T2R2 東京科学大学 リサーチリポジトリ
Science Tokyo Research Repository

論文 / 著書情報
Article / Book Information

Title	Perfectly Secure Oblivious Priority Queue
Authors	Atsunori Ichikawa, Wakaha Ogata
Citation	IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Vol. E106-A, Issue 3, pp. 272-280
Pub. date	2023, 3
Copyright	Copyright(C)2023 IEICE

Powered by T2R2 (Science Tokyo Research Repository)

PAPER *Special Section on Cryptography and Information Security*

Perfectly Secure Oblivious Priority Queue

Atsunori ICHIKAWA[†a)], ***Nonmember and*** **Wakaha OGATA**[††], ***Member***

SUMMARY An Oblivious Priority Queue (OPQ) is a cryptographic primitive that enables a client to outsource its data to a dishonest server, and also to securely manage the data according to a priority queue algorithm. Though the first OPQ achieves perfect security, it supports only two operations; Inserting an element and extracting the top-priority element, which are the minimal requirement for a priority queue. In addition, this OPQ allows an adversary to observe operations in progress, which leaks the exact number of elements in the data structure. On the other hand, there are many subsequent works for OPQs that implement additional operations of a priority queue, hide the running operations, and improve efficiency. Though the recent works realize optimal efficiency, all of them achieve only statistical or computational security. Aiming to reconcile perfect security of the first OPQ with all functions (including the operation hiding) supported by recent OPQs, we construct a novel perfectly secure OPQ that can simulate the following operations while hiding which one is in progress; Inserting an element, extracting the top-priority one, deleting an element, and modifying the priority of an element. The efficiency of our scheme is $O(\log^2 N)$, which is larger than that of the best known statistically secure OPQ but is the same as the known perfectly secure scheme.

key words: *oblivious data structure, oblivious priority queue, perfect security*

1. Introduction

Oblivious Data Structures (ODSs) are cryptographic primitives that allow a client to store its data to an untrusted server and to operate the data in secret. They are used not only for a secure cloud system but also for secure graph algorithms [1], [2] and an oblivious streaming sampler for large-scale federated learning [3] in the context of secure computation.

ODSs are also often described as a computation model in which a trusted CPU manages a data structure in dishonest storage. In contrast to plain (non-oblivious) data structures, in ODS algorithms, the CPU should perform additional storage accesses to obfuscate the *access patterns* that contain information about data and its access structure. Thus, efficiency of an ODS is measured by its *total work* (or simply *work*), which indicates the number of elements that the CPU interacts with the storage to perform one ODS operation.

Specifically, an *Oblivious Priority Queue* (OPQ) is one of the most popular ODSs that obliviously simulates a *priority queue* (PQ). A *queue* is a first-in first-out data structure, and a PQ is a queue that allows data elements to be assigned a *priority*, and the elements can be retrieved in order of priority.

Through this work, we consider a PQ that supports the following four operations:

- **Inserting** an element with its *key* that indicates the priority of the element.
- **Extracting** the top-priority element, i.e., the first inserted element among ones with the smallest key in the PQ.
- **Deleting** a specified element of the PQ.
- **Modifying** the key of a specified element.

1.1 Related Works

Oblivious RAMs. An Oblivious RAM (ORAM) was first introduced by Goldreich and Ostrovsky [4]–[6] to simulate a *Random Access Machine* (RAM) while hiding access patterns. It can convert any data structure represented as a RAM program to an ODS, due to its obliviousness of access patterns.

Asharov et al. [7] proposed a computationally secure ORAM that has optimal $O(\log N)$ total work per access, where N is the size of a program. Since this ORAM simulates any program of size N and of running time T in $O(T \log N)$ total work, it can simulate a PQ (that has running time $O(\log N)$ for the capacity N of the PQ) obliviously in $O(\log^2 N)$ work per operation. On the other hand, Chan et al. [8], [9] introduced an ORAM that perfectly hides its access pattern in $O(\log^3 N/\log\log N)$ total work per access, which can simulate a PQ in $O(\log^4 N/\log\log N)$.

Oblivious Priority Queues. The first OPQ, proposed by Toft [10], supports only two operations, the *insertion* and *extraction*, each of which requires (amortized) $O(\log^2 N)$ total work where N is the capacity of the OPQ. His OPQ is perfectly secure but leaks the operation in progress. I.e., it is free from probabilistic data structure corruption and leakage of access patterns *in each operation*, but the adversary can observe whether the insertion or extraction is performed now *in a sequence of operations*.

Subsequently, Wang et al. [1] and Keller and Scholl [2] each constructed an OPQ with the same communication cost as Toft's OPQ, but which can hide the running operations.

Manuscript received March 15, 2022.
Manuscript revised June 17, 2022.
Manuscript publicized August 23, 2022.

[†]The author is with NTT Social Informatics Laboratories, Musashino-shi, 180-8585 Japan.
[††]The author is with Tokyo Institute of Technology, Tokyo, 152-8550 Japan.
a) E-mail: atsunori.ichikawa.nf@hco.ntt.co.jp

DOI: 10.1587/transfun.2022CIP0019

Copyright © 2023 The Institute of Electronics, Information and Communication Engineers

Table 1 A comparison table for the known OPQs and ours. N denotes the capacity of OPQs.

Scheme	Total work	Operations	Operation hiding	Security
Toft11 [10]	$O(\log^2 N)$	Insert, ExtractMin	✗	Perfect
WNL+14 [1]	$O(\log^2 N)$	Insert, ExtractMin	✓	Statistical
KS14 [2]	$O(\log^2 N)$	Insert, ExtractMin, ModKey	✓	Statistical
Shi20 [3]	$O(\log N)$	Insert, ExtractMin, Delete, ModKey	✓	Statistical
JLS21 [11]	$O(\log N)$	Insert, ExtractMin, Delete, ModKey	✓	Computational
Ours	$O(\log^2 N)$	Insert, ExtractMin, Delete, ModKey	✓	**Perfect**

Moreover, the scheme of Keller and Scholl also supports *modifying a key*. However, these two OPQs are statistically secure, i.e., they impair the correctness of their structures or leak their access patterns stochastically (but in negligible probability). Jafargholi et al. [11] achieved an OPQ of the optimal $O(\log N)$ work per operation that supports all four operations described above while hiding the operations, depending on computationally secure hash functions. Shi [3] achieves an optimal statistically secure OPQ with the similar property as Jafargholi et al., but not relying on the existence of hash functions.

1.2 Our Contributions

As a starting point, we are motivated in the question of whether; *Is there a perfectly secure OPQ that supports the modifying-key operation and others, in practical efficiency (at least asymptotically the same as Toft's OPQ [10])?*

As mentioned above, there are significant tradeoffs between perfect and statistical (or computational) security. For ORAM, achieving perfect security increase the total work considerably. On the other hand, for OPQ, there are no perfectly secure schemes other than Toft's work, and unfortunately, his scheme supports minimal operations and is less efficient than the recent works. It is an important theoretical question to clarify the boundary between perfect and other security, and the work for a perfectly secure OPQ is one step in that study direction.

Moreover, regardless of the tradeoffs, we stress that perfect security is worth enough even in practice. The modifying-key operation of PQ is, for example, used for Dijkstra's algorithm, and hence part of the known OPQs [2], [3], [11] realizes a secure computation for graphs. However, these OPQs have negligible failure probability in N, so they cannot maximize their advantage of efficiency in small N, e.g., working with a small graph.

Through this paper, we show a novel *perfectly secure* OPQ that has the same functionalities as the recent schemes. I.e., our scheme supports *inserting, extracting, deleting*, and *modifying* an element[†] while hiding the operation in progress, and also it never impairs the correctness and also never leaks its access patterns. Similar to the known perfectly secure OPQ [10], our OPQ requires (amortized) $O(\log^2 N)$ total work per operation. We show a comparison for the known OPQs and ours in Table 1.

[†]The scheme of [3] supports *finding the top-priority, increasing* and *decreasing a key* as well as inserting and deleting. We note that *finding* operation can be realized by *extract-then-insert*, and *increasing* and *decreasing* can be realized the general modifying operation.

2. Preliminaries

2.1 Priority Queue

A (non-oblivious) priority queue (PQ) is a data structure that allows to set a *priority* to each data and to extract the data in ascending order of the priorities. We assume the following functionality that realizes a priority queue.

$\underline{\mathcal{F}_{\mathsf{PQ}}}$, holding a set $\mathbf{Q}$ as its state.

- $\mathsf{ref} \leftarrow \mathcal{F}_{\mathsf{PQ}}(\mathsf{Insert}, k, v)$: Receiving an Insert request with inputs key k and value v, $\mathcal{F}_{\mathsf{PQ}}$ returns a reference ref that uniquely corresponds to (k, v) and records it as $\mathbf{Q} \leftarrow \mathbf{Q} \cup \{(k, v, \mathsf{ref})\}$.
- $(k_{\min}, v_{\min}) \leftarrow \mathcal{F}_{\mathsf{PQ}}(\mathsf{ExtractMin})$: $\mathcal{F}_{\mathsf{PQ}}$ removes $(k_{\min}, v_{\min}, \mathsf{ref}_{\min}) \in \mathbf{Q}$ of the minimum key from $\mathbf{Q}$ and returns $(k_{\min}, v_{\min})$. If there exists two or more tuples of the minimum k, according to FIFO, $\mathcal{F}_{\mathsf{PQ}}$ outputs one inserted at first of them.
- $\mathcal{F}_{\mathsf{PQ}}(\mathsf{Delete}, \mathsf{ref})$: $\mathcal{F}_{\mathsf{PQ}}$ removes (k, v, ref) from $\mathbf{Q}$ and returns it. If there are no tuples corresponding to ref, (k, v) is set as $(\perp, \perp)$.
- $\mathcal{F}_{\mathsf{PQ}}(\mathsf{ModKey}, k', \mathsf{ref})$: $\mathcal{F}_{\mathsf{PQ}}$ removes (k, v, ref) from $\mathbf{Q}$ and sets $\mathbf{Q} \leftarrow \mathbf{Q} \cup \{(k', v, \mathsf{ref})\}$. If there are no tuples corresponding to ref, (k, v) is set as $(\perp, \perp)$.

Note that, similar to Shi [3], we define above to suit for the standard interface of a priority queue, i.e., we assume that, to call Delete and ModKey, the reference ref is required as input to handle the item being deleted or modified. We in addition assume that the reference ref is simply the *time*, e.g., the number of operations requested so far, at which the element was inserted.

2.2 Oblivious Simulations

The security of ODSs is defined in a simulation-based model, called *oblivious simulations*. We define an oblivious simulation for perfectly secure ODSs here.

Let $\mathcal{F}$ be an ideal functionality of a data structure, and DS be an algorithm that realizes the interface of $\mathcal{F}$. Let $\mathcal{A}$ be an arbitrary (unbounded) algorithm. Now, consider the following two experiments for a functionality $\mathcal{F}$ of any data structure:

- $\mathbf{Ideal}^{\mathcal{A}}_{\mathcal{F}}(N)$: $\mathcal{A}(N, \mathsf{View})$ adaptively generates a query

query for the data structure and obtain View = View $\cup$ $\{(\mathcal{F}(\mathsf{query}), \mathsf{Sim}(N))\}$, where Sim is a simulator.

- $\mathbf{Real}^{\mathcal{A}}_{\mathsf{DS}}(N)$: $\mathcal{A}(N, \mathsf{View})$ adaptively generates query and sends it to a challenger C that runs the real-world algorithm DS(query). C returns the output out and the access pattern[†] ap of DS(query) to $\mathcal{A}$, and $\mathcal{A}$ sets View = View $\cup$ $\{(\mathsf{out}, \mathsf{ap})\}$.

Definition 2.1 (Perfectly Secure Oblivious Simulation)**.** *We say that* DS *is a perfectly secure oblivious simulation of* $\mathcal{F}$ *iff for any integer* N *and any algorithm* $\mathcal{A}$*, there exists a simulator* Sim *s.t. the distribution of* View *in* $\mathbf{Ideal}^{\mathcal{A}}_{\mathcal{F}}(N)$ *is identical to that in* $\mathbf{Real}^{\mathcal{A}}_{\mathsf{DS}}(N)$*.*

Remark 1. *In the above security definition, we assume that the adversary* $\mathcal{A}$ *is only interested in the access patterns but not the contents of the data operated in* DS*. In practice, to hide the contents, we can employ standard techniques such as encryption.*

Applying Definition 2.1 to $\mathcal{F}_{\mathsf{PQ}}$ defined in Sect. 2.1, we can define the oblivious priority queue (OPQ)

Definition 2.2 (Perfectly Secure Oblivious Priority Queue)**.** *We say that* PQ *is a perfectly secure oblivious priority queue iff it is a perfectly secure oblivious simulator of* $\mathcal{F}_{\mathsf{PQ}}$*.*

Following [3], we assume the challenger in the real world behaves as follows: $\mathcal{A}$ in the ideal world $\mathbf{Ideal}^{\mathcal{A}}_{\mathcal{F}_{\mathsf{PQ}}}$ recognizes only the *time* at which elements were inserted as the references. On the other hand, the real-world algorithm PQ recognizes the references whose format is determined by PQ itself. Also, in contrast to $\mathcal{F}_{\mathsf{PQ}}$, the real-world input/output of PQ should have a certain format independent from a queried operation to hide the operation. Therefore, to bridge the gap between ideal and real, we assume that C in $\mathbf{Real}^{\mathcal{A}}_{\mathsf{PQ}}$ behaves as a man-in-the-middle between $\mathcal{A}$ and PQ s.t. it exchanges queries and answers between $\mathcal{A}$ and PQ while translating the input from the ideal-world format to the real-world format and vice versa.

2.3 Oblivious One-Time Memory

An Oblivious One-Time Memory (OTM), originally proposed by Chan et al. [8], [9] as a building block of their perfectly secure ORAM, is an ODS that completely hides access patterns up to one access for each block. We assume an ideal functionality of OTM, $\mathcal{F}_{\mathsf{OTM}}$, as below.

<u>$\mathcal{F}_{\mathsf{OTM}}$</u>:

- $\mathbf{U} \leftarrow \mathsf{Build}(\mathbf{I})$: Receiving a set $\mathbf{I}$ containing N elements $\mathsf{d}_i; i = 1, \ldots, N$, it returns a list $\mathbf{U}$ containing N positions pos_i uniquely corresponding to each d_i. In addition, $\mathcal{F}_{\mathsf{OTM}}$ stores all N pairs $(\mathsf{d}_i, \mathsf{pos}_i)$ and states $\mathbf{P}$ in its memory.
- $\mathsf{d} \leftarrow \mathsf{Lookup}(\mathsf{pos})$: The input pos is either real ($\in \mathbf{U}$) or dummy ($=\perp$). If pos is real, the output d is a real element s.t. $\mathcal{F}_{\mathsf{OTM}}$ holds a pair $(\mathsf{d}, \mathsf{pos})$, otherwise d is dummy. When $\mathsf{pos} \in \mathbf{P}$, $\mathcal{F}_{\mathsf{OTM}}$ halts without any output. Otherwise, it updates $\mathbf{P} \leftarrow \mathbf{P} \cup \{\mathsf{pos}\}$ then returns d.
- $\widetilde{\mathbf{I}} \leftarrow \mathsf{Getall}()$: $\mathcal{F}_{\mathsf{OTM}}$ returns a set $\widetilde{\mathbf{I}}$ that contains all real d_i s.t. $(\mathsf{d}_i, \mathsf{pos}_i)$ is in its memory and $\mathsf{pos}_i \notin \mathbf{P}$. This $\widetilde{\mathbf{I}}$ is padded with dummy elements and has a size N.

Slightly different from the definition described in [8], [9], in the above functionality definition, we omit a key k_i originally assigned to each element d_i (and also to each position pos_i) since it is enough for our OPQ where the key k_i cannot uniquely identify an exact element d_i. We in our scheme need to refer an element only by its position (treated as part of its reference), not by its key.

A perfectly secure oblivious simulation of $\mathcal{F}_{\mathsf{OTM}}$ can be obtained by slightly modifying the general OTM construction [8], [9]. More specifically, the algorithm is obtained by eliminating k_i from $\mathbf{U}$ and by specifying the parameters so that it holds up to N elements, accept up to N lookup requests, and no parallel query is allowed. Here we briefly describe its construction focusing on the features used in our scheme.

<u>OTM</u>:

- $\mathbf{U} \leftarrow \mathsf{Build}(\mathbf{I})$: Receiving the input set $\mathbf{I} = (\mathsf{d}_1, \ldots, \mathsf{d}_N)$, the algorithm selects an injection $f : \{1, \ldots, N\} \to \{1, \ldots, 2N\}$ uniform randomly and returns a list $\mathbf{U} = (\mathsf{pos}_1, \ldots, \mathsf{pos}_N)$ s.t. $\mathsf{pos}_i = f(i)$. In OTM, a table $\mathbf{T}$ of size $2N$ is maintained, in which all d_i is in $\mathbf{T}[\mathsf{pos}_i]$ and all other elements are dummy $\mathsf{d}_\perp$.
- $\mathsf{d} \leftarrow \mathsf{Lookup}(\mathsf{pos})$: The input pos is real ($\in \mathbf{U}$) or dummy ($=\perp$). If pos is real, the algorithm fetches $\mathsf{d} = \mathbf{T}[\mathsf{pos}]$, and update $\mathbf{T}$ as $\mathbf{T}[\mathsf{pos}] \leftarrow \mathsf{d}_\perp$. If $\mathsf{pos} = \perp$, it fetches $\mathbf{T}[\mathsf{pos}_\perp] = \mathsf{d}_\perp$ and returns $\mathsf{d} = \mathsf{d}_\perp$. For obliviousness, the position $\mathsf{pos}_\perp$ is chosen to be unique over the lifetime of OTM[††]. In our scheme, since we need to know which position pos' ($\in \{\mathsf{pos}, \mathsf{pos}_\perp\}$) has been actually accessed, we include this pos' in the output of OTM.Lookup hereafter.
- $\widetilde{\mathbf{I}} \leftarrow \mathsf{Getall}()$: The output is a set $\widetilde{\mathbf{I}}$ that contains all real elements remaining in $\mathbf{T}$ and dummies, to be a size N.

Fact 2.3. *The above* OTM = (Build, Lookup, Getall) *satisfies that: there exists a stateful simulator* $\mathsf{Sim}_{\mathsf{OTM}}$ *s.t. for any query sequence* $(\mathbf{I}, q_1, \ldots, q_m)$*, the simulation sequence* $\left(\mathsf{Sim}_{\mathsf{OTM}}(\mathsf{Build}, N), \mathsf{Sim}_{\mathsf{OTM}}(\mathsf{Lookup}, N), \ldots, \mathsf{Sim}_{\mathsf{OTM}}(\mathsf{Lookup}, N), \mathsf{Sim}_{\mathsf{OTM}}(\mathsf{Getall}, N)\right)$ *is identical to the real access pattern* $\left(\mathsf{OTM.Build}(\mathbf{I}), \mathsf{OTM.Lookup}(\mathsf{pos}_{q_1}), \ldots,\right.$

[†] By the "access pattern" of an algorithm, we mean "the sequence consisting of all memory addresses accessed" via the algorithm.

[††] In practice, as same as [8], [9], this *one-time dummy* lookup is realized by a linked list of dummy positions provided in Build. To simplify our algorithm descriptions, we omit these details here.

$\mathsf{OTM.Lookup}(\mathsf{pos}_{q_m}), \mathsf{OTM.Getall}())$ *if* $N = |\mathbf{I}|$ *and*

- $m \leq N$,
- $q_i \in \{1, \ldots, N, \perp\}$ *(where we assume* $\mathsf{pos}_\perp = \perp$*) for all* $i \leq m$, *and*
- $q_i \neq q_j$ *for all* $i, j \leq m$ *and* $q_i, q_j \neq \perp$.

Moreover, for any input set $\mathbf{I}$ *of size* N, *the total work of* OTM.Build *and* OTM.Getall *are* $O(N \log N)$, *and that of* OTM.Lookup *is* $O(1)$.

3. Perfectly Secure OPQ

In this section, we propose a novel perfectly-secure OPQ that supports four operations, insertion, extraction of the top-priority element, deletion, and key-modification, and operation hiding.

3.1 Our Data Structure: A Hierarchical Tree

Before describing the details of our OPQ, we introduce the components of our data structure below.

- For each value v stored to the OPQ, we call a tuple (k, τ, v) a *block*, where k is a key that indicates the priority of v, and τ denotes the time in which v is inserted. This τ is used to uniquely identify the first inserted one from the tuples of the same key.
- A dummy block is denoted as $(\perp, \perp, \perp)$.
- Let N be the maximum number of blocks stored to the OPQ. Then, our data structure consists of a L-level hierarchy of OTMs, where $L = \lceil \log N \rceil + 1$. For each level $i = 1, \ldots, L$, we assume the instance OTM_i of capacity 2^{i-1}, i.e., there is a table $\mathbf{T}_i$ of size 2^i in each OTM_i. In our scheme, each $\mathbf{T}_i$ is updated only via the three operations of OTM_i, but a block in a cell $\mathbf{T}_i[j]$ is directly referred by any algorithms specifying i and j. In addition, we assign a binary state filled or empty to each OTM_i. We say that OTM_i is filled if OTM_i.Getall has not been performed after the last OTM_i.Build, otherwise say that OTM_i is empty.
- For each (non-dummy) block (k, τ, v), let $\mathsf{ref} = (k, \tau, \mathsf{pos}, \mathsf{lv})$ be the *reference* of v, which uniquely identifies that the value v of key k, inserted in time τ, is in $\mathbf{T}_{\mathsf{lv}}[\mathsf{pos}]$. Also, let $\mathsf{ref}_\perp = (\perp, \perp, \perp, \perp)$ be a dummy reference.
- Our data structure also contains a complete binary tree of height L, called MinTree, that manages the reference to the top-priority block. In this tree, each node $\mathsf{node}_{i,j}; j = 1, \ldots, 2^i$ of height $i = 1, \ldots, L$ (except the root) corresponds to a cell $\mathbf{T}_i[j]$, and holds a reference ref that satisfies Claim 3.1 described below.

We call the data structure described above a *Hierarchical Tree*, and also show its abstract in Fig. 1.

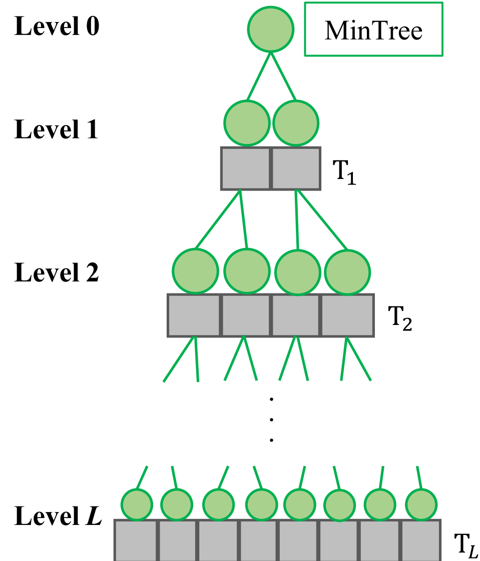

Fig. 1 A Hierarchical Tree: Each block (k, τ, v) in the Hierarchical Tree is placed to one of the cells of $\mathbf{T}_i$, the gray squares. The green circles denote the nodes of MinTree, each of which holds a reference ref.

Claim 3.1 (A state of MinTree)**.** *Each node in* MinTree *satisfies the following conditions.*

- *For level* $i = 0, \ldots, L$, *each node* $\mathsf{node}_{i,j} (j = 1, \ldots, 2^i)$ *holds the top-priority reference* $\mathsf{ref}_{i,j} = (k^{\min}_{i,j}, \tau^{\min}_{i,j}, \mathsf{pos}^{\min}_{i,j}, \mathsf{lv}^{\min}_{i,j})$ *s.t.* $k^{\min}_{i,j}$ *is the minimum key of all cells each corresponding to a node of* MinTree*'s subtree where the root is* $\mathsf{node}_{i,j}$, *and* $\tau^{\min}_{i,j}$ *indicates the first time a block with* $k^{\min}_{i,j}$ *is inserted.* $(\mathsf{pos}^{\min}_{i,j}, \mathsf{lv}^{\min}_{i,j})$ *denotes the position where the block that has* $(k^{\min}_{i,j}, \tau^{\min}_{i,j})$ *is.*
- *In particular, for level* L, *each leaf* $\mathsf{node}_{L,j} (j = 1, \ldots, 2^L)$ *holds the reference* $\mathsf{ref}_{L,j} = (k_{L,j}, \tau_{L,j}, j, L)$ *of its corresponding cell* $\mathbf{T}_L[j]$.

If there exist two or more references of the minimum key, let the top-priority one is that has the minimum τ. Also, if there are no real keys, let $\mathsf{ref}_\perp$ be the top-priority reference.

In Lemma 3.7 of Sect. 3.3, we show that Claim 3.1 always holds in our scheme.

3.2 Priority-Queue Operations for the Hierarchical Tree

Before we describe the details of our entire OPQ, we introduce the separate implementation of each PQ operation. The implementations of Insert and Delete are shown in Sects. 3.2.2 and 3.2.3, respectively. Each of ExtractMin and ModKey is in practice realized by using Insert and Delete (see Sects. 3.2.4 and 3.2.5). A set of these implementations is almost[†] an oblivious simulation of $\mathcal{F}_{\mathsf{PQ}}$, but clearly leaks its running operation.

[†] As we mention in Remark 3 of Sect. 3.2.3, consecutive multiple runs of Delete violate obliviousness.

3.2.1 Subroutine: SelectMin

We first introduce a subroutine SelectMin, described in Algorithm 1, to manage references in MinTree. This SelectMin receives node, a node of MinTree, and it updates the input node with the top-priority reference among one that refers the cell corresponding to node and ones held by its children. It requires $O(1)$ work and has access patterns determined by node.

Algorithm 1 SelectMin(node)

1: Obtain the reference ref of the cell corresponding to node, i.e., if node is the pos-th node in level lv, load the corresponding cell $\mathbf{T}_{\mathsf{lv}}[\mathsf{pos}] = (k, \tau, v)$ and set ref $= (k, \tau, \mathsf{pos}, \mathsf{lv})$.
   If $\mathbf{T}_{\mathsf{lv}}[\mathsf{pos}] = (\perp, \perp, \perp)$ or $\mathsf{OTM}_{\mathsf{lv}}$ is empty, then set ref = $\mathsf{ref}_{\perp}$.
   Also, if node is the root of MinTree, then set ref = $\mathsf{ref}_{\perp}$.
2: Load the references $\mathsf{ref}_L = (k_L, \tau_L, \mathsf{pos}_L, \mathsf{lv}_L)$ and $\mathsf{ref}_R = (k_R, \tau_R, \mathsf{pos}_R, \mathsf{lv}_R)$ from the two children of node.
   If node is a leaf of MinTree, set $\mathsf{ref}_L = \mathsf{ref}_R = \mathsf{ref}_{\perp}$.
3: Select the top-priority one $\mathsf{ref}_{\min}$ from ref, ref_L, and ref_R.
4: Store $\mathsf{ref}_{\min}$ to node.

Lemma 3.2. *Assume that* $\mathbf{T}_{\mathsf{lv}}[\mathsf{pos}]$ *is modified in operating the Hierarchical Tree, and* MinTree *satisfies Claim 3.1 before this modification. Let* $\mathsf{node}_{\mathsf{lv}}$ *be a node corresponding to the cell* $\mathbf{T}_{\mathsf{lv}}[\mathsf{pos}]$*, and let* $\mathsf{node}_{\mathsf{lv}-1}, \ldots, \mathsf{node}_0$ *be nodes on the path from* $\mathsf{node}_{\mathsf{lv}}$ *to the root (=* node_0*). Then, the claim holds even after the modification if* SelectMin(node_i) *is executed in the order from* $i = \mathsf{lv}$ *to* 0.

The proof of Lemma 3.2 is given in Appendix A.1

3.2.2 The Insert Algorithm

Algorithm 2 $\overrightarrow{\mathsf{ref}} \leftarrow$ Insert(k, τ, v)

(Inserting a block (k, τ, v) **to the OTM hierarchy.)**
1: Let ℓ be the lowest level where OTM_ℓ is empty. If all L OTMs are filled, then set ℓ as $L + 1$.
2: Initialize a set $\mathbf{I}_\ell = \{(k, \tau, v)\}$.
3: For $i = 1, \ldots, \ell-1$, perform $\widetilde{\mathbf{I}_i} \leftarrow \mathsf{OTM}_i$.Getall(), then set $\mathbf{I}_\ell \leftarrow \mathbf{I}_\ell \cup \widetilde{\mathbf{I}_i}$ and OTM_i as empty.
4: If $\ell \leq L$, run $\mathbf{U}_\ell \leftarrow \mathsf{OTM}_\ell$.Build($\mathbf{I}_\ell$) and set OTM_ℓ as filled.
   Otherwise, compute a set $\mathbf{I}_L$ of size N, which contains all real blocks of $\mathbf{I}_\ell$†, then run $\mathbf{U}_L \leftarrow \mathsf{OTM}_L$.Build($\mathbf{I}_L$) and set OTM_L as filled.
5: For all $(k_j, \tau_j, v_j) \in \mathbf{I}_\ell$ and $\mathsf{pos}_j \in \mathbf{U}_\ell$, set $\mathsf{ref}_j = (k_j, \tau_j, \mathsf{pos}_j, \ell)$.
   If $\ell = L + 1$, consider the above $\mathbf{I}_\ell$ and $\mathbf{U}_\ell$ as $\mathbf{I}_L$ and $\mathbf{I}_L$, respectively.
(Updating MinTree**.)**
6: For each level i from ℓ to 0, for all $\mathsf{node}_{i,j}; j = 1, \ldots, 2^\ell$, perform SelectMin($\mathsf{node}_{i,j}$).
7: **Return** $\overrightarrow{\mathsf{ref}} = (\mathsf{ref}_1, \ldots, \mathsf{ref}_{2^{\ell-1}})$.

In Algorithm 2, we describe the details of our implementation of Insert that consists of two phases: Inserting the input to the OTM hierarchy and Updating MinTree.

In the first phase, the input block is inserted to the lowest-level OTM that is empty, and also all blocks in upper filled OTMs are moved to that level. Since the capacity of OTM_i is 2^{i-1}, in line 3 of Algorithm 2, the size of $\mathbf{I}_\ell$ finally becomes to $1 + \sum_{i=1}^{\ell-1} 2^{i-1} = 2^{\ell-1}$, and hence it satisfies the capacity of OTM_ℓ when $\ell \leq L$. Moreover, even when $\ell = L + 1$, since the total capacity of the Hierarchical Tree is N, there are at most N real elements in $\mathbf{I}_{L+1}$. Hence, all real elements in the Hierarchical Tree are correctly stored to OTM_L in line 4.

Finally, since $\mathbf{T}_\ell$ is rebuilt and lower $\mathsf{OTM}_i; i > \ell$ are not changed, Insert updates all nodes in level ℓ or higher.

Lemma 3.3 (Efficiency of Insert)**.** Insert *described in Algorithm 2 requires (amortized)* $O(\log^2 N)$ *work.*

Lemma 3.4 (Correctness of Insert)**.** *After an execution of* Insert *described in Algorithm 2, the input block* (k, τ, v) *is correctly stored to the Hierarchical Tree, and any block previously held in the Hierarchical tree remains in it, with probability* 1*. Moreover, if Claim 3.1 has been satisfied before an execution of* Insert*, it is satisfied thereafter.*

The proofs of Lemma 3.3 and 3.4 are given in Appendices A.2 and A.3, respectively.

Remark 2. *Though* Insert *described in Algorithm 2 can work almost as an implementation of* $\mathcal{F}_{\mathsf{PQ}}$(Insert$, k, v$)*, for our goal to hide the running operations, we consider it only as a building block of our entire scheme (see Sect. 3.3). Hence, here we define* Insert *to take* τ *as input, unlike* $\mathcal{F}_{\mathsf{PQ}}$.

3.2.3 The Delete Algorithm

Algorithm 3 $(k, \tau, v) \leftarrow$ Delete(ref)

(Removing (k, τ, v) **from the OTM hierarchy.)**
1: **for all** level $i = 1, \ldots, L$ where OTM_i is filled **do**
2:     For the input ref $= (k, \tau, \mathsf{pos}, \mathsf{lv})$, set $\mathsf{pos}_i = \mathsf{pos}$ if $i = \mathsf{lv}$, otherwise set $\mathsf{pos}_i = \perp$.
3:     Perform $(k_i, \tau_i, v_i), \mathsf{pos}'_i \leftarrow \mathsf{OTM}_i$.Lookup($\mathsf{pos}_i$).
4: Set $(k, \tau, v) = (k_i, \tau_i, v_i)$ for some i s.t. (k_i, τ, v_i) is a non-dummy block. If all (k_i, τ_i, v_i) is dummy, then set $(k, \tau, v) = (\perp, \perp, \perp)$.
(Updating MinTree**.)**
5: **for** level $i = L$ to 0 **do**
6:     Run SelectMin($\mathsf{node}_{i,\mathsf{pos}'_i}$).
7:     Run SelectMin($\mathsf{node}_{i,j}$) for all $\mathsf{node}_{i,j}$ s.t. at least one of the nodes $\{\mathsf{node}_{i+1,\mathsf{pos}'_{i+1}}, \ldots, \mathsf{node}_{L,\mathsf{pos}'_L}\}$ is a descendant of $\mathsf{node}_{i,j}$.
   ▷ The number of $\mathsf{node}_{i,j}$ is at most $L - i$.
8: **Return** (k, τ, v).

We show the details of our implementation of Delete in Algorithm 3. Similar to Insert, this Delete also has two phases to remove the queried block and then update MinTree. In this algorithm, since L cells in the hierarchy are accessed to remove one block to hide which cell is actually queried, we should update all nodes corresponding the L cells as well

†This process is realized by using an *oblivious sorting* relying on such as AKS sorting network [12], i.e., obliviously sort all blocks of $\mathbf{I}_\ell$ by ascending order of keys and then set $\mathbf{I}_L$ as the first N elements of $\mathbf{I}_\ell$ (that contains $2N - 1$ blocks).

as their all ancestors.

Lemma 3.5 (Efficiency of Delete)**.** Delete *described in Algorithm 3 requires* $O(log^2 N)$ *work.*

Lemma 3.6 (Correctness of Delete)**.** *After an execution of* Delete *described in Algorithm 3, the exact element referred by the input is extracted from the Hierarchical Tree, and all other elements remain in the Hierarchical Tree, with probability* 1. *Moreover, if Claim 3.1 has been satisfied before, it still holds after an execution of* Delete.

Proof of Lemma 3.5 and 3.6 are given in Appendices A.4 and A.5, respectively.

Remark 3. *Note that this* Delete *does not work as* $\mathcal{F}_{\mathsf{PQ}}(\mathsf{Delete}, *)$ *by itself. Specifically, for example, operation sequences such as executing* Delete *twice in a row violate obliviousness of* OTM_1 *that allows only one* Lookup *request in its lifetime. We solve this problem in the construction of our entire OPQ.*

3.2.4 The ExtractMin Algorithm

Algorithm 4 $(k_{\min}, \tau_{\min}, v_{\min}) \leftarrow \mathsf{ExtractMin}()$

1: Load the reference $\mathsf{ref}_{\min}$ in the root of MinTree.
2: Perform $(k_{\min}, \tau_{\min}, v_{\min}) \leftarrow \mathsf{Delete}(\mathsf{ref}_{\min})$.
3: **Return** $(k_{\min}, \tau_{\min}, v_{\min})$.

Algorithm 4 denotes an implementation of ExtractMin to obtain the top-priority block. Since the root of MinTree always has the reference to the top-priority block when Claim 3.1 holds, this algorithm correctly extracts the block relying on correctness of Delete.

3.2.5 The ModKey Algorithm

Algorithm 5 $\mathsf{ModKey}(k', \mathsf{ref})$

1: Perform $(k, \tau, v) \leftarrow \mathsf{Delete}(\mathsf{ref})$.
2: Perform $\overrightarrow{\mathsf{ref}} \leftarrow \mathsf{Insert}(k', \tau, v)$.

As shown in Algorithm 5, we can straightforwardly achieve ModKey as that; Delete a queried block, modify the key of the block, and then re-insert it. If Delete correctly removes the target block from the Hierarchical Tree, and if Insert correctly stores the input block to that, it is clear that Algorithm 5 is an implementation of $\mathcal{F}_{\mathsf{PQ}}$.ModKey.

3.3 Our Perfectly Secure OPQ, PQ

Now, we can construct our perfectly secure OPQ that supports all four operations described above, while hiding *which operation is in progress*. I.e., the OPQ simulates all Insert, Delete, ExtractMin, and ModKey with access patterns independent from the running operation.

Algorithm 6 $\overrightarrow{\mathsf{ref}}, (\tilde{k}, \tilde{v}) \leftarrow \mathsf{PQ}(\mathsf{op}, k, v, \mathsf{ref})$

Require: $\mathsf{op} \in \{\mathsf{Insert}, \mathsf{Delete}, \mathsf{ExtractMin}, \mathsf{ModKey}\}$.
(Extracting a block requested by op.**)**
1: Let ctr be a counter that indicates the number of PQ performed so far.
2: Load the reference $\mathsf{ref}_{\min}$ in the root of MinTree.
3: Set $\mathsf{ref}' = \mathsf{ref}$ if $\mathsf{op} \in \{\mathsf{Delete}, \mathsf{ModKey}\}$,
or set $\mathsf{ref}' = \mathsf{ref}_{\min}$ if $\mathsf{op} = \mathsf{ExtractMin}$.
Otherwise, set $\mathsf{ref}' = \mathsf{ref}_{\perp}$.
4: Perform $(\tilde{k}, \tilde{\tau}, \tilde{v}) \leftarrow \mathsf{Delete}(\mathsf{ref}')$.
(Inserting a block requested by op.**)**
5: Set $(k', \tau', v') = (k, \mathsf{ctr}, v)$ if $\mathsf{op} = \mathsf{Insert}$,
or set $(k', \tau', v') = (k, \tilde{\tau}, \tilde{v})$ if $\mathsf{op} = \mathsf{ModKey}$.
Otherwise, set $(k', \tau', v') = (\perp, \perp, \perp)$.
6: Perform $\overrightarrow{\mathsf{ref}} \leftarrow \mathsf{Insert}(k', \tau', v')$.
7: Set $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$.
8: **Return** $\overrightarrow{\mathsf{ref}}, (\tilde{k}, \tilde{v})$.

Remarking that our ExtractMin and ModKey are realized on an execution of Delete (or Delete-then-Insert sequence), and also that our Insert can refresh OTMs reached their limit of lookup requests via Delete†, we can obtain a *deterministic sequence* that realizes OPQ. We show the details of our scheme in Algorithm 6.

Lemma 3.7. *Assume the initial state of the Hierarchical Tree where all* $\mathsf{OTM}_i; i = 1, \ldots, L$ *are* empty *and all nodes in* MinTree *holds* $\mathsf{ref}_{\perp}$. *Then, Claim 3.1 holds after any* PQ *execution.*

Proof. In the initial state, it is clear that MinTree satisfies Claim 3.1. Therefore, from Lemma 3.4 and 3.6, the claim holds after any execution of PQ. □

Theorem 3.8. PQ*, described in Algorithm 6, is a perfectly secure oblivious priority queue that consumes amortized* $O(\log^2 N)$ *total work per operation.*

Proof. Proof of obliviousness is given in Appendix 3.8, and we show efficiency and correctness of our PQ here. The efficiency is derived by Lemma 3.3 and 3.5 as that: Delete requires $O(\log^2 N)$ work and Insert requires amortized $O(\log^2 N)$ work.

To prove correctness of PQ, we show PQ behaviors at each input op.

- If op = Insert, PQ runs $\mathsf{Delete}(\mathsf{ref}_{\perp})$ then $\mathsf{Insert}(k, \tau, v)$ where τ is a query counter.
- If op = Delete, PQ runs Delete(ref) then $\mathsf{Insert}(\perp, \perp, \perp)$.
- If op = ExtractMin, PQ runs the same algorithm as ExtractMin() then $\mathsf{Insert}(\perp, \perp, \perp)$.
- If op = ModKey, PQ runs the same algorithm as $\mathsf{ModKey}(k, \mathsf{ref})$.

Now, by Lemma 3.4 and 3.6, the interface of PQ with each $\mathsf{op} \in \{\mathsf{Insert}, \mathsf{Delete}, \mathsf{ModKey}\}$ is equivalent to that of $\mathcal{F}_{\mathsf{PQ}}$.Insert, $\mathcal{F}_{\mathsf{PQ}}$.Delete, and $\mathcal{F}_{\mathsf{PQ}}$.ModKey, respectively. Moreover, with op = ExtractMin, the interface of PQ is equivalent to that of $\mathcal{F}_{\mathsf{PQ}}$.ExtractMin by Lemma 3.6 and 3.7. □

†The similar *access-then-re-randomize* sequence is commonly used in hierarchical ORAMs [5], [7]–[9].

4. Conclusion

We proposed a novel perfectly secure oblivious priority queue that supports four operations, Insert, Delete, ExtractMin, and ModKey, while hiding not only its access patterns but also queried operations. This property is the same as the known best OPQ of Shi [3] except our stronger security.

Moreover, for any positive integer N, our OPQ of capacity N can be accessed in amortized $O(\log^2 N)$ total work, which is the same cost as the known perfectly secure OPQ of Toft [10].

References

[1] X.S. Wang, K. Nayak, C. Liu, T.H.H. Chan, E. Shi, E. Stefanov, and Y. Huang, "Oblivious data structures," Proc. 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14, New York, NY, USA, pp.215–226, Association for Computing Machinery, 2014.

[2] M. Keller and P. Scholl, "Efficient, oblivious data structures for mpc," Advances in Cryptology – ASIACRYPT 2014, Berlin, Heidelberg, pp.506–525, Springer Berlin Heidelberg, 2014.

[3] E. Shi, "Path oblivious heap: Optimal and practical oblivious priority queue," 2020 IEEE Symposium on Security and Privacy (SP), Los Alamitos, CA, USA, pp.842–858, IEEE Computer Society, May 2020.

[4] O. Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs," Proc. Nineteenth Annual ACM Symposium on Theory of Computing, STOC'87, New York, NY, USA, pp.182–194, Association for Computing Machinery, 1987.

[5] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," J. ACM, vol.43, no.3, pp.431–473, May 1996.

[6] R. Ostrovsky, "Efficient computation on oblivious RAMs," Proc. Twenty-Second Annual ACM Symposium on Theory of Computing, STOC'90, New York, NY, USA, p.514–523, Association for Computing Machinery, 1990.

[7] G. Asharov, I. Komargodski, W. Lin, K. Nayak, E. Peserico, and E. Shi, "Optorama: Optimal oblivious RAM," Advances in Cryptology - EUROCRYPT, pp.403–432, 2020.

[8] T.H.H. Chan, K. Nayak, and E. Shi, "Perfectly secure oblivious parallel RAM," Theory of Cryptography, Cham, pp.636–668, Springer International Publishing, 2018.

[9] T.H.H. Chan, E. Shi, W.K. Lin, and K. Nayak, "Perfectly oblivious (parallel) RAM revisited, and improved constructions," Cryptology ePrint Archive, Report 2020/604, 2020. https://ia.cr/2020/604

[10] T. Toft, "Secure data structures based on multi-party computation," Proc. 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC'11, New York, NY, USA, pp.291–292, Association for Computing Machinery, 2011.

[11] Z. Jafargholi, K.G. Larsen, and M. Simkin, "Optimal oblivious priority queues," Proc. 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA), pp.2366–2383, Society for Industrial and Applied Mathematics, USA, 2021.

[12] M. Ajtai, J. Komlós, and E. Szemerédi, "An 0(n log n) sorting network," Proc. Fifteenth Annual ACM Symposium on Theory of Computing, STOC'83, New York, NY, USA, pp.1–9, Association for Computing Machinery, 1983.

Appendix A: Deferred Proofs for Lemmas

A.1 Proof of Lemma 3.2

If only $\mathbf{T}_{\mathsf{lv}}[\mathsf{pos}]$ is modified when all nodes in MinTree satisfies the condition of Claim 3.1, it is clear that all nodes except ancestors of $\mathsf{node}_{\mathsf{lv},\mathsf{pos}}$ corresponding to $\mathbf{T}_{\mathsf{lv}}[\mathsf{pos}]$ still satisfy the condition after the modification. Furthermore, for any node, if both of its children holds the condition, i.e., each child has the top-priority reference in all of its descendants, then it is clear that SelectMin(node) makes node to meet the condition. Therefore, running $\mathsf{SelectMin}(\mathsf{node}_i)$ in order from $i = \mathsf{lv}$ to 0 while tracing the path from $\mathsf{node}_{\mathsf{lv}}$ to the root node_0 ensures that all nodes in MinTree meet the condition.

A.2 Proof of Lemma 3.3

When $\ell \leq L$, since Algorithm 2 requires executions of OTM_i.GetAll for $i = 1, \ldots, \ell - 1$ and OTM_ℓ.Build, it takes $\sum_{i=1}^{\ell} O(2^i \log 2^i) = O(2^\ell \ell)$ work. It also needs SelectMin for $2^{\ell+1} - 1$ nodes, thus it additionally costs $O(2^\ell)$ work. Hence, the Insert on level ℓ has $O(2^\ell \ell)$ total work.

We can observe that the above work occurs for every 2^ℓ insertions, for each $\ell = 1, \ldots, L$. Consider that $\mathsf{OTM}_1, \ldots, \mathsf{OTM}_{\ell-1}$ are empty and OTM_ℓ is filled, i.e., OTM_ℓ is built in the last insertion. Since $\mathsf{OTM}_1, \ldots, \mathsf{OTM}_{\ell-1}$ can hold totally $2^{\ell-1} - 1$ blocks, OTM_ℓ is deconstructed in $2^{\ell-1}$-th execution of Insert, in which $\mathsf{OTM}_{\ell+1}$ (or lower) is built. OTM_ℓ is constructed again after $2^{\ell-1}$ additional insertions, thus the life-cycle of OTM_ℓ is 2^ℓ insertions.

On the other hand, when $\ell = L + 1$, Algorithm 2 additionally requires an oblivious sorting on $\mathbf{I}_\ell$ of size 2^{L+1}. If we employ the oblivious sorting based on AKS sorting network [12], since it requires $O(N \log N)$ to sort N elements, Insert costs totally $O(2^L L)$ work in this case. Once OTM_L is built, the above work occurs for every 2^{L-1} insertions since the total capacity of $\mathsf{OTM}_1, \ldots, \mathsf{OTM}_{L-1}$ is $2^{L-1} - 1$.

Consequently, the amortized cost of Insert is $\sum_{\ell=1}^{L} O(2^\ell \ell)/2^\ell + O(2^L L)/2^{L-1} = O(\log^2 N)$.

A.3 Proof of Lemma 3.4

Since OTM_i.GetAll realizes $\mathcal{F}_{\mathsf{OTM}}$.GetAll with probability 1, all real blocks of $\mathbf{T}_1, \ldots, \mathbf{T}_{\ell-1}$ are combined to $\mathbf{I}_\ell$ in line 3 of Algorithm 2. In addition, since the input block is also in $\mathbf{I}_\ell$, the Hierarchical Tree holds the input and all (non-dummy) blocks previously held in level 1 to $\ell - 1$ are correctly stored to level ℓ, relying on correctness of OTM.Build.

In Algorithm 2, all cells of level ℓ and higher are modified. Hence, by Lemma 3.2, Claim 3.1 is satisfied after Insert since it updates all nodes from level ℓ to 0 iteratively in line 6.

A.4 Proof of Lemma 3.5

Algorithm 3 requires L executions of OTM_i.Lookup and at

most $\sum_{i=0}^{L}(L-i)$ executions of SelectMin. Thus, it costs $O(L^2) = O(\log^2 N)$ work.

A.5 Proof of Lemma 3.6

In Algorithm 3, since $\mathsf{OTM}_{\mathsf{lv}}$.Lookup realizes $\mathcal{F}_{\mathsf{OTM}}$.Lookup with probability 1, we can find and remove the target block (k,τ,v) of $\mathbf{T}_{\mathsf{lv}}[\mathsf{pos}]$, referred by the input ref $= (k,\tau,\mathsf{pos},\mathsf{lv})$, correctly. On the other hand, since the algorithm runs OTM_i.Lookup(⊥) for all $i \neq \mathsf{lv}$, all other (non-dummy) blocks in the Hierarchical Tree remain in their place.

Moreover, in line 5 to 7 of Algorithm 3, at least all nodes on the path from $\mathsf{node}_{\mathsf{lv},\mathsf{pos}}$ (corresponding to the deleted $\mathbf{T}_{\mathsf{lv}}[\mathsf{pos}]$) to the root of MinTree are updated, thus Claim 3.1 maintains after Delete by Lemma 3.2[†].

A.6 Security Proof of Theorem 3.8

To prove the access-pattern obliviousness of PQ, consider an ideal-world simulator Sim described below.

Sim: At the beginning of the experiment $\mathbf{Ideal}^{\mathcal{A}}_{\mathcal{F}_{\mathsf{PQ}}}(N)$, Sim generates a Hierarchical Tree of capacity N in its memory, where all OTM_i are empty and all nodes of MinTree hold $\mathsf{ref}_\perp$. Then, for each request of $\mathcal{A}$, Sim runs PQ(Insert, ⊥, ⊥, $\mathsf{ref}_\perp$) regardless of $\mathcal{A}$'s query and returns the access pattern of that simulation.

From the definition of the challenger C in the real world, C's behavior is described as follows.

C: At the beginning of the experiment $\mathbf{Real}^{\mathcal{A}}_{\mathsf{PQ}}(N)$, C allocates a Hierarchical Tree of capacity N in its memory to perform PQ. The Hierarchical Tree is initialized as all OTM_i are empty and all nodes of MinTree hold $\mathsf{ref}_\perp$. In addition, C maintains a *dictionary* (initialized as empty). This dictionary is used to store real-world references ref $= (k,\tau,\mathsf{pos},\mathsf{lv})$, and also to search for ref by τ. Note that this τ, representing the time at which a block (k,τ,v) was inserted, is regarded as an ideal-world reference. After this initialization, receiving query consisting of parts op, k, v, and τ from $\mathcal{A}$, C works as a man-in-the-middle between $\mathcal{A}$ and PQ as below:

- If query = (Insert, k, v), C runs $\overrightarrow{\mathsf{ref}}, (\widetilde{k}, \widetilde{v}) \leftarrow$ PQ(Insert, k, v, $\mathsf{ref}_\perp$) and outputs the access pattern of that execution. Then, C updates the dictionary by all references in $\overrightarrow{\mathsf{ref}}$.
- If query = (ExtractMin), C runs $\overrightarrow{\mathsf{ref}}, (\widetilde{k}, \widetilde{v}) \leftarrow$ PQ(ExtractMin, ⊥, ⊥, $\mathsf{ref}_\perp$) and outputs the access pattern of that execution. Then, C updates the dictionary by all references in $\overrightarrow{\mathsf{ref}}$.
- If query = (Delete, τ), C searches the dictionary by τ and obtain ref $= (k,\tau,\mathsf{pos},\mathsf{lv})$ at first. When there is no reference corresponding to τ, C sets ref = $\mathsf{ref}_\perp$. Then, C runs $\overrightarrow{\mathsf{ref}}, (\widetilde{k}, \widetilde{v}) \leftarrow$ PQ(Delete, ⊥, ⊥, ref) and outputs the access pattern of that execution. Finally, C updates the dictionary by all references in $\overrightarrow{\mathsf{ref}}$.
- If query = (ModKey, k', ref), as same as the above Delete, C searches ref corresponding to τ at first. ref = $\mathsf{ref}_\perp$ if there is no reference corresponding to τ. Then, C runs $\overrightarrow{\mathsf{ref}}, (\widetilde{k}, \widetilde{v}) \leftarrow$ PQ(ModKey, k', ⊥, ref) and outputs the access pattern of that execution. Finally, C updates the dictionary by all references in $\overrightarrow{\mathsf{ref}}$.

To show the output of those Sim and C is identical, assume the following hybrid arguments.

Hyb_0: $\mathcal{A}$ receives the output of C.

Hyb_1: The same as Hyb_0 except that C runs PQ while replacing all OTM_i with $\mathcal{F}_{\mathsf{OTM}}$ and $\mathsf{Sim}_{\mathsf{OTM}}$ as follows:

- In Insert;
  1. In line 3 of Algorithm 2, OTM_i.Getall is replaced with $\mathcal{F}_{\mathsf{OTM}}$.Getall, and let the access pattern of this step be $\mathsf{Sim}_{\mathsf{OTM}}(\mathsf{Getall}, 2^{i-1})$.
  2. In line 4, OTM_ℓ.Build is replaced with $\mathcal{F}_{\mathsf{OTM}}$.Build, and let the access pattern of this step be $\mathsf{Sim}_{\mathsf{OTM}}(\mathsf{Build}, 2^{\ell-1})$.
- In Delete;
  1. In line 3 of Algorithm 3, OTM_i.Lookup is replaced with $\mathcal{F}_{\mathsf{OTM}}$.Lookup, and let the access pattern of this step be $\mathsf{Sim}_{\mathsf{OTM}}(\mathsf{Lookup}, 2^{i-1})$.
  2. In line 5 to 7, let $\mathsf{node}_{i,\mathsf{pos}'_i}$ be determined by the above simulation $\mathsf{Sim}_{\mathsf{OTM}}(\mathsf{Lookup}, 2^{i-1})$.

Hyb_2: $\mathcal{A}$ receives the output of Sim(N).

Claim Appendix A.1. Hyb_0 *is identical to* Hyb_1.

Proof. Since PQ performs Insert after Delete in both Hyb_0 and Hyb_1, we can observe that all OTM_ℓ is deconstructed by Getall after $2^{\ell-1}$ executions of Lookup. Moreover, in Hyb_0, since any (real) block once removed from OTM_ℓ is then assigned to another position and level (by ModKey) or completely erased from the Hierarchical Tree and the dictionary of C (by Delete), the same pos is never queried to OTM_ℓ.Lookup twice. Consequently, by Fact 2.3, all $\mathsf{Sim}_{\mathsf{OTM}}(\mathsf{Build}, 2^{\ell-1})$, $\mathsf{Sim}_{\mathsf{OTM}}(\mathsf{Getall}, 2^{\ell-1})$, and $\mathsf{Sim}_{\mathsf{OTM}}(\mathsf{Lookup}, 2^{\ell-1})$ in Hyb_1 are identical to the access patterns of OTM_ℓ.Build, OTM_ℓ.Getall, and OTM_ℓ.Lookup in Hyb_0. □

Claim Appendix A.2. Hyb_1 *is identical to* Hyb_2.

Proof. For any input (op, k, v, ref), in Hyb_1, the access patterns provided by $\mathsf{Sim}_{\mathsf{OTM}}$ are independent from the input. Moreover, the part of Insert to update MinTree (line 6 of Algorithm 2) has the access pattern determined only by ℓ that corresponds to the number of insertions so far. On the other hand, the similar part of Delete (line 5 to 7 of Algorithm 3) has the access pattern determined by $\mathsf{Sim}_{\mathsf{OTM}}$. Therefore, for any input query = (op, k, v, ref), the entire access pattern of PQ(query) is identical to that of PQ(Insert, ⊥, ⊥, ⊥). □

Consequently, there exists a simulator Sim(N) that simulates the access pattern of our PQ described in Algorithm 6.

[†]Note that we can ignore any *deletion of dummies* since it never affects to the priority order in MinTree.

Atsunori Ichikawa received his B.E. and M.E. degrees from Tokyo Institute of Technology in 2015 and 2017, respectively. Since 2017, he has been with NTT Corporation. His current interests are cryptography and information security.

Wakaha Ogata received the B.S., M.E. and D.E. degrees in electrical and electronic engineering in 1989, 1991 and 1994, respectively, from Tokyo Institute of Technology. From 1995 to 2000, she was an Assistant Professor at Himeji Institute of Technology. Since 2000 she has been working for Tokyo Institute of Technology, and now she is a Professor from 2013. Her current interests are cryptography and information security.