do
 9:         Initialize arrays $D_1, D_2, D_3, D_4$ of size $b$;
10:         Read-And-Decrypt subarray $Y_1[i, \ldots, i+b-1]$;
11:         Read-And-Decrypt subarray $Y_1[i+b, \ldots, i+2b-1]$;
12:         Read-And-Decrypt subarray $Y_2[i, \ldots, i+b-1]$;
13:         Read-And-Decrypt subarray $Y_2[i+b, \ldots, i+2b-1]$;
14:         Store $Y_1[i], Y_1[i+2], \ldots, Y_1[i+2b-2]$ at the first available position of $D_1$;
15:         Store $Y_1[i+1], Y_1[i+3], \ldots, Y_1[i+2b-1]$ at the first available position of $D_3$;
16:         Store $Y_2[i], Y_2[i+2], \ldots, Y_2[i+2b-2]$ at the first available position of $D_2$;
17:         Store $Y_2[i+1], Y_2[i+3], \ldots, Y_2[i+2b-1]$ at the first available position of $D_4$;
18:         Encrypt-And-Write $D_1$ in $D$'s row 1, from position $1 + j \cdot b$ onwards;
19:         Encrypt-And-Write $D_2$ in $D$'s row 1, from position $n/4 + 1 + j \cdot b$ onwards;
20:         Encrypt-And-Write $D_3$ in $D$'s row 2, from position $1 + j \cdot b$ onwards;
21:         Encrypt-And-Write $D_4$ in $D$'s row 2, from position $n/4 + 1 + j \cdot b$ onwards;
22:         $j \leftarrow j + 1$;
23:     $\langle \perp, D[1,:] \rangle \leftrightarrow$ ObliviousMerge$\langle (\pi, n/2, b), (D[1, 1:n/4], D[1, n/4+1:n/2]) \rangle$;
24:     $\langle \perp, D[2,:] \rangle \leftrightarrow$ ObliviousMerge$\langle (\pi, n/2, b), (D[2, 1:n/4], D[2, n/4+1:n/2]) \rangle$;
25:     Let $Z_1, \ldots, Z_{n/2b}$ be the $2 \times b$ submatrices that result from partitioning $D$ horizontally;
26:     for $i = 1$ to $n/2b - 1$ do
27:         Read-And-Decrypt $Z_i$;
28:         Read-And-Decrypt $Z_{i+1}$;
29:         Sort $Z_i \cup Z_{i+1}$ and let $y_1, \ldots, y_{2b}$ be the smallest resulting elements;
30:         Encrypt-And-Write $[y_1, \ldots, y_b]$ starting at the first available position of $Y$;
31:         Encrypt-And-Write $[y_{b+1}, \ldots, y_{2b}]$ starting at the first available position of $Y$;
32:     Sort $Z_{n/2b}$ and let $y_1, \ldots, y_{2b}$ be the sorted sequence;
33:     Encrypt-And-Write $[y_1, \ldots, y_b]$ starting at the first available position of $Y$;
34:     Encrypt-And-Write $[y_{b+1}, \ldots, y_{2b}]$ starting at the first available position of $Y$;
35: return $\langle \perp, Y \rangle$;
```

[a] We use π to perform comparisons between two elements of X, i.e., $X[i]$ isLessThan $X[j]$ iff $p[i] < p[j]$.

Fig. 15. Data-oblivious and I/O efficient sorting by Goodrich and Mitchenmacher [14].

Protocol $\langle \sigma, \mathsf{EM}\rangle \leftrightarrow$ ORAMINITIALIZE$\langle(1^\kappa, \mathsf{M}), \perp\rangle$:

```
 1: Parse $\mathsf{M}$ as $(1, v_1), (2, v_2), \ldots, (n, v_n)$ where $|i, v_i| = \lambda$ (the values are $\lambda$ bits long);
 2: Let $n_a \leftarrow n + n^{2/3}$, $n_b \leftarrow n^{2/3} + n^{1/3}$, $n_c \leftarrow n^{1/3}$;
 3: Let $A$, $B$ and $C$ be arrays of size $n_a$, $n_b$ and $n_c$ respectively. Initialize them with **0** entries;
 4: Let $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ be arrays of size $n_a$, $n_b$ and $n_c$ respectively. Initialize them with **0** entries;
 5: Let SCRATCH be an array of size $n_b$. Initialize it with **0** entries;
 6: Let $\pi_a : [n_a] \to [n_a]$ and $\pi_b : [n_b] \to [n_b]$ be pseudorandom permutations;
 7: Let $\pi'_a : [n_a] \to [n_a]$ and $\pi'_b : [n_b] \to [n_b]$ be pseudorandom permutations;
 8: For $i = 1, \ldots, n$, store $(i, v_i)$ at location $\pi_a[i]$ in $A$;
 9: Encrypt-And-Write arrays $A$, $B$, $C$, $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and SCRATCH and add them to EM;
10: Let $\mathsf{count}_a \leftarrow 0$ and $\mathsf{count}_b \leftarrow 0$;
11: Let Tab be an empty hash table;
12: Set $\sigma = (\pi_a, \pi_b, \pi'_a, \pi'_b, \mathsf{Tab}, \mathsf{count}_a, \mathsf{count}_b)$;
13: return $\langle \sigma, \mathsf{EM}\rangle$;
```

Protocol $\langle (v_i, \sigma'), \mathsf{EM}'\rangle \leftrightarrow$ ORAMACCESS$\langle(\sigma, i), \mathsf{EM}\rangle$:

```
 1: Parse $\sigma$ as $(\pi_a, \pi_b, \pi'_a, \pi'_b, \mathsf{Tab}, \mathsf{count}_a, \mathsf{count}_b)$, EM as $(A, B, C, \mathcal{A}, \mathcal{B}, \mathcal{C}, \mathsf{SCRATCH})$;
 2: Increment $\mathsf{count}_a$ and $\mathsf{count}_b$;
 3: Read-And-Decrypt arrays $C$ and $\mathcal{C}$;
 4: if $(i, v_i) \in C$ or $(i, v_i) \in \mathcal{C}$ then        ▷ $(i, v_i)$ was accessed before and is stored in $C$ or $\mathcal{C}$
 5:     $\mathsf{index}_a \leftarrow \pi_a[n + \mathsf{count}_a]$;
 6:     $\mathsf{index}_b \leftarrow \pi_b[n^{2/3} + \mathsf{count}_b]$;
 7: else
 8:     if $\mathsf{Tab}[i] \neq$ null then                     ▷ $(i, v_i)$ is stored in $B[\mathsf{index}_b]$
 9:         $\mathsf{index}_a \leftarrow \pi_a[n + \mathsf{count}_a]$;
10:         $\mathsf{index}_b \leftarrow \pi_b[\mathsf{Tab}[i]]$;
11:     else                                                ▷ $(i, v_i)$ is stored in $A[\mathsf{index}_a]$
12:         $\mathsf{index}_a \leftarrow \pi_a[i]$;
13:         $\mathsf{index}_b \leftarrow \pi_b[n^{2/3} + \mathsf{count}_b]$;
14: Read-And-Decrypt $A[\mathsf{index}_a]$;
15: Read-And-Decrypt $B[\mathsf{index}_b]$;
16: Retrieve $(i, v_i)$ from either $A[\mathsf{index}_a]$ or $B[\mathsf{index}_b]$ or $C$ or $\mathcal{C}$;
17: $C[\mathsf{count}_b] \leftarrow (i, v_i)$;
18: Encrypt-And-Write arrays $C$ and $\mathcal{C}$;
19: $\mathsf{Tab}[i] \leftarrow \mathsf{count}_a$;
20: Execute one I/O of the protocol

        $\langle \perp, \mathcal{A}\rangle \leftrightarrow$ OBLIVIOUSSORTING$\langle(\pi'_a, n_a, c \cdot n^{1/3} \log^2 n), A\rangle$ ;

21: Execute one I/O of the protocol

        $\langle \perp, \mathcal{B}\rangle \leftrightarrow$ OBLIVIOUSSORTING$\langle(\pi'_b, n_a, c \cdot n^{1/3} \log^2 n), \mathsf{SCRATCH}\rangle$ ;

22: if $\mathsf{count}_a > n^{2/3}$ then
23:     $\pi_a \leftarrow \pi'_a$;
24:     Let $\pi'_a, \pi_b, \pi'_b$ be new pseudorandom permutations;
25:     $\mathsf{count}_a \leftarrow 0$ and $\mathsf{count}_b \leftarrow 0$;
26:     Set $A \leftarrow \mathcal{A}$; $B \leftarrow \perp$; $C \leftarrow \perp$; $\mathcal{A} \leftarrow \perp$; $\mathcal{B} \leftarrow \perp$; $\mathcal{C} \leftarrow \perp$; SCRATCH $\leftarrow \perp$; [a]
27:     Set Tab $\leftarrow \perp$;
28: if $\mathsf{count}_b > n^{1/3}$ then
29:     $\pi_b \leftarrow \pi'_b$;
30:     Let $\pi'_b$ be a new pseudorandom permutation;
31:     $\mathsf{count}_b \leftarrow 0$;
32:     Set $\mathcal{C} \leftarrow C$; $C \leftarrow \perp$; $B \leftarrow \mathcal{B}$; $\mathcal{B} \leftarrow \perp$;
33:     Read-And-Decrypt array $\mathcal{C}$;
34:     $y \leftarrow \mathsf{count}_a - n^{1/3}$ and $cnt \leftarrow 0$;
35:     for $(i, v_i) \in \mathcal{C}$ do
36:         Increment $cnt$;
37:         Encrypt-And-Write $(\mathsf{Tab}[i], v_i)$ into $\mathsf{SCRATCH}[y + cnt]$;
38: return $\langle (v_i, (\pi_a, \pi_b, \pi'_a, \pi'_b, \mathsf{Tab}, \mathsf{count}_a, \mathsf{count}_b)), (A, B, C, \mathcal{A}, \mathcal{B}, \mathcal{C}, \mathsf{SCRATCH})\rangle$;
```

[a] These are all pointer operations, and not actual copying.

We can also write without loss of generality that $\sum_{\frac{n}{8}<|U|<\frac{nb}{b+1}} \binom{n}{|U|} = \sum_{\frac{1}{8}<|p|<\frac{b}{b+1}} \binom{n}{pn}$. From the lower bound of Stirling's approximation for the factorial, for these values of p we have that

$$\binom{n}{pn} = \frac{n!}{pn!(n-pn)!} < \frac{n!}{\sqrt{2\pi pn}\left(\frac{pn}{e}\right)^{pn}\sqrt{2\pi(n-pn)}\left(\frac{n-pn}{e}\right)^{n-pn}}$$
$$< \frac{n!}{2\pi\sqrt{1/8}\cdot n\left(\frac{n}{e}\right)^{n}(p^p(1-p)^{(1-p)})^n}$$

Using the upper bound of Stirling's approximation it is easy to verify that for $n > 1$ we have that

$$\binom{n}{pn} < \frac{n!}{2\pi\sqrt{1/8}\cdot n\left(\frac{n}{e}\right)^{n}(p^p(1-p)^{(1-p)})^n}$$
$$< \frac{1.05\sqrt{2\pi n}\left(\frac{n}{e}\right)^{n}}{2\pi\sqrt{1/8}\cdot n\left(\frac{n}{e}\right)^{n}(p^p(1-p)^{(1-p)})^n} = \frac{1.05}{\sqrt{\pi/4}}\cdot\frac{1}{\sqrt{n}}\cdot\left(p^{-p}(1-p)^{(1-p)}\right)^n.$$

Finally, from [21, Lemma 9] we have that

$$\frac{\left(\frac{bp}{b+1}\right)^{p(b+1)}\left(\frac{1-p^2}{1-p-p/b}\right)^{b-p(b+1)}}{p^p(1-p)^{1-p}} < 0.9$$

therefore we get that

$$\sum_{\frac{n}{8}<|U|<\frac{nb}{b+1}} \binom{n}{|U|} P_U^* \le n\cdot\frac{1.05}{\sqrt{\pi/4}}\cdot\frac{1}{\sqrt{n}}\cdot 0.9^n = \frac{1.05}{\sqrt{\pi/4}}\cdot\sqrt{n}\cdot 0.9^n$$

since the sum contains at most n terms, and for $n > 57$ the above is upper bounded by $1.05/(n\sqrt{\pi/4})$.

Putting it all together, we have that $\forall n > 655$, it holds that

$$\Pr[Y_\ell > \lceil m/n\rceil + 1] \le e^2/n + e^4/2n + \frac{1.05}{n\pi/4} < \frac{36}{n}.$$

From the above it follows that $n_0 = 655$ and $c = 36$.[15] Concretely, this implies that in order to apply the bound of Theorem 1 for our main construction, one must ensure that there exist at least 655 bins per layer for medium-size allocations. Recall that the smallest number of bins in our scheme is $\log N \log\log N$. For reasonable database sizes, e.g., $N = 2^{32}$, this yields 160 bins. The desired minimum number can be easily achieved by "padding" with the necessary amount of bins (e.g., by multiplying the number of bins per layer by a constant factor of 4).

[15] We also believe that a more mathematically involved analysis can yield a tighter bound.