Optimal Offline ORAM with Perfect Security via Simple Oblivious Priority Queues

Thore Thießen
University of Münster, Germany

Jan Vahrenhold
University of Münster, Germany

Abstract

Oblivious RAM (ORAM) is a well-researched primitive to hide the memory access pattern of a RAM computation; it has a variety of applications in trusted computing, outsourced storage, and multiparty computation. In this paper, we study the so-called *offline* ORAM in which the sequence of memory access locations to be hidden is known in advance. Apart from their theoretical significance, offline ORAMs can be used to construct efficient oblivious algorithms.

We obtain the first optimal offline ORAM with perfect security from oblivious priority queues via time-forward processing. For this, we present a simple construction of an oblivious priority queue with perfect security. Our construction achieves an asymptotically optimal (amortized) runtime of $\Theta(\log N)$ per operation for a capacity of N elements and is of independent interest.

Building on our construction, we additionally present efficient external-memory instantiations of our oblivious, perfectly-secure construction: For the cache-aware setting, we match the optimal I/O complexity of $\Theta(\frac{1}{B}\log\frac{N}{M})$ per operation (amortized), and for the cache-oblivious setting we achieve a near-optimal I/O complexity of $\mathcal{O}(\frac{1}{B}\log\frac{N}{M}\log\log_M N)$ per operation (amortized).

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Cryptographic protocols

Keywords and phrases offline ORAM, oblivious priority queue, perfect security, external memory algorithm, cache-oblivious algorithm

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2024.55

Related Version *Full Version*: `https://arxiv.org/abs/2409.12021` [31]

Acknowledgements We thank all anonymous reviewers for their constructive comments that helped to improve the presentation.

1 Introduction

Introduced by Goldreich and Ostrovsky [16], *oblivious RAM* (*ORAM*) conceals the memory access pattern of any RAM computation. This prevents the leakage of confidential information when some adversary can observe the pattern of memory accesses. We consider oblivious RAM in the offline setting: This allows an additional pre-processing step on the access pattern while still requiring that the access pattern is hidden from the adversary.

Offline ORAMs can be used to construct efficient oblivious algorithms in situations where at least part of the memory access sequence is either known or can be inferred in advance. As a motivating example, consider the classical Gale–Shapley algorithm for the stable matching problem [15, 27]: In each round of the algorithm, up to n parties make a proposal according to their individual preferences. The preferences must be hidden to maintain obliviousness, and thus the memory access pattern may not depend on them. While it seems that the standard algorithm makes online choices, in fact the preferences and the current matching are known before each round, so the proposals can be determined in advance and an offline ORAM can be used to hide the access pattern in each round.

Many of the previous works on offline (and online) ORAMs focus on statistical and computational security: While optimal offline ORAMs are known for computational and statistical security [4, 29], the same is not true for perfect security. We close this gap and obtain the first (asymptotically) optimal offline ORAM with perfect security. We derive our construction from an oblivious priority queue.

For this, we discuss and analyze a construction of an oblivious priority queue simple enough to be considered part of folklore. In fact, both the construction and its analysis can be used in an undergraduate data structures course as an example of how to construct an efficient oblivious data structure from simple building blocks. Our construction reduces the problem to oblivious partitioning where an optimal oblivious algorithm [4] is known.

1.1 Oblivious Data Structures

Conceptually, (offline) ORAM and oblivious priority queues are *oblivious data structures*. Oblivious data structures provide efficient means to query and modify data while not leaking information, e. g., distribution of the data or the operations performed, via the memory access pattern. There are three main applications:

Outsourced Storage. When storing data externally, oblivious data structures can be used in conjunction with encryption. Encryption alone protects the confidentiality of the data at rest, but performing operations may still leak information about queries or the data itself via the access pattern [20].

Trusted Computing. When computing in trusted execution environments, oblivious data structures safeguard against many memory-related side channel attacks [28].

(Secure) Multiparty Computation. In this setting, actors want to (jointly) compute a function without revealing their respective inputs to each other. Here, oblivious data structures have been used to allow for data structure operations with sublinear runtime [32, 23].

1.1.1 Security Definition

In line with standard assumptions for oblivious algorithms [16], we assume the w-bit word RAM model of computation. Let the random variable $\mathrm{Addr}_{\textsc{Op}(x)}$ with

$$\mathrm{Addr}_{\textsc{Op}(x)} \in (\{0, \ldots, 2^w - 1\} \times \{\textsc{Read}, \textsc{Write}\})^* \tag{1}$$

denote the sequence of *memory probes* for $\textsc{Op}(x)$, i. e., the sequence of memory access locations and memory operations performed by operation $\textsc{Op}$ for input x. Access to a constant number of registers (*private memory*) is excluded from the probe sequence.

For *perfect* oblivious security, we require that all data structure operation sequences of length n produce the same memory access pattern:

▶ **Definition 1** (Obliviousness with Perfect Security). *We say that an (online) data structure $\mathcal{D}_N$ with capacity*[1] *N and operations $\textsc{Op}_1, \ldots, \textsc{Op}_m$ is oblivious with perfect security iff, for every two sequences of n operations*

$$X = \langle \textsc{Op}_{i_1}(x_1), \ldots, \textsc{Op}_{i_n}(x_n) \rangle \quad and \quad Y = \langle \textsc{Op}_{j_1}(y_1), \ldots, \textsc{Op}_{j_n}(y_n) \rangle$$

with valid inputs x_k, y_k, the memory probe sequences are identically distributed, i. e.,

$$\langle \mathrm{Addr}_{\textsc{Op}_{i_1}(x_1)}, \ldots, \mathrm{Addr}_{\textsc{Op}_{i_n}(x_n)} \rangle \equiv \langle \mathrm{Addr}_{\textsc{Op}_{j_1}(y_1)}, \ldots, \mathrm{Addr}_{\textsc{Op}_{j_n}(y_n)} \rangle .$$

[1] To hide the type of operation performed, in particular for intermixed Insert and Delete sequences, it is assumed that the data structure has a fixed capacity N determined a priori. This assumption does not limit any of our analyses, as the capacity can be adjusted using standard (doubling) techniques with (amortized) constant asymptotic overhead per operation.

Table 1 Oblivious priority queues supporting INSERT, MIN, and DELETEMIN. Deletions are noted as supported if an operation DELETE, MODIFYPRIORITY, or DECREASEPRIORITY is available.

security	*runtime*	*priv. memory*	*deletion*	
perfect[p]	$\mathcal{O}(\log^2 N)^{\text{a}}$	$\mathcal{O}(1)$	no	[32]
statistical	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(\omega(1) \cdot \log N)$	no	[33]
statistical	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(\omega(1) \cdot \log N)$	yes[r]	[23, Path ORAM variant]
perfect	$\mathcal{O}(\log^2 N)^{\text{a}}$	$\mathcal{O}(1)$	no	[26]
statistical	$\mathcal{O}(\log N)^{\text{a}}$	$\mathcal{O}(\omega(1) \cdot \log N)$	yes	[22]
statistical	$\mathcal{O}(\omega(1) \cdot \log N)$	$\mathcal{O}(1)$	yes[r]	[29, Circuit variant]
perfect	$\mathcal{O}(\log^2 N)^{\text{a}}$	$\mathcal{O}(1)$	yes[r]	[19]
perfect	$\mathcal{O}(\log N)^{\text{a}}$	$\mathcal{O}(1)$	no	**new**

[p] reveals the operation [a] amortized runtime complexity [r] requires an additional reference

The requirement of identical distribution in the above definition can be relaxed to strictly weaker definitions of security by either allowing a negligible statistical distance of the probe sequences (*statistical security*) or allowing a negligible distinguishing probability by a polynomial-time adversary (*computational security*); see Asharov et al. [4] for more details.

Definition 1 immediately implies that the memory probe sequence is independent of the operation arguments – and, by extension, the data structure contents – as well as the operations performed (*operation-hiding security*). As a technical remark, we note that for perfectly-secure data structure operations with determined outputs, the joint distributions of output and memory probe sequence are also identically distributed. This implies that data structures satisfying Definition 1 are universally composable [4].

1.1.2 Offline ORAM

The (online) ORAM is essentially an oblivious array data structure [24]. By using an ORAM as the main memory, any RAM program can generically be transformed into an oblivious program at the cost of an *overhead* per memory access.

The offline ORAM we are considering here, however, is given the sequence I of access locations in advance. While this allows pre-computations on I, the probe sequence must still hide the operations and indices in I. In anticipation of the offline ORAM construction in Section 3, we take a similar approach as Mitchell and Zimmerman [26] and define an offline ORAM as an *online* oblivious data structure with additional information:

▶ **Definition 2** (Offline ORAM). *An offline ORAM is an oblivious data structure $\mathcal{D}_N$ that maintains an array of length N under an annotated online sequence of read and write operations:*

READ(i, τ) *Return the value stored at index i in the array.*

WRITE(i, v', τ) *Store the value v' in the array at index i.*

The annotation τ indicates the time-stamp of the next operation accessing index i.

Note that this definition implies that $\mathcal{D}_N$ can also be used in an online manner if the time-stamps τ of the next operation accessing the index i are known. When discussing the offline ORAM construction in Section 3, we show how to use sorting and linear scans to compute the annotations τ from the sequence I of access locations given in advance.

Table 2 Best known overhead bounds for online and offline ORAMs with N memory cells, a constant number of private memory cells, and standard parameters [24].

	perfect security		*statistical security*		*comput. security*	
online	$\Omega(\log N)$	[24]	$\Omega(\log N)$	[24]	$\Omega(\log N)$	[24]
	$\mathcal{O}(\log^3 N/\log\log N)$	[11]	$\mathcal{O}(\log^2 N)$	[10]	$\mathcal{O}(\log N)^{\mathrm{p}}$	[5]
offline	$\Omega(\log N)^{\mathrm{i}}$	[16]	$\Omega(\log N)^{\mathrm{i}}$	[16, 8]	$\Omega(1)$	trivial [8]
	$\mathcal{O}(\log^2 N)^{\mathrm{a}}$	e. g., via [26]	$\mathcal{O}(\omega(1)\cdot\log N)$	[29]	$\mathcal{O}(\log N)^{\mathrm{p}}$	[5]
	$\mathcal{O}(\log N)^{\mathrm{a}}$	**new**				

[p] assuming a pseudo-random function family [i] assuming indivisibility [8] [a] amortized

1.2 Previous Work

Oblivious Priority Queues. Because of their many algorithmic applications, oblivious priority queues have been considered in a number of previous works. We provide an overview of previous oblivious priority queue constructions in Table 1.

Jacob et al. [21] show that a runtime of $\Omega(\log N)$ per operation is necessary for oblivious priority queues. Their lower bound holds even when allowing a constant failure probability and relaxing the obliviousness to statistical or computational security.

The first oblivious priority queue construction due to Toft [32] is perfectly-secure and has an amortized runtime of $\mathcal{O}(\log^2 N)$, but reveals the operation performed and lacks operations to delete or modify arbitrary elements. Subsequent perfectly-secure constructions [26, 19] offer operation-hiding security or support additional operations, but do not improve the suboptimal $\mathcal{O}(\log^2 N)$ runtime. A different line of work considers oblivious priority queues with statistical security. Jafargholi et al. [22] and, subsequently, Shi [29] both present constructions with an optimal $\Theta(\log N)$ runtime. All statistically secure priority queue constructions [33, 23, 22, 29] are randomized; many [33, 23, 29] also rely on tree-based ORAMs (e. g., *Path ORAM* [30] or *Circuit ORAM* [10]) in a non–black-box manner.

Offline ORAMs. Though much of the research focuses on online ORAMs, *offline* ORAMs have been explicitly considered in some previous works [26, 8, 22, 29]. We provide an overview of the best known upper and lower bounds for both online and offline ORAM constructions with perfect, statistical, or computational security in Table 2.

Goldreich and Ostrovsky [16] prove a lower bound on the overhead of $\Omega(\log N)$ for (online) ORAMs with perfect security (assuming indivisibility). This bound also applies to offline ORAMs and constructions with statistical security [8].

There is a generic way to construct offline ORAMs from oblivious priority queues (see Section 3). Via their priority queue construction, Shi [29] obtains an optimal offline ORAM with statistical security for a private memory of constant size. For computational security, the state-of-the-art online ORAM construction [5] is simultaneously the best known offline construction (asymptotically). While the upper bounds for statistical and computational security match the (conjectured) $\Omega(\log N)$ lower bound, prior to our work there remained a gap for perfect security.[2]

[2] Boyle and Naor [8] show how to construct an offline ORAM with overhead $\mathcal{O}(\log N)$: In addition to the access locations, their construction must be given the *operands* of the write operations in advance, i. e., the sequence of values to be written. It thus does not fit our more restrictive Definition 2.

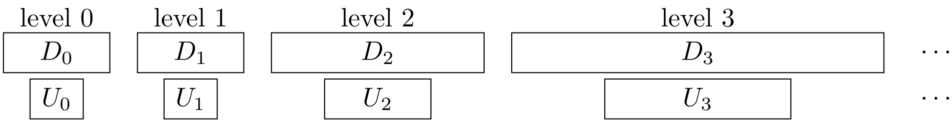

Figure 1 Structure of the oblivious priority queue: Each level $i \in \{0, \ldots, \ell - 1\}$ consists of a down-buffer D_i and an up-buffer U_i half the size of D_i.

1.3 Contributions

Our work provides several contributions to a better understanding of the upper bounds of perfectly-secure oblivious data structures:

- As a main contribution, we present and analyze an oblivious priority queue construction with perfect security. This construction is conceptually simple and achieves the optimal $\Theta(\log N)$ runtime per operation amortized.
  In particular, our construction improves over the previous statistically-secure constructions [22, 29] in that we eliminate the failure probability (perfect security with perfect correctness) and achieve a strictly-logarithmic runtime for $\mathcal{O}(1)$ private memory cells.[3]
- The priority queue implies an optimal $\Theta(\log N)$-overhead offline ORAM with perfect security, closing the gap to statistical and computational security in the offline setting. We show that these bounds hold even for a large number n of operations, i. e., $n = N^{\omega(1)}$.
- We also provide improved external-memory oblivious priority queues: Compared to the I/O-optimal state-of-the-art [22], our cache-aware construction achieves perfect security and only requires a private memory of constant size.
  In the cache-oblivious setting, our construction achieves near-optimal I/O-complexity for perfect security and a private memory of constant size. We are not aware of any previous oblivious priority queues in the cache-oblivious setting.

2 Oblivious Priority Queue from Oblivious Partitioning

An oblivious priority queue data structure maintains up to N elements and must support at least three non-trivial operations prescribed by the abstract data type PriorityQueue:
Insert(k, p). Insert the element $\langle k, p \rangle$ with *priority* p.
Min(). Return the element $\langle k, p_{\min} \rangle$ with the minimal priority $p_{\min}$.
DeleteMin(). Remove the element $\langle k, p_{\min} \rangle$ with minimal priority $p_{\min}$.
We assume that both the key k and the priority p fit in a constant number of memory cells and that the relative order of two priorities p, p' can be determined obliviously in constant time; larger elements introduce an overhead factor in the runtime. To keep the exposition simple, we assume distinct priorities. This assumption can be removed easily, see Section 2.2.

Figure 1 shows the structure of our solution: In a standard data structure layout, it has $\ell \in \Theta(\log N)$ levels of geometrically increasing size. Each level i consists of a *down*-buffer D_i and an *up*-buffer U_i, both of size $\Theta(2^i)$. Insert inserts into the up-buffer U_0 and DeleteMin removes from the down-buffer D_0. Each level i is rebuilt after 2^i operations, moving elements up through the up-buffers and back down through the down-buffers. The main idea guiding the rebuilding is to ensure that all levels $j < i$ can support the next operations until level i is rebuilt; we later formalize this as an invariant for the priority queue (see Lemma 4).

[3] For a private memory of constant size, the construction of Shi [29] requires an additional $\omega(1)$-factor in runtime to achieve a negligible failure probability.

Oblivious Building Blocks. For our construction, we need an algorithm to obliviously permute a given array A of n elements such that the k smallest elements are swapped to the front, followed by the remaining $n - k$ elements. We refer to this problem as *k-selection*.

To obtain an efficient algorithm for k-selection, we use an oblivious modification of the classical (RAM) linear-time selection algorithm [7] as sketched by Lin et al. [25, full version, Appendix E.2]. This reduces k-selection to the *partitioning* problem (also called 1-*bit sorting*). The oblivious k-selection deviates from the classical algorithm in two respects [25]:

- First, it is necessary to ensure that the partitioning step is oblivious. For this, instead of the algorithms proposed by Lin et al., we use the optimal oblivious partitioning algorithm of Asharov et al. [4, Theorem 5.1].[4] This allows us to obtain a linear-time algorithm for k-selection.
- Second, the relative position of the median of medians among the elements cannot be revealed as this would leak information about the input. To address this, Lin et al. propose over-approximating the number of elements and always recursing with approximately $\frac{7n}{10}$ elements.

▶ **Corollary 3** (Oblivious k-Selection via [25, 4])**.** *There is a deterministic, perfectly-secure oblivious algorithm for the k-selection problem with runtime $\mathcal{O}(n)$ for n elements.*

We describe the algorithm in detail and prove its correctness in the full version [31].

Comparison with Jafargholi et al. [22]. Conceptually, our construction is similar to that of Jafargholi et al. [22]: In both constructions, the priority queue consists of levels of geometrically increasing size with lower-priority elements moving towards the smaller levels. Structuring the construction so that larger levels are rebuilt less frequently is a standard data structure technique to amortize the cost of rebuilding.

The main difference lies in the rebuilding itself: In the construction of Jafargholi et al., level i is split into 2^i nodes; overall, the levels form a binary tree. The elements are then assigned to paths in the tree based on their key [22]. While this allows deleting elements by their key efficiently, this inherently introduces the probability of "overloading" certain nodes, reducing the construction to statistical security (with a negligible failure probability).

We instead use k-selection for rebuilding; this allows us to maintain both perfect correctness and security. Unfortunately, this comes at the cost of a more expensive Delete operation: Since we maintain no order on the keys within each level, it is not possible to efficiently delete arbitrary elements by their key. We note that deleting arbitrary elements is not required for our offline ORAM construction.

2.1 Details of the Construction

The priority queue consists of $\ell := \lceil \log_2 N \rceil$ levels, each with a down-buffer D_i of $2^{\max\{1,i\}}$ elements and an up-buffer U_i of $2^{\max\{0,i-1\}} = \frac{|D_i|}{2}$ elements. An element is a pair $\langle k, p \rangle$ of key k and priority p; each buffer is padded with *dummy* elements to hide the number of "real" elements. Initially, all elements in the priority queue are dummy elements. We refer to a buffer containing only dummy elements as *empty*.

The elements are distributed over the levels via a rebuilding procedure: Level i is rebuilt after exactly 2^i operations. Let Δ_i be the remaining number of operations until level i is rebuilt (with $\Delta_i = 2^i$ initially). After each operation, all counters Δ_i are decremented by

[4] Note that Asharov et al. refer to the partitioning problem as *compaction*. We use the term partitioning to stress that all elements of the input are preserved which is necessary for our definition of k-selection.

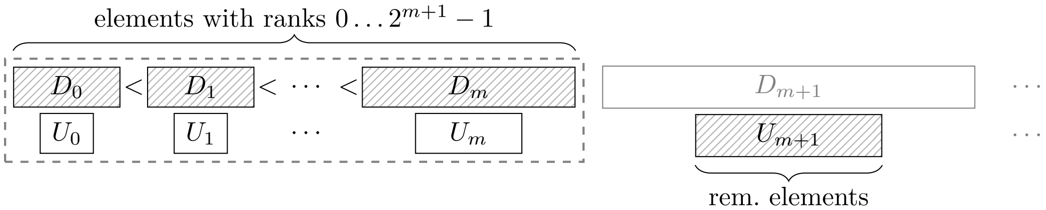

Figure 2 Distribution of the elements when rebuilding level m: The up to 2^{m+1} smallest elements in the levels $0, \dots, m$ are distributed over the down-buffers of the first m levels. The up to 2^m remaining elements are inserted into the (empty) up-buffer U_{m+1} of level $m+1$.

one and all levels i with $\Delta_i = 0$ are rebuilt with REBUILD(m) for $m := \max\{i < \ell \mid \Delta_i = 0\}$; note that $\Delta_i = 0$ if and only if $i \leq m$. The counter Δ_i of each rebuilt level $i \leq m$ is reset to 2^i, so $\Delta_i > 0$ for every level i after each operation.

We will show the correctness of the construction with three invariants (a)–(c):

▶ **Lemma 4** (Invariants). *Before each operation of the priority queue, the following holds:*

(a) *The priority queue contains the correct elements, i. e., $\mathcal{E} = \bigcup_{i<\ell}(U_i \cup D_i)$ where $\mathcal{E}$ denotes the elements that should be contained in the priority queue (with standard semantics).*

(b) *The up-buffer U_0 is empty, i. e., contains exactly one dummy element.*

(c) *For all elements $e := \langle k, p\rangle \in D_i \cup U_i$ with $i \geq 1$, it holds that $\Delta_i \leq \operatorname{rank}(e)$ where $\operatorname{rank}(\langle \cdot, p\rangle) := |\{\langle \cdot, p'\rangle \in \mathcal{E} \mid p' < p\}|$ is the (unique) rank of p in the priority queue.*

The most important invariant (c) guarantees that each level $i \geq 1$ is rebuilt before any of its elements are required for MIN/DELETEMIN in D_0. In turn, this implies that the Δ_i smallest elements potentially required before rebuilding level i are stored in the buffers on levels $0, \dots, i-1$. Formally, this follows since $\Delta_j \leq \Delta_i$ for all $j < i$.

For simplicity of exposition, we ignore the details of the rebuilding step for the time being and discuss how the three priority queue operations can be implemented while maintaining the invariants (a) and (c) of Lemma 4:

INSERT(k, p). The dummy in U_0 is replaced with the new element $\langle k, p\rangle$. After the operation, the priority queue contains the elements $\mathcal{E}' = \mathcal{E} \cup \{\langle k, p\rangle\} = \bigcup_{i<\ell}(U_i \cup D_i)$. Inserting a new element does not decrease the rank of any element while all counters Δ_i decrease; this implies that invariant (c) is maintained.

MIN(). The minimal element $e_{\min}$ with $\operatorname{rank}(e_{\min}) = 0$ must be contained in level 0 since $\Delta_i > 0$ for all $i > 0$ before each operation. Since U_0 is empty, $e_{\min}$ is one of the two elements in D_0. After the operation, the elements $\mathcal{E}' = \mathcal{E} = \bigcup_{i<\ell}(U_i \cup D_i)$ remain the same. Invariant (c) is maintained since the ranks of all elements $e \in \mathcal{E}$ remain unchanged.

DELETEMIN(). For this operation, we replace the minimal element $e_{\min} \in D_0$ with a dummy element. After the operation, the priority queue contains the elements $\mathcal{E}' = \mathcal{E} \setminus \{e_{\min}\} = \bigcup_{i<\ell}(U_i \cup D_i)$. Removing the minimum reduces the rank of all other elements by one, but invariant (c) is maintained since all counters Δ_i also decrease.

For the operation-hiding security, we access memory locations for all three operations but only perform updates for the intended operation. For example, DELETEMIN will access both U_0 and D_0, but only actually overwrite the minimal element in D_0 with a dummy element. We provide pseudocode for the operations in the full version [31].

We now turn to describing REBUILD(m) (Algorithm 1). As shown in Figure 2, this procedure processes all elements in the levels $0, \dots, m$: The non-dummy elements are distributed into $D_0, \dots, D_m, U_{m+1}$ and the up-buffers $U_0, \dots, U_m$ are emptied, i. e., filled

Algorithm 1 Rebuild the levels $0, \dots, m$ in the oblivious priority queue. Let $A \parallel B$ denote the concatenation of two buffers A and B; $A_{0\dots i}$ denotes the concatenation $A_0 \parallel \cdots \parallel A_i$.

1: **procedure** Rebuild(m)
2: KSelect($2^{m+1}, D_{0\dots m} \parallel U_{0\dots m}$) ▷ *move 2^{m+1} smallest elements to $D_{0\dots m}$*
3: **if** m is not the last level **then**
4: $U_{m+1} \leftarrow U_{0\dots m}$ ▷ *copy the elements in $U_{0\dots m}$ to U_{m+1}*
5: $U_{0\dots m} \leftarrow \langle \bot, \dots, \bot \rangle$ ▷ *overwrite $U_{0\dots m}$ with dummy elements*
6: **for** $i \leftarrow m-1, \dots, 0$ **do**
7: KSelect($2^{i+1}, D_{0\dots i+1}$) ▷ *move 2^{i+1} smallest elements to $D_{0\dots i}$*
8: **for** $i \leftarrow 0, \dots, m$ **do**
9: $\Delta_i \leftarrow 2^i$ ▷ *reset counters*

with dummy elements. The down-buffers $D_0, \dots, D_m$ collectively contain up to 2^{m+1} non-dummy elements. Additionally, the up-buffers $U_0, \dots, U_m$ collectively contain up to 2^m non-dummy elements. All these elements are distributed over the buffers $D_0, \dots, D_m$ and U_{m+1} such that

- D_0 contains the two smallest elements (with ranks 0 and 1),
- the other D_i (for $i \leq m$) each contain the elements with ranks $2^i, \dots, 2^{i+1} - 1$,[5] and
- U_{m+1} contains all remaining elements.

For this, we order the elements by their priority p; dummy elements have no priority and are ordered after non-dummy elements.

We can now prove that the overall construction is correct by showing that Rebuild(m) with $m := \max\{i < \ell \mid \Delta_i = 0\}$ maintains the invariants (a)–(c):

Proof of Lemma 4. All invariants trivially hold for the empty priority queue. As described above, the operations Insert, Min, and DeleteMin maintain invariants (a) and (c). After each operation, all counters Δ_i are decremented and the levels i with $\Delta_i = 0$ are rebuilt. We now show that all invariants hold after rebuilding.

Invariants (a) and (b): $\mathcal{E} = \bigcup_{i<\ell}(U_i \cup D_i)$ and U_0 is empty. We first show that prior to Rebuild(m) in each operation where m is not the last level, the up-buffer U_{m+1} is empty. This can be seen by considering the two possible cases:

- If no more than 2^m operation have been performed overall, the level $m+1$ has never been accessed. In this case U_{m+1} is empty since it was empty initially.
- Otherwise, if more than 2^m operations have been performed, the up-buffer U_{m+1} was emptied 2^m operations before by Rebuild(m') for some $m' > m$ (and not accessed since).

This means that by copying the elements in $U_{0\dots m}$ into U_{m+1} (Line 4), only dummy elements are being overwritten and invariant (a) is maintained.

In case m is the last level ($m = \ell - 1$), after Line 2 the up-buffers $U_0, \dots, U_m$ are empty iff there no more than $2^{m+1} = 2^\ell \geq N$ elements in the data structure. This is guaranteed by the capacity bound N. Thus, the up-buffer U_0 is empty after each operation.

[5] Even if there are more than 2^{m+1} elements in the priority queue overall, the buffer D_m may still end up (partially) empty after rebuilding. This is no threat to the correctness, since invariant (c) guarantees that level $m+1$ will be rebuilt before requiring the elements with ranks $\geq 2^m$.

Invariant (c): $\Delta_i \le \mathrm{rank}(e)$ for all $e \in U_i \cup D_i$ with $i \ge 1$. Next, we show that rebuilding maintains the rank invariant for all redistributed elements. Using k-selections to redistribute the elements makes sure that a buffer in level i receives non-dummy elements only if all $D_0, \ldots, D_{i-1}$ have been filled to capacity. Consider any level $i \ge 1$: If an element $e := \langle k, p \rangle$ is redistributed into level i, exactly $2^i = \sum_{j<i} |D_j|$ elements $\langle \cdot, p' \rangle$ with $p' < p$ must have been redistributed into lower levels, so $2^i \le \mathrm{rank}(e)$. Thus, for all non-dummy elements e inserted into a level $i \ge 1$, it holds that $\Delta_i \le 2^i \le \mathrm{rank}(e)$. For elements that remain in a level $i > m$, invariant (c) is trivially maintained. ◀

With this, we obtain our perfectly-secure priority queue construction:

▶ **Theorem 5** (Optimal Oblivious Priority Queue). *There is a deterministic, perfectly-secure oblivious priority queue with capacity N that supports each operation in amortized $\mathcal{O}(\log N)$ time and uses $\mathcal{O}(N)$ space.*

Proof. Apart from the rebuilding, the runtime for Insert, Min, and DeleteMin is constant. The amortized runtime per operation for Rebuild is bounded by

$$\sum_{m=0}^{\ell-1} \frac{T_{\text{KSelect}}(2^{m+1} + 2^m) + \sum_{i=0}^{m-1} T_{\text{KSelect}}(2^{i+2}) + c \cdot 2^m}{2^m}$$
$$\le \sum_{m=0}^{\ell-1} \frac{\mathcal{O}(2^m)}{2^m} \in \mathcal{O}(\ell) = \mathcal{O}(\log N)\,.$$

The space bound follows immediately since all algorithmic building blocks have a linear runtime and the combined size of all up- and down-buffers is linear in N.

By using the deterministic, perfectly-secure algorithm for k-selection (Corollary 3), the obliviousness follows since the access pattern for each operation is a deterministic function of the capacity N and the number of operations performed so far. ◀

Due to the lower bound for oblivious priority queues [21], this runtime is optimal. If the type of operation does not need to be hidden, Min can be performed in constant time since rebuilding is only required for correctness when adding or removing an element. By applying Rebuild$(\ell - 1)$ directly, the priority queue can be initialized from up to N elements in $\mathcal{O}(N)$ time.

2.2 Non-Distinct Priorities

For non-distinct priorities, we want to ensure that ties are broken such that the order of insertion is preserved, i. e., that elements inserted earlier are extracted first. For this, we augment each element with the time-stamp t of the Insert operation and order the elements lexicographically by priority and time-stamp. This only increases the size of each element by a constant number of memory cells and thus does not affect the runtime complexity.

It remains to bound the size of the time-stamp for a super-polynomial number of operations:[6] Here we note that when rebuilding the last level (Rebuild$(\ell - 1)$), we can additionally sort all elements by time-stamp and compress the time-stamps to the range $\{0, \ldots, N-1\}$ (preserving their order). We then assign time-stamps starting with $t = N$ until

[6] Shi [29, Section III.E] also address this issue, but for our amortized construction we can use a simpler approach based on oblivious sorting.

Algorithm 2 Algorithm to perform an operation $\textsc{Op} \in \{\textsc{Read}, \textsc{Write}\}$ in the offline ORAM at the access location i; v' is the value to be written ($v' = \bot$ for $\textsc{Op} = \textsc{Read}$). The time-stamp t is incremented after each access (with $t = 1$ initially).

1: **procedure** $\textsc{Access}(\textsc{Op}, i, v')$
2: $\quad \langle v, t_{\text{next}} \rangle \leftarrow Q.\textsc{Min}()$
3: $\quad Q.\textsc{DeleteMin}()$ iff $t_{\text{next}} = t$; perform a dummy operation iff $t_{\text{next}} \neq t$
4: $\quad v \leftarrow \begin{cases} v & \text{iff } \textsc{Op} = \textsc{Read} \wedge t_{\text{next}} = t, \\ v_{\text{default}} & \text{iff } \textsc{Op} = \textsc{Read} \wedge t_{\text{next}} \neq t, \\ v' & \text{iff } \textsc{Op} = \textsc{Write} \end{cases}$
5: $\quad Q.\textsc{Insert}(v, \mathcal{T}[t-1])$; $t \leftarrow t + 1$ $\qquad \triangleright\ \mathcal{T}[t-1] = \tau_t$
6: $\quad$ **return** v

the last level is rebuilt again. This ensures that $2N$ is an upper bound for the time-stamps, so $\mathcal{O}(\log N)$ bits suffice for each time-stamp. With an (optimal) oblivious $\mathcal{O}(n \log n)$-time sorting algorithm [2], the additional amortized runtime for sorting is bounded by

$$\frac{1}{2^{\ell-1}} \cdot T_{\textsc{Sort}}(N) \in \Theta(\log N) \tag{2}$$

and does not affect the overall runtime complexity.

3 Offline ORAM from Oblivious Priority Queues

As mentioned in the introduction, an offline ORAM is an oblivious array data structure given the sequence $I = \langle i_1, \ldots, i_n \rangle$ of access locations in advance. While an offline ORAM may pre-process I to perform the operations more efficiently afterward, the data structure must still adhere to Definition 1, i. e., the memory probes must be independent of the values in I.

An offline ORAM can be constructed from any oblivious priority queue using a technique similar to *time-forward processing* [12]. We describe our construction for N memory cells below: In Section 3.1 we show how to realize $\textsc{Read}(i_t)$ and $\textsc{Write}(i_t, v'_t)$; there we assume that the time τ_t at which the index i_t is accessed next is known. In Section 3.2 we then show how to pre-process I to derive these values τ_t.

Jafargholi et al. [22] describe an alternative offline ORAM construction. We simplify the construction by decoupling the information τ_t from the values written to the offline ORAM.

3.1 Online Phase: Processing the Operations

For the offline ORAM with N cells, initialize a priority queue Q with capacity N; the annotations $\mathcal{T} = \langle \tau_1, \ldots, \tau_n \rangle$ as well as the current time t are stored alongside the priority queue. The procedure for processing the t-th operation $\textsc{Read}(i_t)$ or $\textsc{Write}(i_t, v'_t)$ is shown in Algorithm 2. Some fixed value v_{default} is used as the initial value of all ORAM cells.

It is easy to verify that the resulting construction is correct given $\mathcal{T}$ and oblivious given a perfectly-secure priority queue Q. For the array access $\mathcal{T}[t-1]$ in Line 5, note that t is the number of the current operation, so the access can be performed "in the clear" without a linear scan; this simplifies the construction w. r. t. Jafargholi et al. [22].

3.2 Offline Phase: Pre-Processing

To obtain the annotations $\mathcal{T} = \langle \tau_1, \ldots, \tau_n \rangle$ we now describe how to pre-process the sequence $I = \langle i_1, \ldots, i_n \rangle \in \{0, \ldots, N-1\}^n$ of memory access locations.

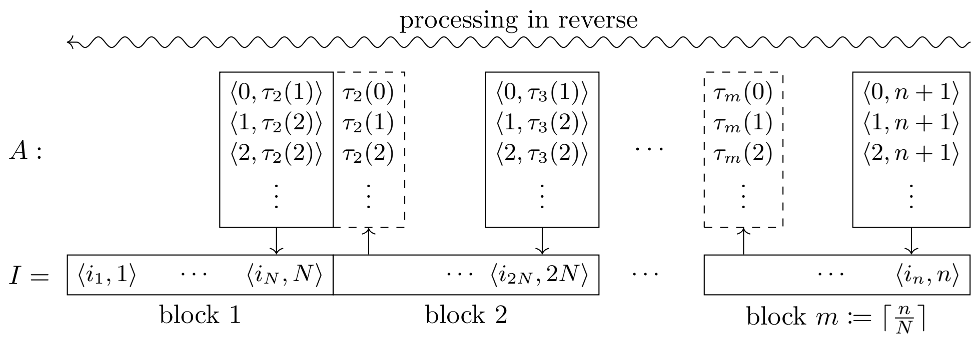

Figure 3 Pre-processing the sequence of memory access locations I when n is super-polynomial in N. In this figure, $\langle i, t\rangle$ denotes a tuple of index i and time-stamp t while $\tau_j(i)$ denotes the time-stamp at which the index i is accessed next in block j.

The basic pre-processing proceeds as follows:

1. Annotate each index i_t with the time-stamp t.
2. Obliviously sort the indices I lexicographically by i_t and t.
3. Scan over indices in reverse, keeping track of the index i and the time-stamp t, and annotate each index i_t with the time-stamp τ_t it is accessed next (or some value larger than n if there is no next access).
4. Obliviously sort the indices I by t and discard everything but the annotations τ.

This can be done in amortized $\mathcal{O}(\log n)$ time per index with an $\mathcal{O}(n \log n)$-time oblivious sorting algorithm [2] and results in the annotations $\mathcal{T} = \langle \tau_1, \dots, \tau_n\rangle$.

However, when the number of operations n is super-polynomial in the capacity N, i. e., when $n \in \omega(N^c)$ for all constants c, the time per index exceeds the optimal runtime of $\mathcal{O}(\log N)$. In this case, the pre-processing needs to be performed more carefully as shown in Figure 3 to maintain the amortized runtime of $\mathcal{O}(\log N)$: We divide the sequence I into blocks of size N. Additionally, we maintain an auxiliary block A with – for each index $i \in \{1, \dots, N\}$ – the time-stamp τ at which index i is accessed next. Initially (for the last block), we initialize the time-stamp τ for each index i to some value greater than n.

The time-stamps τ_t are then determined block by block, from the last to the first. When processing each block, we update the time-stamps τ in A for the block processed next. For this, we can process the $\mathcal{O}(N)$ elements of each block (and A) as described above.

Since we process $\mathcal{O}(N)$ elements for each of the $\mathcal{O}(\lceil \frac{n}{N} \rceil)$ blocks by sorting and scanning, the pre-processing has a runtime of $\mathcal{O}(n \log N)$ overall. This maintains the desired runtime of $\mathcal{O}(\log N)$ per operation amortized.

With our priority queue construction from Section 2, we obtain the following:

▶ **Theorem 6** (Optimal Offline ORAM). *There is a deterministic, perfectly-secure offline ORAM with capacity N that has amortized $\mathcal{O}(\log N)$ overhead and uses $\mathcal{O}(N + n)$ space.*

Note that in contrast to the optimal statistically-secure constructions [22, 29], our offline ORAM maintains security and correctness for operation sequences of arbitrary length, e. g., when n is super-polynomial in N.

4 External-Memory Oblivious Priority Queue

In many applications of oblivious algorithms and data structures, e. g., for outsourced storage and for trusted computing in the presence of cache hierarchies, access to the main memory incurs high latencies. In these applications, the complexity of an algorithm is more

Table 3 Best known I/O upper bounds for obliviously partitioning n elements.

	cache-aware		*cache-agnostic*	
statistical security	$\mathcal{O}(\lceil \frac{n}{B} \rceil)^{\text{t, w}}$	[17]	$\mathcal{O}(\lceil \frac{n}{B} \rceil)^{\text{t, w}}$	[25]
perfect security	$\mathcal{O}(\lceil \frac{n}{B} \rceil \log_M n)$	[25]	$\mathcal{O}(\lceil \frac{n}{B} \rceil \log_M n)$	[25]
	$\mathcal{O}(\lceil \frac{n}{B} \rceil)$	**new, via [4]**	$\mathcal{O}(\lceil \frac{n}{B} \rceil \log\log_M n)^{\text{t}}$	**new, via [4]**

$^{\text{t}}$ assuming a *tall cache* ($M \geq B^{1+\varepsilon}$) $^{\text{w}}$ assuming a *wide cache-line* ($B \geq \log^{\varepsilon} n$)

appropriately captured by the number of cache misses. This motivates the study of oblivious algorithms in the *external-memory* [1] and *cache-oblivious* [14] models; in this section we refer to these as *cache-aware* and *cache-agnostic* algorithms.

In this section, we instantiate I/O-efficient variants of our priority queue construction with perfect security and a private memory of constant size. For this, we sketch how to obtain I/O-efficient partitioning algorithms with perfect security in Section 4.1. We then analyze the I/O-efficiency of our priority queue construction in Section 4.2.

External-Memory Oblivious Algorithms. In cache-aware and cache-agnostic models, the CPU operates on the data stored in an *internal memory* (*cache*) of M memory words. Blocks (*cache-lines*) of B memory words can be transferred between the internal and a large external memory (*I/O operations*). The number of these I/O operations, depending on the problem size n as well as M and B, is the primary performance metric for external algorithms [1].

Cache-aware algorithms depend on the parameters M and B and explicitly issue the I/O operations. In contrast, cache-agnostic algorithms are unaware of the parameters M and B; here the internal memory is managed "automatically" through a *replacement policy* [14]. We assume an optimal replacement policy and a *tall cache*, i. e., $M \geq B^{1+\varepsilon}$ for a constant $\varepsilon > 0$; both are standard assumptions [14, 9].

For *oblivious external-memory* algorithms, we apply the Definition 1 to cache-aware and cache-agnostic algorithms. In both cases we assume that the internal memory is conceptually distinct from the constant-size private memory. That is, we guarantee that memory words both in the internal memory and within a block are accessed in an oblivious manner (*strong obliviousness* [9]). For this reason, our security definition remains unchanged. Note that this implies that the block access pattern is also oblivious, i. e., independent of the operations/inputs as in Definition 1.

Previous Work. While there is a line of research explicitly considering cache-aware [17, 18] and cache-agnostic [9, 25] oblivious algorithms, most works on oblivious algorithms consider internal algorithms with runtime and bandwidth overhead as performance metrics. To the best of our knowledge, cache-agnostic oblivious priority queues have not been explicitly considered in the literature. Implicitly, I/O-efficiency is sometimes [21, 22] treated through parameters: An oblivious algorithm with $(B \cdot w)$-width memory words and $M \cdot w$ bits of private memory can equivalently be stated as an oblivious external-memory algorithm with M words of internal memory and blocks of size B. We note that this re-parameterization does not allow distinguishing internal and private memory and that the resulting algorithms are inherently cache-aware.

Algorithm 3 Cache-aware oblivious partitioning algorithm. For simplicity, we assume $m \geq 2$.

procedure $\textsc{CacheAwarePartition}_P(A)$
  conceptually partition A into $m := \lceil \frac{|A|}{B} \rceil$ blocks G_i of B consecutive elements each (where the last block may have fewer elements)
  for $i \leftarrow 0, \dots, m-1$ **do** $\textsc{Partition}_P(G_i)$
  for $i \leftarrow 1, \dots, m-2$ **do** $\textsc{PurifyHalf}_P(G_{i-1}, G_i)$ ▷ *consolidate the blocks*
  $\textsc{Partition}_{P' \colon X \mapsto P(X[0])}(\langle G_0, \dots, G_{m-3} \rangle)$ ▷ *apply* PARTITION *to the blocks*
  $\textsc{Reverse}(G_{m-1})$; $\textsc{PartBitonic}_P(G_{m-2} \parallel G_{m-1})$
  $\textsc{Reverse}(G_{m-2} \parallel G_{m-1})$; $\textsc{PartBitonic}_P(G_0 \parallel \cdots \parallel G_{m-1})$

This equivalence allows us to restate upper and lower bounds in terms of external-memory algorithms: Jacob et al. [21] show that $\Omega(\frac{1}{B} \log \frac{N}{M})$ I/O operations amortized are necessary for a cache-aware oblivious priority queue; this also applies to cache-agnostic oblivious priority queues.[7] This bound is matched by Jafargholi et al. [22], but the construction is cache-aware, requires $\Omega(\log N)$ words of private memory, and is randomized with statistical security.

The optimal internal partitioning algorithm [4] has – due to the use of expander graphs – an oblivious, but highly irregular access pattern and is thus not I/O-efficient. There are external oblivious partitioning algorithms [17, 25], but they are either only statistically secure or inefficient. We provide an overview of existing partitioning algorithms in Table 3.

4.1 External-Memory Oblivious Partitioning

For I/O-efficient instantiations of our priority queue, we need I/O-efficient partitioning algorithms. For this reason, we show how to construct an optimal cache-aware and a near-optimal cache-agnostic oblivious partitioning algorithm, respectively, with perfect security. Remember that for partitioning with a predicate P, we need to permute the elements such that all elements x with $P(x) = 0$ precede those with $P(x) = 1$.

We mainly rely on the optimal (internal) oblivious partitioning algorithm [4, Theorem 5.1] (PARTITION) and standard external-memory techniques. We also use oblivious building blocks from the previous work by Lin et al. [25, full version, Appendix C.1.2]:

$\textsc{PurifyHalf}_P(A, B)$. This procedure is given two partitioned blocks A and B with $|A| = |B|$ and permutes the elements such that A is *pure*, i. e., either only consists of elements x with $P(x) = 0$ or only consists of elements with $P(x) = 1$, and B is again partitioned.

$\textsc{PartBitonic}_P(A)$. This procedure is given a *bitonically partitioned* [25] array A, i. e., an array where all elements x with $P(x) = 1$ or all elements with $P(x) = 0$ are consecutive, and partitions A.

Both building blocks are deterministic with perfect security, cache-agnostic, and have a linear runtime of $\mathcal{O}(\lceil \frac{n}{B} \rceil)$ [25].

Our cache-aware partitioning algorithm is shown in Algorithm 3. The idea is to split A into blocks of size B, partition each block, and then apply the internal partitioning algorithm to the blocks.

[7] In contrast, $\Theta(\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/O operations amortized are sufficient for non-oblivious priority queues [3]. Here, the base of the logarithm is not constant but depends on the parameters M, B.

Algorithm 4 Cache-agnostic oblivious partitioning algorithm for $M \geq B^{1+\varepsilon}$. We assume $m \geq 2$.

1: **procedure** $\text{CacheAgnosticPartition}_P(A)$
2:   **if** $n \leq 4$ **then**
3:     compact A via oblivious sorting
4:   **else**
5:     conceptually partition A into $m := \lceil \frac{|A|}{k} \rceil$ groups G_i of $k := \lceil \sqrt[1+\varepsilon]{|A|} \rceil$ consecutive elements (where the last group may have fewer elements)
6:     **for** $i \leftarrow 0, \dots, m-1$ **do** $\text{CacheAgnosticPartition}_P(G_i)$
7:     **for** $i \leftarrow 1, \dots, m-2$ **do** $\text{PurifyHalf}_P(G_{i-1}, G_i)$ ▷ *consolidate the groups*
8:     $\text{Partition}_{P' \colon X \mapsto P(X[0])}(\langle G_0, \dots, G_{m-3} \rangle)$ ▷ *apply* Partition *to the groups*
9:     $\text{Reverse}(G_{m-1})$; $\text{PartBitonic}_P(G_{m-2} \parallel G_{m-1})$
10:     $\text{Reverse}(G_{m-2} \parallel G_{m-1})$; $\text{PartBitonic}_P(G_0 \parallel \cdots \parallel G_{m-1})$

▶ **Corollary 7** (Optimal Cache-Aware Oblivious Partitioning via [4, 25])**.** *There is a cache-aware, deterministic, perfectly-secure oblivious partitioning algorithm that requires $\mathcal{O}(\lceil \frac{n}{B} \rceil)$ I/O operations for n elements.*

Proof. For the correctness, note that after Line 4 all blocks except G_{m-2} and G_{m-1} are pure. By applying the internal partitioning algorithm to the blocks in Line 5, all 0-blocks are swapped to the front. The partitioning is completed by first merging the partitions G_{m-2} and G_{m-1} in Line 6 and then merging both with the rest of the blocks in A.

The partitioning of each individual block is performed in internal memory and thus requires $\mathcal{O}(\lceil \frac{n}{B} \rceil)$ I/O operations overall. The consolidation and merging of the partitions can also be performed with $\mathcal{O}(\lceil \frac{n}{B} \rceil)$ I/O operations [25]. For the partitioning in Line 5, the I/O-efficiency follows from the construction of the internal partitioning algorithm [4, Theorem 5.1]: The algorithm operates in a "balls-in-bins"-manner, i.e., the elements are treated as indivisible. The algorithm performs a linear number of operations on $\lfloor \frac{n}{B} \rfloor$ elements, where each element has size $\mathcal{O}(B)$. This leads to an I/O complexity of $\mathcal{O}(\lceil \frac{n}{B} \rceil)$ overall.

The obliviousness follows since the access pattern for each operation is a deterministic function of the input size $n := |A|$. ◀

For the cache-agnostic partitioning algorithm, the elements can be processed similarly. Here the parameter B is unknown, so the idea is to recursively divide into smaller groups until a group has size $\leq B$. The resulting algorithm is shown in Algorithm 4.

▶ **Corollary 8** (Cache-Agnostic Oblivious Partitioning via [4, 25])**.** *Assuming a tall cache of size $M \geq B^{1+\varepsilon}$ for a constant $\varepsilon > 0$, there is a cache-agnostic, deterministic, perfectly-secure oblivious partitioning algorithm that requires $\mathcal{O}(\lceil \frac{n}{B} \rceil \log \log_M n)$ I/O operations for n elements.*

Proof. The correctness of the base case is obvious. For the recursive case, the algorithm proceeds as the cache-aware Algorithm 3 above, so the correctness can be seen as in Corollary 7.

For the I/O complexity of Line 8, we distinguish two cases:

$k \leq B$. In this case $B \geq k \geq \sqrt[1+\varepsilon]{n}$, so $n \leq B^{1+\varepsilon} \leq M$ with the tall cache assumption. This means that the problem instance fits in the internal memory, and the step thus has an I/O complexity of $\mathcal{O}(\lceil \frac{n}{B} \rceil)$.

$k > B$. Here we can rely on the same insight as for the cache-aware partitioning, i.e., that the internal algorithm performs $\mathcal{O}(\frac{n}{k})$ operations on elements of size $k \geq B$. This leads to an I/O complexity of $\mathcal{O}(\frac{n}{k} \cdot \lceil \frac{k}{B} \rceil) = \mathcal{O}(\frac{n}{B})$.

The other steps have an I/O complexity of $\mathcal{O}(\lceil \frac{n}{B} \rceil)$ as in the cache-aware algorithm.

On depth i of the recursion tree, the instance size is

$$n_i := \sqrt[(1+\varepsilon)^i]{n} \qquad \text{so that on depth} \qquad i \geq \log_{1+\varepsilon} \frac{\log n}{\log M} \in \Theta(\log \log_M n)$$

the instances fit in the internal memory and no further I/O operations are required for the recursion. This leads to an I/O complexity of $\mathcal{O}(\lceil \frac{n}{B} \rceil \log \log_M n)$ overall.

As in Corollary 7, the obliviousness follows since the access pattern for each operation is a deterministic function of the input size $n := |A|$. ◀

4.2 Analysis of the External-Memory Oblivious Priority Queue

As for the internal algorithm introduced in Section 2, we can obtain efficient external oblivious algorithms for k-selection via partitioning [25, full version, Appendix E.2]. We thus obtain cache-aware and cache-agnostic algorithms for k-selection with the same asymptotic complexities as the partitioning algorithms described above.

With these external algorithms, we can analyze the construction described in Section 2 in the cache-aware and cache-agnostic settings:

▶ **Theorem 9** (External-Memory Oblivious Priority Queues)**.** *There are deterministic, perfectly-secure oblivious priority queues with capacity N that support each operation with I/O complexity $\mathcal{O}(\frac{1}{B} \log \frac{N}{M})$ amortized (cache-aware) or $\mathcal{O}(\frac{1}{B} \log \frac{N}{M} \log \log_M N)$ amortized (cache-agnostic), respectively.*

Proof. We prove the theorem via a slightly more general statement: Assuming the existence of a deterministic, perfectly-secure k-selection algorithm with I/O complexity $O_{\text{KSelect}}(n) \in \Omega(\frac{n}{B})$, there is a deterministic, perfectly-secure priority queue with capacity N that supports each operation with I/O complexity $\mathcal{O}(\frac{O_{\text{KSelect}}(3N)}{N} \log \frac{N}{M})$ amortized.

Since the external k-selection algorithms are functionally equivalent to the internal algorithm and oblivious, the correctness and obliviousness follows from Theorem 5. For the I/O complexity, note that the first $j := \log_2 M - \mathcal{O}(1)$ levels of the priority queue fit into the M cells of the internal memory. The operations Insert, Min, and DeleteMin only operate on D_0 and U_0, so they do not require additional I/O operations.

It remains to analyze the I/O complexity of rebuilding the data structure. For this, we only need to consider rebuilding the levels $m \geq j$ since rebuilding levels $m < j$ only operates on the internal memory. When rebuilding a level $m \geq j$, the levels $i < j$ can be stored to and afterward retrieved from the external memory with $\mathcal{O}(\frac{M}{B})$ I/O operations. Assuming optimal page replacement, the amortized number of I/O operations for rebuilding is bounded by

$$\sum_{m=j}^{\ell-1} \frac{O_{\text{KSelect}}(2^{m+1} + 2^m) + \sum_{i=0}^{m-1} O_{\text{KSelect}}(2^{i+2}) + c \cdot \frac{2^m}{B}}{2^m} + \underbrace{\mathcal{O}\left(\frac{M}{B \cdot 2^j}\right)}_{\text{levels} < j}$$

$$\leq \sum_{m=j}^{\ell-1} \frac{\mathcal{O}(O_{\text{KSelect}}(3 \cdot 2^m))}{2^m} + \mathcal{O}\left(\frac{1}{B}\right) \in \mathcal{O}\left((\ell - j) \cdot \frac{O_{\text{KSelect}}(3N)}{N}\right)$$

$$= \mathcal{O}\left(\frac{O_{\text{KSelect}}(3N)}{N} \log \frac{N}{M}\right).$$

With the cache-aware k-selection via Corollary 7 and the cache-agnostic k-selection via Corollary 8, we obtain the claimed I/O complexities. ◀

With this, we obtain an optimal cache-aware oblivious priority queue and a near-optimal cache-agnostic oblivious priority queue, both deterministic and with perfect security. Using a cache-agnostic, perfectly-secure oblivious sorting algorithm with (expected) I/O complexity $\mathcal{O}(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B})$ [9][8], we can apply the same construction as in Section 3 to obtain external offline ORAMs with the same (expected) I/O complexities as in Theorem 9. We exploit that the construction is a combination of sorting, linear scans, and time-forward processing.

5 Conclusion and Future Work

In this paper, we show how to construct an oblivious priority queue with perfect security and (amortized) logarithmic runtime. While the construction is simple, it improves the state-of-the-art for perfectly-secure priority queues, achieving the optimal runtime. The construction immediately implies an optimal offline ORAM with perfect security. We extend our construction to the external-memory model, obtaining optimal cache-aware and near-optimal cache-agnostic I/O complexities.

Future Work. The optimal perfectly-secure partitioning algorithm [4] has enormous constant runtime factors (in the order of $\gg 2^{111}$ [13]) due to the reliance on bipartite expander graphs.[9] Nevertheless, our construction can also be implemented efficiently in practice – albeit at the cost of an $\mathcal{O}(\log N)$-factor in runtime – by relying on merging [6] (instead of k-selection via linear-time partitioning). We leave comparing such a practical variant to previous protocols as a future work.

On the theoretical side, a main open problem is to obtain a perfectly-secure oblivious priority queue supporting deletions (of arbitrary elements) in optimal $\mathcal{O}(\log N)$ time. An additional open problem is the de-amortization of the runtime complexity. Considering external oblivious algorithms, an open problem is to close the gap on cache-oblivious partitioning, i. e., remove the remaining $\mathcal{O}(\log \log_M n)$ factor in I/O complexity for perfectly-secure algorithms. We consider all of these interesting problems for future works.

References

1 Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988. doi:10.1145/48529.48535.

2 Miklós Ajtai, Jánoss Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 1–9. ACM, 1983. doi:10.1145/800061.808726.

3 Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. Cache-oblivious priority queue and graph algorithm applications. In John H. Reif, editor, *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing*, pages 268–276. ACM, 2002. doi:10.1145/509907.509950.

4 Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. *Journal of the ACM*, 70(1):4:1–4:70, 2022. doi:10.1145/3566049.

[8] Chan et al. [9] only describe a *statistically-secure* sorting algorithm that works by randomly permuting and then sorting the elements. Since the failure can only occur when permuting, can be detected, and leaks nothing about the input, we can repeat the permuting step until it succeeds [11, Section 3.2.2]. This leads to a perfectly-secure sorting algorithm with the same complexity in expectation.

[9] The algorithm of Dittmer and Ostrovsky [13] has lower constant runtime factors, but is randomized and only achieves statistical security due to a (negligible) failure probability.

5 Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, and Elaine Shi. Oblivious RAM with worst-case logarithmic overhead. *Journal of Cryptology*, 36(2):7, 2023. doi:10.1007/s00145-023-09447-5.

6 Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. ACM, 1968. doi:10.1145/1468075.1468121.

7 Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973. doi:10.1016/S0022-0000(73)80033-9.

8 Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In Madhu Sudan, editor, *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, pages 357–368. ACM, 2016. doi:10.1145/2840728.2840761.

9 T.-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Cache-oblivious and data-oblivious sorting and applications. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2201–2220. SIAM, 2018. doi:10.1137/1.9781611975031.143.

10 T.-H. Hubert Chan and Elaine Shi. Circuit OPRAM: Unifying statistically and computationally secure ORAMs and OPRAMs. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography*, volume 10678 of *Lecture Notes in Computer Science*, pages 72–107. Springer, 2017. doi:10.1007/978-3-319-70503-3_3.

11 T.-H. Hubert Chan, Elaine Shi, Wei-Kai Lin, and Kartik Nayak. Perfectly oblivious (parallel) RAM revisited, and improved constructions. In Stefano Tessaro, editor, *2nd Conference on Information-Theoretic Cryptography (ITC 2021)*, volume 199 of *LIPIcs*, pages 8:1–8:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITC.2021.8.

12 Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In Kenneth L. Clarkson, editor, *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149. SIAM, 1995. URL: http://dl.acm.org/citation.cfm?id=313651.313681.

13 Samuel Dittmer and Rafail Ostrovsky. Oblivious tight compaction in $O(n)$ time with smaller constant. In Clemente Galdi and Vladimir Kolesnikov, editors, *Security and Cryptography for Networks*, volume 12238 of *Lecture Notes in Computer Science*, pages 253–274. Springer, 2020. doi:10.1007/978-3-030-57990-6_13.

14 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297. IEEE, 1999. doi:10.1109/SFFCS.1999.814600.

15 D. Gale and L. S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962. doi:10.1080/00029890.1962.11989827.

16 Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996. doi:10.1145/233551.233553.

17 Michael T. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In Rajmohan Rajaraman and Friedhelm Meyer auf der Heide, editors, *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 379–388. ACM, 2011. doi:10.1145/1989493.1989555.

18 Michael T. Goodrich and Joseph A. Simons. Data-oblivious graph algorithms in outsourced external memory. In Zhao Zhang, Lidong Wu, Wen Xu, and Ding-Zhu Du, editors, *Combinatorial Optimization and Applications*, volume 8881 of *Lecture Notes in Computer Science*, pages 241–257. Springer, 2014. doi:10.1007/978-3-319-12691-3_19.

19 Atsunori Ichikawa and Wakaha Ogata. Perfectly secure oblivious priority queue. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E106.A(3):272–280, 2023. doi:10.1587/transfun.2022CIP0019.

20 Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *19th Annual Network and Distributed System Security Symposium*. The Internet Society, 2012. URL: https://www.ndss-symposium.org/ndss2012/access-pattern-disclosure-searchable-encryption-ramification-attack-and-mitigation.

21 Riko Jacob, Kasper Green Larsen, and Jesper Buus Nielsen. Lower bounds for oblivious data structures. In Timothy M. Chan, editor, *Proceedings of the 2019 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2439–2447. SIAM, 2019. doi:10.1137/1.9781611975482.149.

22 Zahra Jafargholi, Kasper Green Larsen, and Mark Simkin. Optimal oblivious priority queues. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2366–2383. SIAM, 2021. doi:10.1137/1.9781611976465.141.

23 Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology — ASIACRYPT 2014*, volume 8874 of *Lecture Notes in Computer Science*, pages 506–525. Springer, 2014. doi:10.1007/978-3-662-45608-8_27.

24 Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology — CRYPTO 2018*, volume 10992 of *Lecture Notes in Computer Science*, pages 523–542. Springer, 2018. doi:10.1007/978-3-319-96881-0_18.

25 Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. Can we overcome the $n \log n$ barrier for oblivious sorting? In Timothy M. Chan, editor, *Proceedings of the 2019 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2419–2438. SIAM, 2019. doi:10.1137/1.9781611975482.148.

26 John C. Mitchell and Joe Zimmerman. Data-oblivious data structures. In Ernst W. Mayr and Natacha Portier, editors, *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*, volume 25 of *LIPIcs*, pages 554–565. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2014. doi:10.4230/LIPIcs.STACS.2014.554.

27 Arup Mondal, Priyam Panda, Shivam Agarwal, Abdelrahaman Aly, and Debayan Gupta. Fast and secure oblivious stable matching over arithmetic circuits. Cryptology ePrint Archive, Paper 2023/1789, 2023. URL: https://eprint.iacr.org/2023/1789.

28 Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *Topics in Cryptology — CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006. doi:10.1007/11605805_1.

29 Elaine Shi. Path oblivious heap: Optimal and practical oblivious priority queue. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 842–858. IEEE, 2020. doi:10.1109/SP40000.2020.00037.

30 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 Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 299–310. ACM, 2013. doi:10.1145/2508859.2516660.

31 Thore Thießen and Jan Vahrenhold. Optimal offline ORAM with perfect security via simple oblivious priority queues, 2024. doi:10.48550/arXiv.2409.12021.

32 Tomas Toft. Secure datastructures based on multiparty computation. Cryptology ePrint Archive, Paper 2011/081, 2011. URL: https://eprint.iacr.org/2011/081.

33 Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 215–226. ACM, 2014. doi:10.1145/2660267.2660314.