An Efficient Data-Independent Priority Queue and its Application to Dark Pools

Sahar Mazloom
J.P. Morgan AI Research
New York, NY, United States
sahar.mazloom@jpmorgan.com

Benjamin E. Diamond*
Coinbase
New York, NY, United States
benediamond@gmail.com

Antigoni Polychroniadou
J.P. Morgan AI Research
New York, NY, United States
antigoni.polychroniadou@jpmorgan.com

Tucker Balch
J.P. Morgan AI Research
New York, NY, United States
tucker.balch@jpmorgan.com

ABSTRACT

We introduce a new *data-independent priority queue* which supports amortized polylogarithmic-time insertions and constant-time deletions, and crucially, (non-amortized) constant-time *read-front* operations, in contrast with a prior construction of Toft (PODC'11). Moreover, we reduce the number of required comparisons. *Data-independent data structures*—first identified explicitly by Toft, and further elaborated by Mitchell and Zimmerman (STACS'14)—facilitate computation on encrypted data without branching, which is prohibitively expensive in secure computation. Using our efficient data-independent priority queue, we introduce a new privacy-preserving dark pool application, which significantly improves upon prior constructions which were based on costly sorting operations.

Dark pools are securities-trading venues which attain ad-hoc order privacy, by matching orders outside of publicly visible exchanges. In this paper, we describe an efficient and secure dark pool (implementing a full continuous double auction), building upon our priority queue construction. Our dark pool's security guarantees are cryptographic—based on secure multiparty computation (MPC)—and do not require that the dark pool operators be trusted. Our approach improves upon the asymptotic and concrete efficiency attained by previous efforts. Existing cryptographic dark pools process new orders in time which grows *linearly* in the size of the standing order book; ours does so in polylogarithmic time. We describe a concrete implementation of our protocol, with malicious security in the honest majority setting. We also report benchmarks of our implementation, and compare these to prior works. Our protocol reduces the total running time by several orders of magnitude, compared to prior secure dark pool solutions.

KEYWORDS

secure computation, privacy-preserving dark pools, data-independent data structure

* Work done in part while at J.P. Morgan.

Proceedings on Privacy Enhancing Technologies 2023(2), 5–22

https://doi.org/10.56553/popets-2023-0038

1 INTRODUCTION

One cannot implement priority queues naïvely—say, using the standard *binary heap* construction—in MPC (secure Multi-Party Computation) or FHE (Fully Homomorphic Encryption)-based environments, as Toft [Tof11, §5] observes. Indeed, these computing environments operate on encrypted or secret-shared data, and cannot execute branching code directly. So-called *data-independent data structures*—first identified explicitly by Toft, and further elaborated by Mitchell and Zimmerman [MZ14]—serve exactly to accommodate this constraint. To evade the necessity of expensive linear-time insertions in MPC or FHE, we introduce a data-independent data structure which admits better asymptotic complexity than that of prior work [Tof11]. In the taxonomy of [MZ14, §2], Toft's work presents, in fact, an efficient, *deterministically data-independent* priority queue.

We pause to emphasize the difference between *data-independent data structures* and *oblivious data structures*, in the sense of, e.g., Shi [Shi20]. Oblivious data-structures hide access patterns from a malicious RAM. In the data-independent setting, even the "CPU"—that is, the entity which executes the algorithm—is itself deprived of the inputs' plaintexts. Whereas oblivious algorithms must avoid *revealing* anything on the basis of their branching patterns, data-independent algorithms cannot branch at all. We refer to Mitchell and Zimmerman for a thorough comparison [MZ14]. In the setting of MPC, data-independence is the most suitable paradigm; indeed, oblivious algorithms are less suitable, since the participants in an MPC protocol possess no information about the data on which the protocol operates (and cannot branch directly).

We show by construction that since it is not possible to branch, an operation cannot traverse the queue by a path that depends on the data. Therefore, splitting into buckets as it is introduced by Toft [Tof11] allows us to ignore the data and pay attention only to the priorities of the data. In our work we maintain these invariants from Toft but improve its efficiency. Our main argument about the combination of oblivious algorithms and MPC is that they are not as efficient as the ones of using deterministic data independent data structures. In particular combining MPC and oblivious algorithms, such as the work of [Shi20], introduces an overhead of $O(\log N)$ per min-extraction operation where N denotes the maximum number of elements the priority queue can store. Unlike the other oblivious MPC solutions which all incur $O(\log N)$ overhead for extraction,

our crucial contribution is a "constant time" extraction operation on our data independent priority queue.

1.1 Priority Queues in Dark Pools

We observe that buy and sell lists in continuous double auctions are naturally conceived as (respectively, max and min) *priority queues*, where "priority" is given throughout by price. In particular, a *continuous double auction* is a standard mechanism for asset exchange, implemented by all major stock markets. This mechanism accepts *buy* and *sell* orders—for each among a number of assets—from multiple market participants. Each order specifies an asset, along with a *direction* (buy or sell), a desired trading *volume*, and a *price* (indicating the "worst" price at which the participant would accept an exchange). For instance, a client can submit a sell order for 1000 shares of the Apple stock with minimum price of $100 per share. The market venue "matches" compatible orders, which, by definition, have opposite directions, and for which the price of the buy order (the "bid") is at least the price of the sell order (the "ask"). We refer to [BBDG18, §3] for details on this fundamental mechanism.

Standard continuous double actions (CDAs), however, leak information in at least two related ways. For one, orders which do not immediately fill reside in the order book in a public manner. On the other hand, *all orders* (i.e., regardless of whether or not they immediately fill) are necessarily exposed, before being processed, to the market's operators, who in turn can act in advance of them. *Front-running*—the illegal practice whereby brokers act preemptively upon their clients' orders—remains a persistent threat, and has been the subject of numerous federal enforcement actions; we refer to [CSA19] for a thorough bibliography on front-running. In response to this challenge *dark pools* have arisen which are trading venues running CDAs in which submitted orders are hidden from public view. Only the operator(s) of the dark pool can see the submitted orders. Dark pools now account for 14% of trading volume in the United States (we refer to [BRW17] for background). Yet while dark pools (in theory) eliminate the first source of leakage identified above, they do not prevent the second, as their operators likewise gain privileged access to their clients' hidden orders. Dark pools operators too have been targeted by law enforcement [CSA19] for front-running.

A recent line of research has attempted to protect the information contained in orders *cryptographically*. The systems described in these works allow users to submit orders in an "encrypted" form; the dark pool operators then compare orders "through the encryptions", unveiling them only if matches occur. Specifically, a proposal of Cartlidge, Smart and Alaoui [CSA19] secret-shares order information across a number of participating servers which take the role of the dark pool operators. These servers check new orders for matches using the "SPDZ" secure multi-party computation protocol of [DPSZ12], revealing the orders' contents only when indicated. For each new order processed, [CSA19] requires that a number of messages which grows quadratically in the number of participating servers be exchanged; it also requires a number of *rounds* of communication which grows linearly in the number of standing orders. Asharov et al. [AJLA+12, ABPV20] proposed a general approach based on the use of threshold fully homomorphic encryption. Their dark pool accepts orders encrypted under a "master" public key, and matches them homomorphically; matched orders are finally threshold-decrypted. The communication of [ABPV20]'s approach is only linear in the number of participants (for each match processed), and moreover is independent of the overall size of the order book. It also supports t-out-of-n decryption (for arbitrary t). This approach conveys an additional benefit whereby its architecture is "star-shaped", and clients need to communicate only with the service provider (and not with each other). While [CSA19] and [ABPV20] differ in their underlying cryptographic mechanisms, both employ similar—and naïve—sorting (by the price of the order) and searching algorithms. In particular, both protocols' online order-processing algorithms incur linear-time passes, either during order insertion (in the case of [CSA19]) or during match evaluation (in the case of [ABPV20]).

1.2 Our Contributions

Dark Pools. In order to avoid the expensive use of sorting of prior privacy-preserving dark pools [CSA19, ABPV20], we introduce to the dark pool setting the use of *data-independent data structures*.

As we have already mentioned buy and sell lists are naturally conceived as (respectively, max and min) *priority queues*, where "priority" is given throughout by price. Matching a new order requires access to the front-most order of opposite type (or *direction*), as well as the ability to remove this order; finally, after being processed, each new order must be inserted into its own queue. The prior work of [CSA19] represents both queues as sorted lists, and inserts elements using linear-time "insertion sort"-style passes. Meanwhile, [ABPV20] represents both queues as unsorted lists, and extracts elements using linear-time searches. We introduce a new data-independent priority queue and our adaptation of Toft's queue, which finally, supports (amortized) polylogarithmic-time operations, as the costs of all operations are shown in Table 1 (see Theorem 3.1). Furthermore, in this work, we build on [CSA19], by introducing data-independent data structures to the MPC setting to achieve better asymptotic complexity (see Table 2 for a detailed comparison).

Data-Independent Priority Queues. Toft's queue, unfortunately, supports only the insertion and removal of elements, and admits no constant-time 'read-front' operation. This capability is crucial in many applications, and in particular in our dark pool construction. Of course, read-front can easily be 'emulated' by performing a removal and an insertion in succession. On the other hand, this approach fails to be constant-time, and moreover is unacceptably inefficient in practice. In fact, an efficient read-front operation is surprisingly difficult to incorporate directly into Toft's construction. We thus introduce a substantially modified priority queue in which we are able to significantly reduce the number of comparisons (which are costly) required to perform different operations on Toft's priority queue. In a nutshell, we require no comparisons for the read-front operations and amortized constant comparisons for deletion operations. We do so by maintaining a (tree-like) structure of sorted subsequences which we process in a different way than Toft in order to avoid and minimize the comparison operations. In particular, our queue allows for $O(\log^2 N)$-time insertions as Toft's queue. Our queue's deletions, on the other hand, require *zero* amortized cost (in that they can be fully funded using prior insertions);

moreover, its read-front operation requires zero comparisons *even in the worst case*. We stress that worst-case constant time is stronger than amortized constant time. Section 3 describes our approach.

1.3 Implementation

We will describe an end-to-end implementation of our framework including the queue data structure, fully done in MPC. We implement our scheme in the honest majority three-server setting (three-dark pool operators), tolerating one corruption in the presence of a malicious adversary, as the security model in [CSA19]. The underlying MPC scheme is an efficient honest-majority MPC scheme based on Replicated Secret Sharing and is introduced by [ADEN19] which supports secure computations over rings $\mathbb{Z}_{2^k}$. We use the MP-SPDZ library [Kel20] to implement our scheme. To demonstrate the performance of our privacy-preserving dark pool construction, we conduct several experiments in order to benchmark our performance with two other schemes, [CSA19] and our implementation of our proposed dark pool, constructed based on threshold Fully Homomorphic Encryption (tFHE). The result of these benchmarks are given in Tables 3, 4, 5, and show that our construction outperforms other solutions by several orders of magnitude. More concretely, for the case where the number of incoming orders is 30, our construction can process almost 20 transaction per second, whereas the FHE-based solution can only process almost 0.07 orders per second. And in the case that there is already a sorted order book as in [CSA19], assuming the scenario that the incoming buy order gets matched with one sell order, the throughput of our system is 136 vs 0.6 as in [CSA19]; considering the throughput as the number of transactions per second.

1.4 Related Work

The MPC-based construction of [CSA19] derives its security by separating the system's *service operator/provider* into several (e.g., three) distinct entities, whose collusion would void the system's security guarantees. [CSA19] proposes, in fact, that a regulatory body can be given control of one of the servers. The fully homomorphic approach [ABPV20], on the other hand, imposes a computational burden on a single server in a star topology network in which clients only communicate with the server. However, the concrete efficiency of FHE schemes is notoriously slow. While our data-independent data structure-based approach, we emphasize, is generic—it can equally serve both settings. We pursue the MPC-based approach in this paper.

The work of Massacci et al. [MNN+18] considers a distributed Market Exchange for futures assets which has a different functionality from the one considered in dark pools. Part of the futures exchange functionality includes a double auction as in the dark pools but the authors consider a different topology model which is only feasible for a small number of traders. They run a protocol across all the traders in which all traders have to participate in all the steps of the protocol, unlike our model where traders participate only in the registration phase in which they just secret share their input to the operators. Also, orders are not fully concealed; in particular an aggregated list of all waiting buy and sell orders is revealed which is not the case in our dark pool. The authors leave as an open problem to extend their functionality to the dark pool setting where the orders are not fully public.

Note that there are works which propose dark pool constructions on the blockchain [NMKW21, GY21, BHSR19] which is not the focus of our work. Moreover, these solutions have different guarantees and security goals.

2 BACKGROUND AND DEFINITIONS

In Section 2.1 we give background material on priority queues and in Section 2.2 on dark pools.

2.1 Priority Queues

In this section, first we recall the general notion of a *priority queue* (following the classic treatment of Cormen, Leiserson, Rivest and Stein [CLRS09]) and introduce the operations supported by our priority queue in this work. Abstractly, each priority queue Q maintains a *direction* d (either Min or Max), and supports the following external operations:

- Q.Initialize(d): Assigns the direction $Q.d$, and initializes Q to empty.
- Q.Insert(x): Adds x to the Q.
- Q.Size(): Returns the size of Q.
- Q.Extract-Front(): Returns and removes the highest-priority element from Q.
- Q.Front(): Only returns the highest-priority element from Q, without removing the element.

We assume that the operation Q.Insert(x) is *stable*, in the sense that the oldest (i.e., first inserted) among equally-keyed elements is necessarily prioritized. In both Q.Extract-Front() and Q.Front() operations, the highest-priority element directly depends on the direction of the priority queue, so that if $Q.d$ = Max they return the largest element, and if $Q.d$ = Min they return the smallest element. The important difference between these two operation–which also relates to our contribution–is that, Q.Extract-Front() returns and removes the element from the queue (which affects the structure of the queue), whereas Q.Front() only returns the value of the element and does not remove it from the queue. As a side note, our queue does not allow element priorities to be changed.

Here in this preliminary section, we only declare the main operations designed for our priority queue, and more comprehensive explanations will be given in following section where they are being utilized. The following internal operations are used to build the aforementioned external operations;

- Q.Flush($args$): Move and relocates the data elements inside Q. The arguments determine which elements and how they should be flushed throughout the Q.
- Q.Retrieve($args$): Arranges the Q's contents, and also returns the requested element from Q. The arguments determine which elements at what location should be processed.
- Merge($\cdot$, $\cdot$): Merges two pre-sorted lists and converts them to a larger sorted list, where its length is the sum of their initial lengths.
- Merge-Split($\cdot$, $\cdot$): Merges two pre-sorted lists and splits them to two separate sorted lists with the same size as their initial sizes, whose concatenation re-constructs the merged list.

It is important to note that our priority queue has a special internal structure, where it consists of several sub-queues. These sub-queues unlike the ones used in previous work by Toft [Tof11], shown in Figure 1(a)[1], are not regular FIFO queues; instead we use a basic "double-ended queue"–that is a generalized version of a FIFO queue–which allows insertion and removal of elements from both ends of the queue. The standard double-ended queue supports four basic operations in order to access the data from each end, as depicted in Figure 1(b). Q.Push-Front(x) inserts the element x at the front end of the queue, Q.Push-Back(x) inserts the element x to the back end of the queue. Q.Pop-Front() deletes an element from front end of the queue, and Q.Pop-Back() deletes an element from back end.

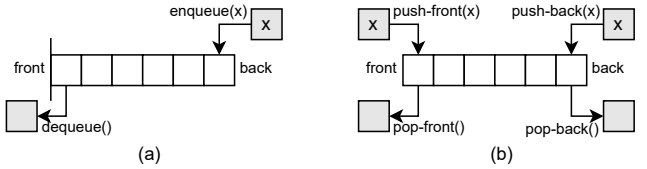

Figure 1: The structure of internal queues used as building blocks in constructing priority queues (a) The priority queue by Toft [Tof11] uses FIFO queues as building block, (b) Our data-independent priority queue uses double-ended queues as its building block.

2.2 Dark Pool Model and Functionality

We start by defining our model and then present the dark pool functionality.

2.2.1 Threat Model & Network Topology. Consider n parties $P_1, \ldots, P_n$ that hold private inputs $x_1, \ldots, x_n$ and wish to compute some arbitrary function $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$, where the output of P_i is y_i. MPC [Yao86, GMW87, BGW88, CCD87] enables the parties to compute the function using an interactive protocol, where each party P_i learns exactly y_i, and nothing else.

A semi-honest adversary (also known as "honest-but-curious" or "passive"), follows the protocol specification but may attempt to learn secret information about the private information of the honest parties from the messages it receives. *A malicious adversary* (also known as "active") may, in addition, deviate from the protocol specification and follow any arbitrary behavior.

For the dark pool application, we run an MPC protocol in the pre-processing model across three computation servers, S_1, S_2, S_3 to emulate the operator of the dark pool. Each client P_i is connected to the servers, S_is, and can submit his/her orders in a secret shared form. For our application, we consider a three-server maliciously secure protocol in the honest majority setting with one corruption and security with abort; it is the same threat model as [CSA19]. Note that in this threat model, clients, P_i, only secret share their private input (orders) with computation servers, S_i, and do not participate in the computation themselves. No single operator can recover the secret shared orders. The model is depicted in Figure 2. The use of our new data independent priority queue is not tight to any cryptographic technique, the values of the queue can remain private either using secret sharing, homomorphic encryption etc.

[1]The operations enqueue(x) and dequeue() are the standard operations in FIFO queue, and we use a solid line at the front of the queue to emphasize that the data cannot be inserted to the front of the queue; we use the same solid line in sub-queues shown in Figure 3.

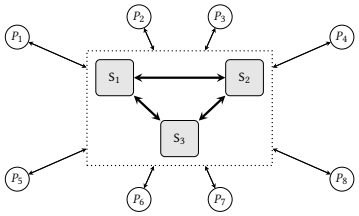

Figure 2: Three-server privacy-preserving dark pool.

2.3 Secret Sharing

We recall the general notation for secret sharing [Sha79].

Definition 2.1 (Secret-sharing)**.** Let $\mathbb{F}$ be a finite field and let $n, t \in \mathbb{N}$. A pair of algorithms $\mathscr{S}_t^n = (\text{Share}, \text{Recover})$ where Share is randomized and Recover is deterministic, are said to be a secret-sharing scheme if for every $n, t \in \mathbb{N}$, the following conditions hold:

- Reconstruction: For any set $\mathcal{T} \subseteq \{1, \ldots, n\}$ such that $|\mathcal{T}| > t$ and for any $s \in \mathbb{F}$ it holds that
$$\Pr[\text{Recover}(\text{Share}_{\mathcal{T}}(s, n, t)) = s] = 1$$
where $\text{Share}_{\mathcal{T}}$ is the restriction of the outputs of Share to the elements in $\mathcal{T}$.
- Privacy: For any set $\mathcal{T} \subseteq \{1, \ldots, n\}$ such that $|\mathcal{T}| \leq t$ and for any $s, s' \in \mathbb{F}$ it holds that
$$\text{Share}_{\mathcal{T}}(s, n, t) \equiv \text{Share}_{\mathcal{T}}(s', n, t)$$
where we use $\equiv$ to denote that two random variables have the same distribution.

2.4 Order Books and Dark Pool Functionality

We now describe the abstract functionality which our dark pool implements. An *order* x consists of a *name* n (an identifier reflecting who submitted the requested order), a *direction* d (either Buy or Sell), a *price* p and a *volume* v (both real numbers). Abstractly, our *dark pool* functionality DP maintains two priority queues, one for the Sell orders, called $\mathcal{S}$, and one for the Buy orders, called $\mathcal{B}$; each queue contains orders and keyed by the price of the orders, and implements the following operations:

- $\mathcal{F}_{\text{main}}$.Initialize():
Calls $\mathcal{B}$.Initialize(Max) and $\mathcal{S}$.Initialize(Min) keyed by the price p.
- $\mathcal{F}_{\text{main}}$.Process($x$): Calls the appropriate matching algorithm on order x.

We also assume access to the routine Execute that executes the trade if a match between a buyer and a seller is found. Depending on the order of arguments, it can represents a buy transaction or a sell transaction; we define both of them as follows:

- Execute(n_b, n_s, v, p): Executes a trade in which n_b buys v units from n_s, at the price p.

- Execute(n_s, n_b, v, p): Executes a trade in which n_s sells v units to n_b, at the price p.

The dark pool consists of two main functionalities $\mathcal{F}_{\text{Match-Buy}}$ and $\mathcal{F}_{\text{Match-Sell}}$, that are presented in Functionality 2.3 and Functionality A.1. In what follows we describe our main functionality assuming a new order x enters the dark pool.

FUNCTIONALITY 2.2 ($\mathcal{F}_{\text{main}}$–Dark Pool Functionality)**.**
Upon initialization, $\mathcal{F}_{\text{main}}$ initializes two empty queues, $\mathcal{B}$ for the buy orders and $\mathcal{S}$ for the sell orders and an empty list $\mathcal{T}$ for the orders to be executed.
Input: Buy and sell orders processed one-by-one. For simplicity of exposition, we denote each individual order by x.

1: **procedure** $\mathcal{F}_{\text{main}}$.Initialize()
2: $\mathcal{B}$.Initialize(Max)
3: $\mathcal{S}$.Initialize(Min).
4: $\mathcal{T}$:= []
5: **procedure** $\mathcal{F}_{\text{main}}$.Process(x)
6: **if** $x.d$ = Buy **then** call $\mathcal{F}_{\textbf{Match-Buy}}(x)$
7: **else** call $\mathcal{F}_{\textbf{Match-Sell}}(x)$

In Functionality 2.3 we present $\mathcal{F}_{\text{Match-Buy}}$. At the beginning of $\mathcal{F}_{\text{Match-Buy}}$, when a new buy order arrives, if the sell order queue is empty, then it simply inserts the incoming buy order to the buy queue. But in case the sell queue is not empty, if the price of the incoming buy order is more than the price of the sell order standing at the Front of the sell queue, it indicates that there is a potential match. In that case, it finds the minimum volume between the incoming buy order and Front order in the sell queue, followed by storing the matching order on the list $\mathcal{T}$. In the next step, it computes the remaining volume of the Front sell order. If the remaining volume is 0, then it calls the function Extract-Front to remove the Front sell order from the sell queue and consequently re-arranges the queue. It also computes the remaining volume of the incoming buy order; if it has no remaining volume, meaning it is fully filled, the while loop ends and that buy order is added to the buy queue with zero price. We note that, if a match takes place, the price of the Front order is used in the match, and not the price of the *incoming* order. This is an important—and standard—feature of continuous double auctions (see for example [BBDG18, §3.1.8]).

FUNCTIONALITY 2.3 ($\mathcal{F}_{\text{Match-Buy}}$–Matching Buy Functionality)**.**
Input: A buy order x.

1: **while** $\mathcal{S}$.Size()! = 0 **and** $x.p \geq \mathcal{S}$.Front().p **do**
2: $v \leftarrow \min(x.v, \mathcal{S}.\text{Front}().v)$
3: $\mathcal{T}$.Insert($x.n, \mathcal{S}.\text{Front}().n, v, \mathcal{S}.\text{Front}().p$)
4: $\mathcal{S}.\text{Front}().v \mathrel{-}= v$
5: **if** $\mathcal{S}.\text{Front}().v = 0$ **then** $\mathcal{S}$.Extract-Front()
6: $x.v \mathrel{-}= v$
7: **if** $x.v = 0$ **then** $x.p = 0$
8: $\mathcal{B}$.Insert(x)
9: **if** $\mathcal{T}$.Size()! = 0 **then** Execute($\mathcal{T}$)

A similar procedure happens when a sell order enters the dark pool, as presented in $\mathcal{F}_{\text{Match-Sell}}$ in Functionality A.1 given in Appendix A.

Our secure construction hides each order's sender, price, and amount, but *not* its symbol (i.e., which asset pair it concerns) or its direction (whether it's a buy or a sell). This corresponds to the setting of [CSA19]. The leakage of the orders in $\mathcal{T}$ is unavoidable since the orders have to be executed by the operators of the dark pool. We also conceal whether the incoming new order x was fully matched or not. In particular, in *line* 7 if an order is fully matched it is inserted in the queue with the minimum 0 price for the case where x is a buy order. (Likewise, the order is inserted with the maximum ∞ price if x is a sell order). Since the buy order is inserted in the queue at the lowest priority, there is no leakage on whether it was fully matched or not. Jumping ahead, our secure approach *also* conceals whether or not the last processed standing order in $\mathcal{S}$, to be matched with x, is fully filled (or $\mathcal{B}$ for the $\mathcal{F}_{\text{Match-Sell}}$ functionality), in addition to whether or not the new order fully fills. This contrasts with [CSA19], which hides *only* whether the incoming new order fully fills, but does not conceal the final processed standing order's status.

3 AN IMPROVED DATA-INDEPENDENT PRIORITY QUEUE

In order to understand how we improved the construction of a data-independent priority queue, first we need to recall how a data-independent priority queue works by describing an existing priority queue, by Toft [Tof11].

3.1 Overview of Toft's Queue

Toft [Tof11] constructs a priority queue which is *deterministically data-independent*, in a sense described explicitly by Mitchell and Zimmerman [MZ14, §2]. Before going to technical details, first we need to understand the intuition behind this data-independent priority queue. The primary goal in any priority queue is to have all the data stored in a list D in sorted order (based on their priority). However, sorting the list is an expensive operation, if done naively. Toft's queue, inspired by the bucket heap of Brodal et al. [BFMZ04], operates with the aid of an auxiliary data structure called buffer B (which intrinsically is a FIFO queue, Figure 1(a)). The high level idea is simple; rather than inserting d data elements directly into the queue to their correct locations (according to their priority), which is very expensive, the data elements will be placed temporarily into the buffer as they arrive, until there are sufficient data elements available in the buffer to be inserted to the queue D at once. Inserting the elements this way pays for the combined cost of all insertions.

Technically, the data is split into sub-queues, called buckets $\boldsymbol{Q}.D_0$, $\boldsymbol{Q}.D_1$, ..., such that the elements of $\boldsymbol{Q}.D_i$ are greater (higher priority) than those of $\boldsymbol{Q}.D_{i+1}$, assuming the sorting order is descending by priority. The size of each bucket doubles with each step, meaning that the size of bucket in level i, $\boldsymbol{Q}.D_i$, is double the size of the bucket in the upper level, $\boldsymbol{Q}.D_{i-1}$. In addition to this, the bucket B is also splits to sub-queues, such that at each level i, there is a buffer $\boldsymbol{Q}.B_i$ of the same size as $\boldsymbol{Q}.D_i$, as depicted in Figure 3. Both of these (FIFO) queues, D and B, consist the priority queue $\boldsymbol{Q}$.

Now we need to understand how the data is being processed and sorted inside this priority queue. Figure 3 provides an example of a priority queue consisting of buckets and buffers; where (a) shows the initial state of the queue when it is empty, and there is

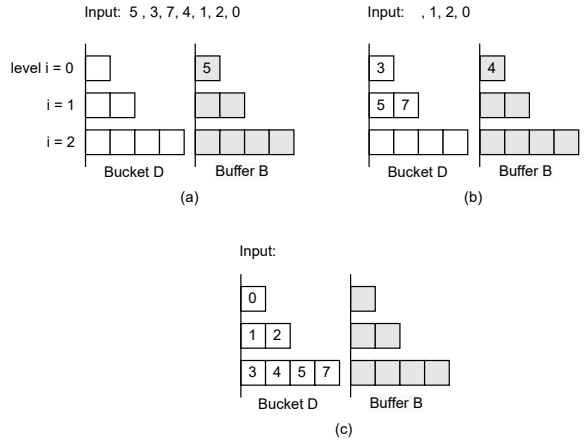

Figure 3: Buckets D_i and Buffers B_i as building blocks of the priority queue by Toft [Tof11]. (a) initial state of the queue when it is empty and the input list $(5, 3, 7, 4, 1, 2, 0)$**, (b) state of the queue after inserting some elements into the queue, and (c) state of the queue after all input elements are inserted into the queue.**

an input list $(5, 3, 7, 4, 1, 2, 0)$ to be inserted to the queue; and (b) state of the queue after inserting some elements into the queue, and (c) shows the state of the queue after all the input elements are inserted to the queue in their correct locations. We provide a step by step explanation of this example in Appendix B.1. It is important to note that the size of buffers and buckets at each level $i + 1$ is double their size at level i, later we will explain why this property is important and should be maintained throughout the entire queue structure.

There are two main operations on the priority queue, as introduced by [Tof11], which are Q.Insert(x) and Q.Extract-Front()[2]. There is also an important internal operation called Q.Flush that has a fundamental role in moving the data throughout the queue. The details of these operations are shown in Algorithms 1, 3 and 2, respectively. Here we explain each of these operations in more details[3]. There are two primitive operations that are invoked inside these algorithms, called Merge$(\cdot,\cdot)$ and Merge-Split$(\cdot,\cdot)$, as declared in section 2.1. The Merge operation takes two arguments, both need to be sorted lists with some known length l, and it has the ability to merge these two sorted lists to a larger sorted list. In case these input lists have different sizes, the shorter list should be padded—with some elements e_∞ which have the lowest priority than others—to become of equal length. This padding will be removed from the end of the merged list after the merge is done. Merge-Split$(\cdot,\cdot)$ has similar behaviour as Merge$(\cdot,\cdot)$, only difference is that, after it merges the two input sorted lists, it splits the merged list to two new lists whose concatenation re-constructs the merged list. The length of the new lists must be equal to the lengths of the original ones. The effect of a Merge-Split is that the most significant elements end up in one of the lists, while the least significant ones end up in the other one. Naturally, both new lists are still sorted. Both of these primitives can be constructed using Batcher's even–odd merge network (see e.g., Knuth [Knu98, §5.3.4]). Merging alone requires only $O(l \log l)$ conditional swaps in $O(\log l)$ rounds. Merge-Split has the same complexity as Merge (as the split merely only renames the variables).

The high level idea of Q.Insert(x) is that, the data is initially being inserted to the buffer, and as soon as the buffer becomes full, its content will be processed. To be more precise, as shown in Algorithm 1, when a new data element x arrives, it is first inserted into the top buffer $Q.B_0$, and eventually being processed and pushed downwards whenever the buffer gets full—or so called "flushed" in [Tof11]—through the buffers, until it reaches the buffer at the *correct level*, then it will be moved to the bucket in that level; the *correct level* refers to the level i in which the input element x is bigger than some elements p in the bucket at that level; more formally, x will be inserted to the bucket $Q.D_i$ if $\exists p \in Q.D_i$ s.t. $x \geq p$.

Algorithm 1 Q.Insert(x) [Tof11]: Inserting new data element x to the queue Q

```
1: if |Q.B_0| = 0 then
2:     Q.B_0 ← x
3: Q.Flush(0)                          ▷ Flush elements of level 0
```

In other words, the data arrives and being stored in the buffers, and when a buffer $Q.B_i$ gets full, elements of that buffer that are bigger than some elements $p \in Q.D_i$ are pushed to the bucket $Q.D_i$ which is at the same level i, and the rest of the elements are flushed down to the buffer at the lower level $Q.B_{i+1}$. Hence, according to Algorithm 2, there are two possible states when flushing a buffer $Q.B_i$: either i is the last level where data exists and $Q.D_i$ is empty; or (*line* 6) there is data in $Q.D_i$ or i is not the last level. In the first case, where the bucket $Q.D_i$ is empty and i is the last level, we may simply move the 2^i top elements into the data bucket $Q.D_i$ (*line* 2), since all the elements in the buffer $Q.B_i$ have lower priority than the elements in the buckets $Q.D_0$ to $Q.D_{i-1}$.

In the second case there may be data in the bucket $Q.D_i$ at level i or lower levels $i + 1, i + 2, \ldots$, which have lower priority than the elements of the buckets in upper levels $0, 1, \ldots, i - 1$. The Merge-Split ensures that the highest-priority elements of any level i end up in the bucket and the rest stay in the buffer. At this point it is guaranteed that the elements of $Q.D_i$ have higher priority than those of $Q.B_i$, hence the latter can be pushed into the buffer $Q.B_{i+1}$ when $Q.B_i$ gets full. A comprehensive example on how arriving data elements are being processed and inserted into the queue using the Flush operation is given in Appendix B.1.

Algorithm 3, describes the operation Q.Extract-Front(i) by Toft [Tof11]; it takes an argument i as an input, which indicates 2^i number of elements to be extracted from the level i and any level j where $i < j$. Note that if $i = 0$, this algorithm extracts the front-most element from the queue. A comprehensive example on how the data elements are being extracted from front of the queue is given in Appendix B.2. In order to extract the highest-priority element from the queue, Algorithm 3 returns the value of the front-most[4]

[2]In [Tof11], this operation is called Q.DelMin() which refers to returning and removing the smallest element from the queue, however we change the name to Q.Extract-Front to generalize it to support both Min and Max queues.

[3]For their correctness and security analysis please refer to [Tof11].

[4]We use the terms "front-most", "top-most", and "highest-priority" interchangeably.

Algorithm 2 Q.Flush(i) [Tof11]: Flushing buffer $Q.B_i$ at level i

1: **if** $|Q.D_i| = 0$ and i is last level **then**
2: $Q.D_i \leftarrow Q.B_i[1 : 2^i]$ ▷ Move 2^i elements from B_i to D_i
3: $Q.B_i \leftarrow Q.B_i[2^i + 1 : |Q.B_i|]$ ▷ Shift remaining elements
4: **if** $|Q.B_i| \geq 2^i$ **then**
5: Q.Flush(i) ▷ Flush elements at level i
6: **else**
7: $(Q.D_i, Q.B_i) \leftarrow$ Merge-Split$(Q.D_i, Q.B_i)$
8: $Q.B_{i+1} \leftarrow$ Merge$(Q.B_i, Q.B_{i+1})$
9: Set $Q.B_i$ empty
10: **if** $|Q.B_{i+1}| \geq 2^{i+1}$ **then**
11: Q.Flush($i + 1$)

element, that technically must come from a bucket. Only if there are no elements in any buckets can an element be taken from a buffer (line 5). It is clear that the highest-priority element will either be in the front-most, non-empty bucket or a buffer above that. Hence, starting with level 0, the buffers are flushed (merged with the buffer below) one after the other until a non-empty bucket is found (lines 6 and 10). Note that this simply merges buffered elements above any full buckets. Once a non-empty bucket is found, it is Merge-Split with the buffer at that level to ensure that it contains the 2^i highest-priority elements, not only at this level, but overall: buckets and buffers above are empty, and any element in the bucket has higher priority than any at a level below. The present bucket is then emptied into the buckets above, which fills them and leaves one element that can be returned – this task is trivial as the elements of the bucket are sorted, and the concatenation of the buckets above should be a sorted list. If all buckets are empty, then all buffers are merged until only a single non-empty one exists (at the last level, i).

Algorithm 3 Q.Extract-Front(i) [Tof11]: Return the top 2^i elements from level i and below

1: **if** $|Q.D_i| = 2^i$ **then**
2: $(Q.D_i, Q.B_i) \leftarrow$ Merge-Split$(Q.D_i, Q.B_i)$
3: **return** $Q.D_i$ and set it empty
4: **else if** i is the last level **then**
5: **return** $Q.B_i$ and set it empty
6: **else**
7: $Q.B_{i+1} \leftarrow$ Merge$(Q.B_i, Q.B_{i+1})$
8: Set $Q.B_i$ empty
9: **if** $|Q.B_{i+1}| \geq 2^{i+1}$ **then**
10: Q.Flush($i + 1$)
11: $Q.\tilde{D} \leftarrow Q$.Extract-Front($i + 1$)
12: **if** $|Q.\tilde{D}| \geq 2^{i+1}$ **then**
13: $Q.D_i \leftarrow Q.\tilde{D}[2^i + 1 : 2^{i+1}]$
14: $Q.B_{i+1} \leftarrow Q.\tilde{D}[2^{i+1} + 1 : |Q.\tilde{D}|]$
15: **return** $Q.\tilde{D}[1 : 2^i]$
16: **else**
17: **return** $Q.\tilde{D}$

The data is being processed (whether inserted or extracted) in a sorted order as described above, however there are two fundamental constrains or so called *invariants*. The invariants need be maintained constantly throughout the queue to have a priority queue, and all the operations we described above maintain them after they finish. We follow [Tof11, §5.2] in defining these invariants, with an exception for our improved queue which we indicate at the end. Namely, upon the termination of each external routine:

1) At each level i, $|Q.B_i| < 2^i$ and $|Q.D_i| \in \{0, 2^i\}$ (our invariant is more relaxed s.t. $|Q.D_i| \leq 2^i$).
2) At levels i, j where $i < j$, each element of $Q.D_i$ is of higher priority than any elements of either $Q.D_j$ or $Q.B_j$.

The first invariant of [Tof11] indicates that the buffers $Q.B_i$ must contain strictly less than 2^i elements, because as soon as it gets full, its data should be processed and moved to its appropriate location. On the other hand, the bucket $Q.D_i$ can contain 2^i elements, and it is either completely full or completely empty which means $|Q.D_i| \in \{0, 2^i\}$. However unlike Toft, we do *not* have this requirement and $|Q.D_i| \leq 2^i$. The second invariant assures that the elements of buffer $Q.B_j$ are less (have lower priority) than the elements of the higher-lying buckets, $Q.D_i$ where $i < j$. And finally, the data with highest-priority can be obtained by returning the contents of the top of the queue at the location $Q.D_0$.

Toft's priority queue admits the following complexities:
Its Q.Insert(x) and Q.Extract-Front() operations each take amortized $O(\log^2 N)$ time; Q.Empty() and Q.Initialize(d) each take $O(1)$ time. Operation Q.Initialize(d), assigns the direction d, and initializes Q to empty.

3.2 Technical Challenges

As previously introduced in 2.1, the Q.Front() operation only returns the element of the queue with the highest-priority without removing it from the queue. The operation is an interesting and useful operation when we only need to read the front-most element of the queue and do not need to remove it–which consequently can lead to change the order and structure of other elements in the queue. However, Toft's queue [Tof11] does *not*, unfortunately, support a Q.Front() operation, let alone one which runs in constant time. In fact, this operation is surprisingly challenging to incorporate directly into Toft's construction. For completeness, we briefly explain the nature of the difficulty, before introducing our own approach.

The highest-priority element of a queue satisfying the above invariants cannot in general be inferred directly from its internal state. This determination is possible only when the queue's topmost non-empty bucket $Q.D_{i^*}$ also satisfies $|Q.B_i| = 0$ for each $i \leq i^*$. In fact, the essential purpose of Toft's Extract-Front routine [Tof11, Prot. 2] is to *make* this state obtain, by repeatedly flushing buffers until it encounters a non-empty bucket. After finding one—say, $Q.D_{i^*}$—the algorithm [Tof11, Prot. 2] returns its front-most element, and *pack*s its remaining contents into the higher buckets (for which $i < i^*$). *Pack* intuitively means that the set of data elements all together are being moved.

Toft's deletion protocol—and its efficiency analysis—rely crucially on the size assumption $|Q.D_{i^*}| = 2^{i^*}$ (and the equality $2^{i^*} - 1 = \sum_{i=0}^{i^*-1} 2^i$). This assumption guarantees that, upon the Extract-Front's termination, all buckets $i < i^*$ become *completely* full, and, heuristically, that a deletion of level i^* need only occur once every 2^{i^*} deletions. This approach fails when the front-most

element only needs to be read, and not extracted (removed). In fact, the requirement $|Q.D_i| \in \{0, 2^i\}$ completely fixes the queue's bucket structure, at least given their combined capacity; so long as no element is removed, the buckets' arrangement cannot change. Leaving $Q.D_{i^*}$ as it is—that is, reporting its front-most element, without moving its contents upward—would violate [Tof11, Prot. 2]'s invariant on Extract-Front. Simply removing and re-inserting the front-most element would discard information (namely, the fact that the newly inserted element is of higher priority than all others). Another intermediate approach would pack $Q.D_{i^*}$'s front-most $2^{i^*} - 1$ elements and move into higher buckets, while stowing its last element in the buffer $Q.B_{i^*}$ (flushing it if necessary). This approach also yield a correct Extract-Front operation; however, it also discard information, and in particular the fact that $Q.D_{i^*}$'s last element is of higher priority than those buffers $Q.B_j$ for which $j > i^*$. Both approaches perform significantly worse than ours in practice.

3.3 Our Approach

We now describe our improved queue construction. We significantly modify the queue's internal algorithms, so as to facilitate our introduction of the Q.Front() operation. As previously introduced in 2.1, the Q.Front() operation returns the element of the queue with the highest-priority. In fact, we relax the requirement (see [Tof11, §5.2]) whereby each bucket must be either full or empty at all times, that is $|Q.D_i| \in \{0, 2^i\}$. We achieve this relaxation by replacing the internal sub-queues of Toft's priority queue with double-ended queues which allows to insert or remove the elements from both ends of the queue, rather FIFO queues that only allow the insertion at the back end and removal from front end of the queue. By relaxing this invariant, buffers do not require to be over loaded in order to be processed (flushed). We also develop appropriate generalizations of [Tof11, Prot. 1] and [Tof11, Prot. 2]. In fact, we separate the "flushing logic"—used by both Q.Insert(x) and Q.Extract-Front()—and the "retrieval logic"—used by Q.Extract-Front()—into distinct routines. This approach generalizes and unifies [Tof11].

To introduce our unified internal method Q.Flush(i, t, e), it is important to note that it accepts an additional parameter e, valued either Regular or Extract, and parameter t to record cumulative bucket size. This method, Q.Flush(i, t, e), flushes the buffer $Q.B_i$ at the level i. Q.Flush($0, 0,$ Regular) operates essentially as does [Tof11, Prot. 1], by flushing only over loaded buffers. If e equals Extract, then the routine flushes *all* buffers—i.e., even non-full ones—until encountering an adequately full bucket. Specifically, Q.Flush($0, 0,$ Extract) flushes buffers until finding a bucket i^* for which $\sum_{j \le i^*} |Q.D_j| \ge 2^{i^*}$. This condition exactly recovers that of [Tof11, Prot. 2] in case each $|Q.D_i| \in \{0, 2^i\}$. For technical reasons, we also require that $i^* > 0$ (see Subsection 3.4 for further discussion). Notably, these two procedures differ only subtly, and can be subsumed into a single method. The latter method Q.Flush($0, 0,$ Extract) serves to *free up* certain buckets $Q.D_i$, which can then be packed upwards. We abstract this latter logic into a second internal routine, which is almost *free* as it executes no comparisons. This routine, which we call Q.Retrieve(i, v), recursively packs the queue's contents using the vector-valued parameter v, and also returns the queue's front-most element. These internal operations can be summarized as follows, and more discussion will be given in Correctness Subsection 3.4:

- Q.Flush(i, t, e): Assumes inductively that any buffer in levels above i are empty, $|Q.B_j| = 0$ for each $j < i$; as well as that $t = \sum_{j<i} |Q.D_j|$. It ensures that $|Q.B_j| < 2^j$ for each $j \ge i$, as well as, if $e =$ Extract, that $|Q.B_j| = 0$ for each $j \le i^*$, where either $\sum_{j \le i^*} |Q.D_j| \ge 2^i$ and $i^* > 0$ or else $i^* = |Q.D| - 1$.
- Q.Retrieve(i, v): Assumes inductively that $|Q.D_j| = 0$ and $|Q.B_j| = 0$ for each level $j < i$, and that vector v gives a consecutive sequence of the queue's front-most elements, with size $|v| < 2^i$. Extends vector v so as to contain as many elements as 2^i of the queue's consecutive front-most elements, after packing any element with higher priority than those 2^i's, into those buckets $Q.D_j$ for which their level $j \ge i$.

Algorithm 4 Q.Insert(x)

```
1: if Q.Empty() then
2:     Q.D.Push-Back([])          ▷ Initialize the Bucket
3:     Q.B.Push-Back([])          ▷ Initialize the Buffer
4: Q.B_0.Push-Back(x)
5: Q.Flush(0, 0, Regular)
```

Algorithm 5 Q.Read-Front()

```
1: return Q.D_0[0]
```

Algorithm 6 Q.Extract-Front()

```
1: Q.Flush(0, 0, Extract)
2: assign v := []
3: Q.Retrieve(0, v)
4: return v[0]
```

Algorithm 7 Q.Flush(i, t, e)

```
1: (Q.D_i, Q.B_i) ← Merge-Split(Q.D_i, Q.B_i)
2: if i = |Q.D| − 1 then
3:     while |Q.D_i| < 2^i and |Q.B_i| ≠ 0 do
4:         Q.D_i.Push-Back(Q.B_i.Pop-Front())
5:     if |Q.B_i| = 0 then return
6:     Q.D.Push-Back([])
7:     Q.B.Push-Back([])
8: Q.B_{i+1} ← Merge(Q.B_i, Q.B_{i+1})
9: t += |Q.D_i|
10: if i > 0 and t ≥ 2^i then e = Regular
11: if |Q.B_{i+1}| ≥ 2^{i+1} or e = Extract then
12:     Q.Flush(i + 1, t, e)
```

For completeness, we record the additional algorithm 13, Q.Size(), given in Appendix A that returns the current size of the queue.

Algorithm 8 Q.Retrieve(i, v)

if $|Q.B_i| > 0$ **and** $(i < |Q.D| - 1$ **or** $|Q.D_i| > 0)$ **then**
  return
while $|Q.D_i| \neq 0$ **do** v.Push-Back($Q.D_i$.Pop-Front())
while $|Q.B_i| \neq 0$ **do** v.Push-Back($Q.B_i$.Pop-Front())
if $i < |Q.D| - 1$ **then**
  Q.Retrieve$(i + 1, v)$
while $|v| > 2^i$ **do** $Q.D_i$.Push-Front(v.Pop-Back())
if $i = |Q.D| - 1$ **and** $|Q.D_i| = 0$ **then**
  $Q.D$.Pop-Back()
  $Q.B$.Pop-Back()

3.4 Correctness

We begin by observing a basic, but important, fact, whereby, at any given moment, the concatenation of those *top-most* $Q.D_i$ for which $|Q.B_i| = 0$ gives a vector containing the queue's highest-priority elements. An exception is given when i is Q's "last level" *and* $Q.D_i$ is empty; in this case, $Q.B_i$ too can be concatenated (in this case there are no lower levels, so invariant 2) becomes vacuous). These facts, which directly follow from the invariants, are summarized by the *illustrative* algorithm 12, Q.Highest-Priority(), given in Appendix 12 (which we never call directly).

This free, read-only algorithm necessarily returns a (possibly empty) vector v consisting of a *contiguous subsequence* of Q's fully sorted contents, starting from its highest-priority element. In fact, the vector v returned by this algorithm gives the *longest* such contiguous subsequence whose correctness can be directly inferred from the queue's present state (i.e., without making additional comparisons). Our algorithms make essential use of this "longest contiguous sorted subsequence", as we now explain. The algorithm Q.Flush(i, t, e) operates in two modes. If e = Regular, then the function's purpose is merely to re-establish the first part of invariant 1), whereby the size buffer $|Q.B_i| < 2^i$ for each i. The routine proceeds by ensuring, for each i, that each element of bucket $Q.D_i$ is of higher priority than each element of buffer $Q.B_i$—by calling Merge-Split$(Q.D_i, Q.B_i)$—before pushing the contents of buffer $Q.B_i$ into the next buffer, and finally calling Q.Flush$(i + 1, t,$ Regular) recursively if $Q.B_{i+1}$ becomes over loaded. A special case arises when i is the queue's "last level", in which case, again by invariant 2), the contents of buffer $Q.B_i$ may be moved to bucket $Q.D_i$.

The parameter e = Extract, on the other hand, establishes a stronger guarantee; that is, it additionally serves to attempt to make the vector v "adequately long". Notably, this latter purpose can be achieved by subtly adjusting Q.Flush. Indeed, vector v can be lengthened simply by successively flushing *all* buffers $Q.B_i$ (i.e., regardless of whether they are over loaded $|Q.B_i| \geq 2^i$) until some i^* is reached for which $\sum_{i \leq i^*} |Q.D_i|$ satisfies an appropriate length condition; at this point, the flag is flipped to Regular in all subsequent recursive calls; the procedure may continue, but only for the purposes of maintaining invariant 1). Concretely, e = Extract flushes until encountering some bucket $Q.D_{i^*}$ for which $i^* > 0$ and $\sum_{i \leq i^*} |Q.D_i| \geq 2^{i^*}$. On one hand, these conditions together ensure that $|v| \geq 2$, so that $v[0]$—necessarily the queue's front-most element—can be returned, and moreover that $v[1]$ may be placed in $Q.D_0$. On the other hand, they also make v sufficiently long so as to "re-pack" all flushed buffers. This latter property, which is important for efficiency reasons, is discussed further in Subsection 3.5; we also discuss there the degenerate case in which no appropriate i^* is found. It remains to explain Q.Retrieve(i, v). The essential purpose of Q.Retrieve(i, v) is to "pack" the vector v upwards—wherever it resides—so that it comes to contiguously occupy the queue's highest buckets (it is essentially a recursive variant of Q.Highest-Priority(), which also rearranges). Specifically, Q.Retrieve$(0, [])$ packs the subvector $v[1:]$, and returns its front-most element $v[0]$. This algorithm maintains the queue's correctness, by definition of v.

The external routine Q.Insert(x) therefore inserts x into $Q.B_0$, before using Q.Flush$(0, 0,$ Regular) to re-establish 1). On the other hand, Q.Extract-Front() uses Q.Flush$(0, 0,$ Extract) to ensure that $|v| \geq 2$ (while moreover preserving 1)); finally, the latter routine calls Q.Retrieve$(0, [],$ Extract). This latter routine, among other things, *returns* $v[0]$ and *lifts* $v[1]$ into $Q.D_0[0]$. Both routines preserve all invariants, as well as the non-emptiness of $Q.D_0[0]$, which itself guarantees immediate access to the queue's front-most element. This concludes the explanation of correctness.

3.5 Efficiency

We now discuss the asymptotic efficiency of our priority queue. We follow the amortized analysis paradigm of [Tof11, §5.5], in which each "comparator module"—i.e., each including a comparison and two multiplexers (see [Knu98, §5.3.4])—incurs a unit cost. We also incorporate certain ideas from the *bucket heap* of [BFMZ04, §2], which itself inspired Toft's construction (see [Tof11, §1.1]). In particular, our potential function is inspired by theirs. A brief explanation on the potential function and the (coin) method used to analyze the performance cost of construction is given in Appendix C. As [BFMZ04] does, we use the letter q to denote the size of buckets $|Q.D|$ at any given time; this size is dynamically allocated (and is always less than $\log N$). We note that for exactly those $i \in \{0, \ldots, q-1\}$, either $Q.D_i$ or $Q.B_i$ is non-empty. We show that an $O(\log^2 N)$ amortized cost, paid up-front upon each insertion, suffices to fund all future deletions; as a result, the latter routine requires "eventually non-positive"—or, in other words, $O(1)$—amortized cost. By the correctness explanation above, Q.Front() requires $O(1)$ comparisons, even in the worst case (its complexity analysis is *not* amortized). These facts make our queue highly efficient, both in theory and in practice.

Theorem 3.1. *Consider an arbitrary sequence of intermixed queue operations, and denote by N an upper bound on the capacity attained by the queue throughout the sequence of operations. Then there exists a potential function for which Q.Insert(x) and Q.Extract-Front() have amortized costs of $O(\log^2 N)$ and $O(1)$, respectively, measured in comparator modules. Q.Front() has a worst-case cost of $O(1)$.*

For the proof, refer to Appendix C.1.

Remark 3.2. The proof of Theorem 3.1 (in Appendix C.1) shows that there is some flexibility in the choice of i^*. Indeed, the proof goes through so long as i^* is chosen so that $i^* > 0$, $\sum_{j \leq i} |Q.D_j| \leq 2^i$ for each $i < i^*$, and finally $\sum_{j \leq i^*} |Q.D_j| \geq 2^{i^*}$. Our approach simply uses the smallest i^* for which these conditions hold.

We summarize these asymptotic results in Table 1. We note that our queue is essentially asymptotically optimal, at least barring the use of sophisticated technology like the AKS network (see for example [Knu98, §5.3.4]). Indeed, our queue yields an $O(N \log^2 N)$-time, data-independent sorting algorithm in the obvious way.

Operation	Cost (in comparators modules)
Insertion	$O(\log^2 N)$, amortized
Extraction	$O(1)$, amortized
Read-Front	$O(1)$ worst-case

Table 1: Asymptotic performance of our priority queue.

We note that these bounds could be achieved rather trivially from Toft's queue, simply by increasing the constant factor hidden in the initial $O(\log^2 N)$ cost of insertion. Indeed, the total number of deletions which occur throughout a sequence of operations is bounded by the number of insertions; moreover, the total amount of read-front operations is also bounded by double the number of insertions (at least if one restricts to "non-redundant" read-front operations, which immediately follow a state change). Because each read-front, in Toft's queue, involves a removal and an insertion, this would require multiplying the initial constant factor by 6. Our queue, on the other hand, has superior constants factors, doesn't require an ad-hoc restriction on read-front operations, and finally admits a more "intrinsic" proof that the latter operations have 0 cost.

3.6 Queue Stability

Our queue construction, like Toft's, lack the "stability" property whereby equally keyed elements are prioritized in a *first-in, first-out* manner. In fact, Batcher's merge network itself fails to be stable. Moreover, the queue *remains* unstable even if Batcher's network is artificially made stable (i.e., by temporarily adding extra bits to the elements' keys before conducting the merge). In order to make our queue construction stable, the goal is to help it to preserve the relative order of elements with equal values. We provide a naïve solution to preserve the order of elements, by assigning them a sequence ID upon their arrival, such that these IDs indicate their relative order. Another alternative solution is to tag the orders with a timestamp, to indicate which elements arrived earlier if they have equal values, and hence provide stability for our queue. These approaches impose a cost, in that all comparisons must be made on numbers containing more bits. We do not currently see a more efficient solution.

4 APPLICATION: PRIVACY-PRESERVING DARK POOL

4.1 Our Construction

In this section, we show how to instantiate the functionalities $\mathcal{F}_{\text{Match-Buy}}$, $\mathcal{F}_{\text{Match-Sell}}$ and $\mathcal{F}_{\text{main}}$ of Subsection 2.4 in the MPC setting where the orders x are secret shared across three computation servers in the honest majority setting, and implement all priority queues using our improved queue. In Algorithm 9 we present our main protocol.

Algorithm 9 Π_{Main}–Server Dark Pool Protocol

Upon initialization, each Server initializes two empty queues, $\mathcal{B}$ for the buy orders and $\mathcal{S}$ for the sell orders and an empty list $\mathcal{T}$ for the orders to be executed.
Input: Each party P secret shares the volume and the price of its order x to the three servers invoking $(x_1, x_2, x_3) \leftarrow \text{Share}(x, 3, 1)$.
Each server S_i given x_i proceeds as follows:

1: **procedure** Π_{main}.Initialize()
2: $\quad \mathcal{B}_i$.Initialize(Max)
3: $\quad \mathcal{S}_i$.Initialize(Min)
4: $\quad \mathcal{T}_i := []$
5: **procedure** Π_{main}.Process(x_i)
6: $\quad$ **if** $x.d =$ Buy **then** call $\Pi_{\textbf{Match-Buy}}(x_i)$
7: $\quad$ **else** call $\Pi_{\textbf{Match-Sell}}(x_i)$

In Algorithm 10 and Algorithm 11 we present the buy and the sell matching protocols which run on secret shared values. The routine Execute, in this setting, accepts *secret shared* arguments; we assume that it internally performs all necessary share recovering operations (perhaps asynchronously). Execute is the only process that recovers the shares revealing the orders. However, this leakage is acceptable since the orders need to be executed.

Algorithm 10 $\Pi_{\text{Match-Buy}}(x)$

1: $t \leftarrow$ Null
2: **while not** $\mathcal{S}$.Empty() **and**
3: $\quad$ Recover($x.p \geq \mathcal{S}$.Front().p) **do**
4: $\quad v \leftarrow \min(x.v, \mathcal{S}.\text{Front}().v)$
5: $\quad \mathcal{T}$.Insert($x.n, \mathcal{S}$.Front().$n, v, \mathcal{S}$.Front().p)
6: $\quad \mathcal{S}$.Front().v −= v
7: $\quad x.v$ −= v
8: $\quad x.p \leftarrow (x.v = 0)\ ?\ 0 : x.p$
9: $\quad t \leftarrow \mathcal{S}$.Extract-Front()
10: $\mathcal{B}$.Insert(x)
11: **if** $t \neq$ Null **then**
12: $\quad t.p \leftarrow (t.v = 0)\ ?\ \infty : t.p$
13: $\quad \mathcal{S}$.Insert(t)
14: **if not** $\mathcal{T}$.Empty() **then** Execute($\mathcal{T}$)

4.2 Correctness

In each matching algorithm's main loop, we "recklessly" pop the front-most standing order from the stack (after matching it), and store it in the temporary variable t, in line 8. The event whereby the next iteration's condition *passes* implies in particular that $x.p$ was *not* set to an extreme value in line 9, and hence that $x.v \neq 0$; this in turn implies that $t.v = 0$ and that t's removal was (in retrospect) justified. The failure of the loop's condition, on the other hand, implies either that x was fully exhausted or that the next-most standing order's price simply failed to cross (or both, in the case of a simultaneous full match). In this event, we insert x, and also

Algorithm 11 $\Pi_{\text{Match-Sell}}(x)$

$t \leftarrow \text{Null}$
while not $\mathcal{B}$.Empty() **and**
    Recover($x.p \le \mathcal{B}$.Front().p) **do**
  $v \leftarrow \min(x.v, \mathcal{B}.\text{Front}().v)$
  $\mathcal{T}$.Insert($\mathcal{B}.\text{Front}().n, x.n, v, \mathcal{B}.\text{Front}().p$)
  $\mathcal{B}.\text{Front}().v \mathrel{-}= v$
  $x.v \mathrel{-}= v$
  $x.p \leftarrow (x.v = 0)\ ?\ \infty : x.p$
  $t \leftarrow \mathcal{B}$.Extract-Front()
$\mathcal{S}$.Insert(x)
if $t \neq \text{Null}$ **then**
  $t.p \leftarrow (t.v = 0)\ ?\ 0 : t.p$
  $\mathcal{B}$.Insert(t)
if not $\mathcal{T}$.Empty() **then** Execute($\mathcal{T}$)

re-insert the popped item t, after possibly changing its price to an extreme value. That said, we conceal whether there was a full or a partial match.

4.3 Security

Our approach leaks *only* those matches which occur and have to be executed in the list $\mathcal{T}$, as well as the fact of each *non*-final standing order's having been fully filled. This represents exactly the information leakage which is inherent to the setting of this subsection (where only price and volume are concealed), and is thus optimal. The remaining security properties reduce to those of the underlying MPC protocol based on secret sharing. Moreover, our priority queue provides a significant advantage and only leaks whatever the priority queue leaks given the organization of orders into a tree like structure. Namely, top-priced orders but maintaining the contents private since the orders are secret shared among three servers. The exact price is revealed only if there is a match. A linear scanned structure as in [CSA19] is not leak-free since an insertion sort is used which can leak info about the order of the prices.

We note that both [CSA19] and this work could be extended so as to hide the *direction* of each new order (i.e., whether it's a buy or sell), in addition to its sender, price, and volume. In effect, each client should submit—for each actual order—two secret-shared orders, one of whose prices is "extreme" (i.e, guaranteed not to trigger matches). On the other hand, this approach could be self-defeating, at least if matches are instantly revealed. The fact of *one* among two newly submitted orders' immediately inducing partial matches reveals, of course, which among the two was honest. This recourse, therefore, would be effective only when even the *non-dummy* order fails to immediately match. In our implementation (see Subsection 5), we do not include this feature.

4.4 Efficiency

We use N in what follows to denote the total number of orders processed throughout a period of amortization. The algorithms $\Pi_{\text{Match-Buy}}$ and $\Pi_{\text{Match-Sell}}$ each perform a number of MPC operations and queue extractions proportional to the number of matches executed, along with up to two insertions. Each insertion incurs $O(\log^2 N)$ amortized cost (where N reflects the appropriate queue); extractions are free, amortized. In one sense, the *worst-case* per-case per-order complexity depends on the number of "partial matches"—say, M—which occur while any given order is processed. On the other hand, M itself is bounded by N—even when aggregated across all orders—and so "disappears" throughout the course of amortization. We thus exclude M from our analyses of order-processing cost, which itself must be amortized. We preserve M in our analyses of messages and rounds, which are *not* amortized (they reflect worst case complexity per-order).

Our total order processing complexity is thus $O(\log^2 N)$ atomic operations per order. This contrasts favorably with [CSA19] and [ABPV20], which both take $O(N)$ time. (These latter protocols' analyses are *not* amortized; in fact, they respectively require $O(M + N)$ and $O(M \cdot N)$ operations per order in the worst case.) Each call to $\Pi_{\text{Match-Buy}}$ and $\Pi_{\text{Match-Sell}}$ results in at most one "dummy" (i.e., empty) order being added to one of the two lists, in addition to the actual unfilled order (except in the unusual case of a "simultaneous full match", after each such one dummy is added to *each* list). On the other hand, all non-final standing orders (which are necessarily fully matched) are removed. This situation—like that of [CSA19]—results in linear growth, *over time*, in the sizes of the two lists. We deem this acceptable, as dummies can be removed "overnight" using generic MPC when new orders are not submitted. How we add dummies is as follows: At the end of the sell or buy procedures, any remaining volume will be added to the corresponding buy or sell queue. For example if the remainder of a buy order is 0, to avoid leakage, we still add it to the queue, however we set the price to 0 and consider it as a dummy order, which can be removed later when the market is closed. For applications where this leakage is not crucial, dummies can be discarded. We summarize these results in Table 2. We write N for the total number of orders processed; P refers to the total number of parties in the system (i.e., the number of service providers in [CSA19], and the number of clients in the remaining two works). "Atomic operations" include integer operations (like comparison and addition) whose complexity is independent of M and N. "Messages" refers to those sent by the operators, not by clients. Each entry reflects cost *per order*, amortized in the first column only.

Protocol	Operations (Amortized)	Messages	Rounds
[CSA19]	$O(N)$	$O(P \cdot M \cdot N)$	$O(M \cdot N)$
[ABPV20]	$O(N)$	$O(P \cdot M)$	$O(M)$
This work	$O(\log^2 N)$	$O(P \cdot M)$	$O(M)$

Table 2: Performance comparison with other protocols.

5 IMPLEMENTATION AND EVALUATION

5.1 Implementation

We provide an end-to-end implementation of our framework including the queue data structure, fully in MPC. We use MP-SPDZ (Multi-Protocol SPDZ) compiler - introduced by [Kel20] - which is a fork from SPDZ-2 library and extends it to 34 protocols; it supports

commonly used security models as well as binary and arithmetic computation circuits. MP-SPDZ library supports several protocols, especially in the three-party setting which we used to instantiate our dark pool scheme; among the honest majority three party setting protocols we choose 'ps-rep-ring' which supports an efficient scheme introduced in [ADEN19], over the ring $\mathbb{Z}_{2^k}$ and uses replicated secret sharing. We set the statistical security parameter to 40[5] and the computation is done over a 64-bit ring. We conduct experiments in the three-party honest majority setting, tolerating one corruption in the presence of a malicious adversary. There are three-parties and each of them is deployed on a separate `c5.9xlarge` AWS instance, with 36 vCPUs and 72 GiB memory, located on the same AWS region, while they are connected through a network channel established by sockets, with 10 Gbps bandwidth. To implement the order-book, namely the buy and sell lists, we use 64-bit secure-integer (sint) data types to represent *name*, *price* and *volume*, which we aim to keep secret during the auction. We compare the performance of our privacy-preserving dark pool with two other schemes. The first benchmark scheme is introduced in [CSA19] as a CDA (Continuous Double Auction) method which has been originally implemented using SCALE-MAMBA library [ACC+21] (SCALE-MAMBA is another MPC library forked from SPDZ) [6]. The second benchmark scheme is our implementation of our proposed dark pool construction using threshold fully homomorphic encryption (tFHE) instead of MPC, that we have implemented using the GPU FHE library of [BDP20].

5.2 Experiments

Our first set of experiments, compares the performance of our privacy-preserving scheme versus the CDA method of [CSA19]. The benchmark scenario in [CSA19] is as follows: There is a pre-generated order book that contains a *pre-sorted* list of buy orders (bids) with size n and a *pre-sorted* list of sell orders (asks) with size m, in which bids are ordered by price descending, and asks are ordered by price ascending. When an incoming buy order arrives, the goal is to match it against the ask list. [CSA19] measures the performance of their scheme by calculating how much time it takes to match an incoming buy order with the sell orders in the sell order-book, which depends on how many sell orders are opened and filled (either fully or partially filled) by that incoming buy order. They considered 4 different scenarios, where the number of opened sell order are $s \in \{0, 1, 2, 3\}$. To mimic this scenario, the incoming buy order should be tailored such that it can match with specific number of sell orders in each scenario[7]. We extend this experiment by increasing the number of opened sell orders to $s \in \{2, 4, 8, ..., 1024\}$ and show the results in Table 5 given in Appendix D

[CSA19] algorithm runs in two main phases; Matching phase, where they try to match the incoming buy order against any offer on the sell list, and the Insertion phase, where they insert any remaining volume of the buy order to the buy list (meaning it is partially-fulfilled) according to its price. In case the remaining volume is zero (meaning it is fully-fulfilled), first they set the price to also be zero, then insert it to the buy list, which in this case the order will be inserted at the bottom of the buy list. In Table 6 in Appendix D, we provide the total run time (latency) as well as the run time of each phase separately. Note that the latency of the first phase of the [CSA19] algorithm, the Matching phase, depends on the number of fulfilled sell orders, whereas the latency of the Insertion phase depends on the size of the buy list and where the remaining of the buy order is being inserted into the buy list which depends on its price; in the best case when the price is high the remaining of the buy order will be inserted to the top region of the list, and in worst case when the price is low, or even zero (in case remaining volume is zero, we set the price to zero as well), it will be inserted at the bottom region of the list. Therefore in this experiment, the latency is shown as a range, where Insertion latency varies between $[0.8150 - 0.83214]$ seconds (to consider best and worst cases scenarios). The Total Latency which is the sum of the Matching latency and the Insertion latency, is also shown as a range. We re-created their experiments in our dark pool scheme to benchmark the results, with the same experimental parameters and setting where we use three computation parties, each deployed on an AWS instance. The result are given in Table 3.

The reported metrics are: Total *Latency* which is the total time takes to process transactions in each case (in seconds), *Approximate Latency per order* which is the total latency divided by the number of opened sell orders, and the *Throughput* which is one over the latency and measures the number of transactions per second. In all of these tests, we set the size of order books as $n = m = 40$, that means there are 40 pre-sorted bid orders in the buy list, and 40 pre-sorted ask orders in the sell list. The latency includes the time for both online and offline phases, and it is the sum of computation and communication costs. Note that the Total Latency is reported as a range to demonstrate the effect of Insertion Latency in the Total Latency. While the amortized cost of Insertion in our construction is $O(\log^2 N)$.

Protocol	Opened Sell Orders	Total Latency (S)	Approx Latency per Order (S)	Throughput
[CSA19]	0	1.567-1.584	1.567-1.584	0.631-0.637
	1	1.571-1.588	1.571-1.588	0.629-0.6363
	2	1.575-1.592	0.787-0.796	1.255-1.269
	3	1.579-1.598	0.526-0.532	1.878-1.898
This work	0	0.00516	0.00516	193.74
	1	0.00732	0.00732	136.56
	2	0.00937	0.00468	213.37
	3	0.01161	0.00387	258.34

Table 3: Performance of CDA method [CSA19] vs our privacy-preserving dark pool scheme. The latency is in seconds (S). The size of the book is chosen to be $n = m = 40$.

We extend the experiment by running it with larger order book, n = m = 1024, and increase the number of open sell orders, $s \in$

[5] 40 is the default statistical security parameter for the underlying protocol we use from [CDE+18]. MP-SPDZ uses a trick in SPDZ2K [CDE+18] where it takes a ring mod $2^{(k+\text{sec})}$ to work (mod 2^k). Note that [CSA19] also uses 40 bits.

[6] We chose to implement our scheme using MP-SPDZ over SCALE-MAMBA since it provides variety of protocols under different security models, and supports computations over extended rings as well as prime fields. It gives more flexibility to benchmark our platform, and also provides better performance for the functionalities we need.

[7] For example, when $s = 0$, it means the buy order cannot match with any of the sell orders in the sell order-book

$\{2, 4, 8, ..., 1024\}$. We report the running time and the variance [8] of our results in our experiments, since we reported the average of 5 runs of the algorithm in each benchmark in Table 5 given in Appendix D. The result of this experiment shows that as we increase the number of opened orders, the latency also increases over time, however the growth was close to linear and no sudden jump was observed. In order to increase the number of requests significantly (to help finding the peak performance of our solution), we need to upgrade the computation machines to stronger machines with higher RAM capacity, to be able to run these experiments.

For the second set of experiments, we depart from the scenario of [CSA19] and design a more realistic experiment in which a *presorted* order book does not exist at the beginning of the trade day. We assume the orders, both buy and sell, are entering the order book randomly, and as they enter the book, they will be matched against any existing order in the order book. In that case, the incoming order either will be matched immediately as they enter the book, otherwise, they will be inserted into their corresponding priority queue for future potential matches. We choose the price of the orders randomly from the range [1..512] according to a binomial distribution with probability-of-success 0.5, and set the volume of the orders uniformly. To demonstrate the performance of our privacy-preserving dark pool construction in a more realistic setting, we assume the order book is initially empty and as the incoming orders arrive randomly, they will be matched against the opposite side. If an order does not get matched or gets partially matched, any remaining will be added to the corresponding list, and in case it is fully fulfilled, meaning the volume becomes zero, we set its price to zero, if it is a buy order, or to infinity, if it is a sell order. The results are given in Table 7 in Appendix D. The order book is initially empty, and the number of incoming orders are $n + m = \{10, 20, 30, 40, 50, 60\}$. We measure the total computation time that it takes to process all the incoming orders, as well as the communication cost which indicates the amount of the data transferred between the parties during the computation, per party.

To illustrate the efficiency of our queue construction based on MPC, as opposed to other approaches, we conduct the aforementioned experiment on our privacy-preserving dark pool as well as on a dark pool construction based on threshold fully homomorphic encryption (tFHE) that we implemented using the GPU FHE library of [BDP20]. Note that, in the threshold FHE construction, we consider only a single server doing the computation, where it receives the orders in encrypted format from the clients and securely instantiates the priority queue data structure based on FHE. And it securely processes the incoming orders against the order book to find matches and execute trades. The FHE implementation of the dark pool framework with its priority queue is using GPU-enabled library to leverage the GPU-based parallelism, which can compute some of the operations almost over twenty-fold faster than its CPU-based TFHE implementation does. Therefore, we conduct this experiment on an AWS instance that supports GPU programming. The specs of the AWS instance is p3.2xlarge machine with 1 Tesla V100 GPUs, 16 GPU Memory and 8 vCPUs. For a better depiction on how our MPC-based solution performs as compare to FHE based construction, we run our MPC-based dark pool with the same experimental setup and on the same AWS instance, however it uses the cpu cores available on that machine and not the GPUs. For this experiment, we run all the three parties on the same machine, however they still use virtual sockets to establish their communication channel, and do not rely on the shared memory. We conduct the experiments on different number of incoming orders $n + m = \{10, 20, 30, 40\}$, and report the Total Latency, Approximate Latency per Order, as well as the Throughput for each case in Table 4.

Protocol	No. Incoming Orders	Total Latency (S)	Approx Latency per Order (S)	Throughput
FHE-based	10	64.618	6.4618	0.15475
	20	213.364	10.6682	0.09373
	30	438.242	14.60806	0.06845
	40	618.737	15.468425	0.06464
This work	10	1.39325	0.13932	7.17746
	20	1.43245	0.07162	13.96209
	30	1.45972	0.04865	20.55188
	40	1.50177	0.03754	26.63523

Table 4: Performance comparison between our privacy-preserving dark pool construction based on MPC vs. FHE. The order book is initially empty, and the number of incoming orders are $n + m = \{10, 20, 30, 40\}$.

5.3 Discussion and Conclusion

The results in the Table 3 show that our privacy-preserving dark pool has a throughput almost two orders of magnitude better than the previous scheme in dark side [CSA19]. For instance, in the case of 1 opened sell order, our scheme can process almost 136 transaction per second, whereas [CSA19] can only process less than 1 (≈ 0.6) orders per second. However, it is important to note that, the assumption in this experiment is that the order book is sorted beforehand, however our data structure handles sorting, as the orders arrives, more efficiently than that of [CSA19] with complexity $O(\log^2 N)$.

As the results show in Table 4, our MPC-based construction outperforms the FHE-based construction by two order of magnitude. For example in the case the number of incoming orders are 30, our construction can process almost 20 transaction per second, whereas FHE-based solution can only process almost 0.07 orders per second. Although, it is important to note that the implementation results given for FHE based solution in this table, do not contain the cost of the threshold key generation phase, as well as distributed decryption, as in reality they are expensive operations and as a result, it will reduce the performance of FHE-based construction even further significantly, as compared to our MPC-based construction.

ACKNOWLEDGMENTS

This paper was prepared for informational purposes by the Artificial Intelligence Research group of JPMorgan Chase & Coand its affiliates ("JP Morgan"), and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and

[8] Variance is a measure that is used to quantify the amount of variation of a set of data values from its mean in a low variance for a variable indicates that the data points tend to be close to its mean, and vice versa.

REFERENCES

[ABPV20] Gilad Asharov, Tucker Hybinette Balch, Antigoni Polychroniadou, and Manuela Veloso. PPDPs: Privacy-preserving dark pools. In *19th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2020)*, 2020. Extended abstract.

[ACC+21] A. Aly, K. Cong, D. Cozzo, M. Keller, E. Orsini, D. Rotaru, O. Scherer, P. Scholl, N.P. Smart, T. Tanguy, and T. Wood. Scale-mamba v1.2: Documentation, 1.4, 2021. https://homes.esat.kuleuven.be/~nsmart/SCALE/Documentation.pdf.

[ADEN19] Mark Abspoel, Anders Dalskov, Daniel Escudero, and Ariel Nof. An efficient passive-to-active compiler for honest-majority mpc over rings. Cryptology ePrint Archive, Report 2019/1298, 2019. https://eprint.iacr.org/2019/1298.

[AJLA+12] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 483–501, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[BBDG18] Jean-Philippe Bouchaud, Julius Bonart, Jonathan Donier, and Martin Gould. *Trades, Quotes and Prices: Financial Markets Under the Microscope.* Cambridge University Press, 2018.

[BDP20] Tucker Balch, Benjamin E. Diamond, and Antigoni Polychroniadou. Secretmatch: Inventory matching from fully homomorphic encryption. In *Proceedings of the First ACM International Conference on AI in Finance*, ICAIF '20, New York, NY, USA, 2020. Association for Computing Machinery.

[BFMZ04] Gerth Stølting Brodal, Rolf Fagerberg, Ulrich Meyer, and Norbert Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In Torben Hagerup and Jyrki Katajainen, editors, *Algorithm Theory – SWAT 2004*, pages 480–492, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10, 1988.

[BHSR19] Samiran Bag, Feng Hao, Siamak F Shahandashti, and Indranil Ghosh Ray. Seal: Sealed-bid auction without auctioneers. *IEEE Transactions on Information Forensics and Security*, 15:2042–2052, 2019.

[BRW17] Sabrina Buti, Barbara Rindi, and Ingrid M. Werner. Dark pool trading strategies, market quality and welfare. *Journal of Financial Economics*, 124(2):244–265, 2017.

[CCD87] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (abstract). In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, page 462, 1987.

[CDE+18] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. Spd $\mathbb{Z}2k$: efficient mpc mod $2k$ for dishonest majority. In *Annual International Cryptology Conference*, pages 769–798. Springer, 2018.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, third edition, 2009.

[CSA19] John Cartlidge, Nigel P. Smart, and Younes Talibi Alaoui. Mpc joins the dark side. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 148–159. Association for Computing Machinery, 2019. Full version.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229, 1987.

[GY21] Hisham S Galal and Amr M Youssef. Publicly verifiable and secrecy preserving periodic auctions. In *International Conference on Financial Cryptography and Data Security*, pages 348–363. Springer, 2021.

[Kel20] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1575–1590, 2020.

[Knu98] Donald E. Knuth. *The Art of Computer Programming*, volume 3 / Sorting and Searching. Addison–Wesley, second edition, 1998.

[MNN+18] F. Massacci, C.N. Ngo, J. Nie, D. Venturi, and J. Williams. Futuresmex: Secure, distributed futures market exchange. In *Proceedings - IEEE Symposium on Security and Privacy*, pages 335–353, 2018.

[MZ14] 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, pages 554–565. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014.

[NMKW21] Chan Nam Ngo, Fabio Massacci, Florian Kerschbaum, and Julian Williams. Practical witness-key-agreement for blockchain-based dark pools financial trading. In *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part II 25*, pages 579–598. Springer, 2021.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[Shi20] Elaine Shi. Path oblivious heap: Optimal and practical oblivious priority queue. In *IEEE S&P 2020*, 2020.

[Tof11] Tomas Toft. Secure data structures based on multi-party computation. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 291–292, 2011. Full version.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167, 1986.

A SOME FUNCTIONALITIES AND QUEUE OPERATIONS

FUNCTIONALITY A.1 ($\mathcal{F}_{\text{Match-Sell}}$-Matching Sell Functionality)**.** Input: A sell order x.

while $\mathcal{B}$.Size()! = 0 **and** $x.p \le \mathcal{B}$.Front().p **do**
  $v \leftarrow \min(x.v, \mathcal{B}.\text{Front}().v)$
  $\mathcal{T}$.Insert($x.n, \mathcal{B}.\text{Front}().n, v, \mathcal{B}.\text{Front}().p$)
  $\mathcal{B}.\text{Front}().v \mathrel{-}= v$
  if $\mathcal{B}.\text{Front}().v = 0$ **then** $\mathcal{B}$.Extract-Front()
  $x.v \mathrel{-}= v$
  if $x.v = 0$ **then** $x.p = \infty$
$\mathcal{S}$.Insert(x)
if $\mathcal{T}$.Size()! = 0 **then** Execute($\mathcal{T}$)

Algorithm 12 Q.Highest-Priority()

assign $v := []$
for $i \in \{0, \ldots, |Q.D| - 1\}$ **do**
  if $|Q.B_i| > 0$ **and**
    not ($i = |Q.D| - 1$ **and** $|Q.D_i| = 0$) **then**
      break
  v.Extend($Q.D_i$) ▷ adds all elements of D_i to end of v
  v.Extend($Q.B_i$) ▷ adds all elements of B_i to end of v ▷ at most one can be non-empty.
return v

B EXAMPLE OF QUEUE OPERATIONS

Here we give some step by step examples of main operations supported in the data-independent priority queue. Explaining these

Algorithm 13 Q.Size()

```
1: assign s := 0
2: for i ∈ {0, ... |Q.D| − 1} do
3:     s += |Q.D_i| + |Q.B_i|
   return s
```

fundamental operations is helpful in understating how the data-independent priority queue essentially works and consequently how it is being improved and used in our dark pool application.

B.1 An example for Toft's Insert Operation

Figure 4, demonstrates a step by step example for Q.Insert(x) and Q.Flush(i) operations in Toft [Tof11], and how the content of its internal data structures, buckets and buffers, changes after each operation. In step (a), the input is an array of unsorted elements 5, 3, 7, 4, 1, 2, 0, and queue is empty. The goal is to sort this array in ascending order, so assume the elements with smaller value have higher priority.

In step (b), a new element 5 arrives and invokes Q.Insert(5), from Algorithm 1. In line 1, since $|Q.B_0| = 0$, 5 will be moved to front-most buffer, $Q.B_0 \leftarrow 5$. Line 3 invokes the flush operation at level 0, $i = 0$, Q.Flush(0). In algorithm 2, in line 1, it checks if the bucket in that level i is empty and if the level i is the last level with data. Since in (b) both conditions are true, the operation proceeds to line 2, where $Q.D_0 \leftarrow Q.B_0[1 : 2^0]$. This line means any data from buffer $Q.B_0$ in positions $[1 : 2^0]$ should be "move"d to the bucket $Q.D_0$ (in the same level). Since there is only one element in buffer $Q.B_0$, it will be moved to the bucket, $Q.D_0 \leftarrow 2$.

It is important to note that, as we mentioned in 2.1, buffers and buckets are intrinsically FIFO sub-queues and as shown in Figure 1(a), they support two operations, *Enqueue*(x) and *Dequeue*(). When we "move" data elements between these sub-queues, the element is being de-queued from the source queue, and then being en-queued to the target queue. However, for the ease of demonstration, we omit these primitive operations and use $\leftarrow$ to abstract these operations to only show the movement of the data. In line 3, any remaining element from buffer $Q.B_0$ in positions $\left[2^0 + 1 : |Q.B_0|\right]$ will be moved inside the the same buffer $Q.B_0$ (i.e. the elements are shifted from positions $\left[2^i + 1 : |Q.B_i|\right]$ to $[1 : 2^i]$. In line 4, if the buffer is still over loaded ($|Q.B_i| \geq 2^i$), Flush will be re-invoked for level i. Now the insertion of first element, 5, is finished and the front-most buffer $Q.B_0$ is empty.

In step (c), new element 3 is inserted to $Q.B_0$ and Q.Flush(0) is invoked. Since $Q.D_0$ is not empty, the condition in line 1 is not satisfied and the algorithm proceeds to *else* in line 6. In line 7, we call $(Q.D_0, Q.B_0) \leftarrow \text{Merge-Split}(Q.D_0, Q.B_0)$. The elements 5 and 3 are being merged, sorted and split to 3 and 5, so that 3 moves to $Q.D_0$ and 5 moves to $Q.B_0$. In line 8, it invokes $Q.B_1 \leftarrow \text{Merge}(Q.B_0, Q.B_1)$, where it merges and sorts the buffers at level 0 and 1, and move the result to buffer $Q.B_1$. In line 9, we set the buffer $Q.B_0$ to empty. Since the buffer at level 1 is not overloaded, we do not go to line 11 and algorithm finishes here. At the end of this step (c), element 3 is in $Q.D_0$, 5 is in $Q.B_1$ and $Q.B_0$ is empty. Note that both invariants are being maintained at the end of each insertion operation. According to the first invariant, at each level i, no buffer is full, $|Q.B_i| < 2^i$, and the buckets are either full or empty, $|Q.D_i| \in 0, 2^i$. The second invariant is also held since at any level i, j, if i is at level above j, any element in buckets at level i, $Q.D_i$, has higher priority than any element of either bucket or buffer at lower level j, $Q.D_j$, $Q.B_j$.

In step (d), new element 7 is being inserted to the queue. It is placed in $Q.B_0$ and then Q.Flush(0) is invoked. Since the bucket $Q.D_0$ is not empty, we move to line 7, where $(Q.D_0, Q.B_0) \leftarrow \text{Merge-Split}(Q.D_0, Q.B_0)$, since 3 has higher priority than 7, it stays in $Q.D_0$ and 7 in $Q.B_0$. Then in line 8, $Q.B_1 \leftarrow \text{Merge}(Q.B_0, Q.B_1)$, 5 and 7 are merged and sorted and placed into buffer $Q.B_1$, and buffer $Q.B_0$ is set to empty. In line 10, since the buffer $Q.B_1$ is overloaded and $|Q.B_1| \geq 2^1$, Q.Flush(1) is invoked (for the first time). Since Flush is a recursive function and takes level i as its argument, now it is being called at level $i = 1$. In line 1 of Q.Flush(1), since $Q.D_1$ and it is the last level with data, elements 5 and 7 are being moved to the bucket at the same level, $Q.D_1 \leftarrow Q.B_1[1 : 2^1]$ and any remaining elements (which there is nothing in this case), will be moved inside the buffer $Q.B_1 \leftarrow Q.B_1\left[2^1 + 1 : |Q.B_1|\right]$. Since the buffer is not overloaded, the algorithm ends here. In this state, all elements are in buckets and buffers are empty.

In step (e), element 4 inserted into $Q.B_0$, and since the $Q.D_0$ is not empty, $(Q.D_0, Q.B_0) \leftarrow \text{Merge-Split}(Q.D_0, Q.B_0)$ is being revoked; since 3 has higher priority than 4, it stays in $Q.D_0$ and 4 is being merged with buffer $Q.B_1$, and $Q.B_0$ is set to empty. Since the buffer is not overloaded, the algorithm ends.

In step (f), new element 1 is being inserted into $Q.B_0$, and since the $Q.D_0$ is not empty, $(Q.D_0, Q.B_0) \leftarrow \text{Merge-Split}(Q.D_0, Q.B_0)$ is being revoked; this time since 1 has higher priority than 3, it moves to $Q.D_0$ and 3 placed in $Q.B_0$. Then we call, $Q.B_1 \leftarrow \text{Merge}(Q.B_0, Q.B_1)$. 4 and 3 are being merged and sorted to 3 and 4. Since the buffer $Q.B_1$ becomes overloaded we need to invoke Q.Flush(2). When $i = 1$ in Flush operation, since the bucket $Q.D_1$ is not empty, even though it is the last level, we should invoke $(Q.D_1, Q.B_1) \leftarrow \text{Merge-Split}(Q.D_1, Q.B_1)$ in line 7. By invoking Merge-Split, the element 5,7 and 3,4 are being merged and sorted and replaced in $Q.D_1, Q.B_1$ with their new order. However, since the buffer $Q.B_1$ is now overloaded, Q.Flush(2) should be invoked in line 11. When $i = 2$ in Flush operation, since the bucket $Q.D_2$ is empty, and it is the last level, the content of buffer can be moved to the bucket in the same level, as we have $Q.D_2 \leftarrow Q.B_2[1 : 2^2]$. At the end of this insertion, the elements 5,7 are in the last level buffer $Q.B_2$ and 3,4 are in the buckets at level 1, and 1 at bucket level 0.

In step (g), the new element 2 is being inserted into $Q.B_0$ and since bucket $Q.D_0$ is not empty, it is being Merge-Split with element 1, and since 1 has higher priority, it stays in front-most bucket $Q.D_0$ and 2 is being Merge with the empty buffer $Q.B_1$ and consequently placed in lower buffer $Q.B_1$.

At this point, before step (h), all the input elements are correctly placed into their corresponding locations in the queue and as can be observed, both queue invariants are being maintained. These elements can be extracted correctly any time, despite the fact that they are not all in the buckets and seem to be spread between buckets and buffers. However, for the sake of demonstration, we decided to insert one more element 0 to the queue, at step (h).

In step (h), the element 0 is being inserted into $Q.B_0$ and since bucket $Q.D_0$ is not empty, it is being Merge-Split with element 1,

since 0 has higher priority than 1, it moves to $Q.D_0$ and 1 placed in $Q.B_0$. Then we call, $Q.B_1 \leftarrow \mathsf{Merge}(Q.B_0, Q.B_1)$. 2 and 1 are being merged and sorted to 1 and 2. Since the buffer $Q.B_1$ becomes overloaded, we need to invoke Q.Flush(1). Since the bucket at level 1, $Q.D_1$, is not empty, Merge-Split is being invoked; it merges elements 1,2 in buffer and 3,4 in bucket at level 1, then sorts and splits them to 1,2 in bucket and 3,4 in buffer. Then it calls $Q.B_2 \leftarrow \mathsf{Merge}(Q.B_1, Q.B_2)$ to merge the elements 3,4 in buffer at level 1 with elements 5,7 in buffer at level 2, sort them and place them in buffer at level 2, and set the buffer at level 1 to empty. However, now the buffer at level 2 is overloaded and needs to be flushed. When we call Flush(2), since the bucket $Q.D_2$ is empty and $i = 2$ is the current last level, the content of buffer $Q.B_2$ move to bucket at the same level $Q.D_2$. As demonstrated in state (i), all the inputs elements now are placed inside buckets in the correct order, and all the variants are still maintained.

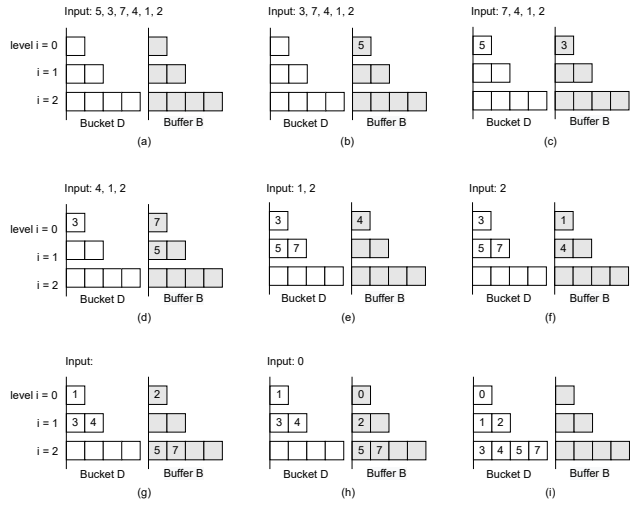

Figure 4: An example of Q.Insert(x) operation and consequently Q.Flush(i) operation in Toft [Tof11] queue, (a) shows the initial state of the queue when it is empty, every state from (b) to (g) shows the content of the queue as soon as a new element from the input is being inserted to the front-most buffer in the queue and will be processed accordingly. In (h) a new element arrives and is being added to the queue, (i) shows the content of the queue when all inserted elements are processed and are being placed in the correct location.

B.2 An example for Toft's Extract-Front Operation

In order to understand the intuition behind Extract-Front operation, we provide a simple example on Extract-Front operation of Toft. Assume we have access to the queue shown in Figure 5 (a). The Extract-Front operation in Algorithm 3, takes an argument i which indicates 2^i number of elements should be extracted from the level i and any level j where $i < j$. Note that if $i = 0$, this algorithm extracts the front-most element from the queue–which is what our modified Extract-Front operation does in our improved queue, it does not take any arguments and only removes and returns the front-most element).

For this example, imagine $i = 0$ and the goal is to extract the front-most element from the queue. In line 1, it checks whether the bucket at level 0 is full or not; if it is full, it invokes $(Q.D_0, Q.B_0) \leftarrow \mathsf{Merge\text{-}Split}(Q.D_0, Q.B_0)$, to insure that the element in bucket $Q.D_0$ still has the higher priority than the element in buffer $Q.B_0$. Then it return the $Q.D_0$ and set the $Q.D_0$ to empty. The algorithm ends here. However, if in state (b), we would like to extract the front-most element of the queue, since $Q.D_0$ is not full anymore, we check the first **else if** at line 4, where is asking if the level $i = 0$ is the last level in the current queue; since this condition fails, algorithm goes to the **else** at line 6. It calls $Q.B_1 \leftarrow \mathsf{Merge}(Q.B_0, Q.B_1)$, since $Q.B_0$ was empty, nothing moves and 2 stays in $Q.B_1$. Since the buffer $Q.B_1$ is not overloaded, algorithm proceeds to line 11 and invokes Q.Extract-Front(1) for level 1.

At first line of Q.Extract-Front(1), since $Q.D_1$ is full, it invokes $(Q.D_1, Q.B_1) \leftarrow \mathsf{Merge\text{-}Split}(Q.D_1, Q.B_1)$, which merges the elements 3,4 and 2, and sort them into 2,3,4 and split them to 2,3 in bucket $Q.D_1$ and 4 in buffer $Q.B_1$, as depicted in step (c). Then at line 3, it returns the content of $Q.D_1$ and set it to empty. We return from Q.Extract-Front(1) and back to line 11 in Q.Extract-Front(0), the returned elements, 2 and 3, are placed into temporary bucket called $Q.\tilde{D}$. In line 12, since the temporary bucket is considered full, s.t. $|Q.\tilde{D}| \geq 2^1$; hence in line 13, any element from $Q.\tilde{D}$ at position $2^0 + 1$ to 2^{0+1} (since $i = 0$), should be moved to bucket $Q.D_0$, formally $Q.D_0 \leftarrow Q.\tilde{D}[2^0 + 1 : 2^{0+1}]$. Since only element 3 has this condition and is at the position $Q.\tilde{D}[2]$, it will be moved to bucket $Q.D_0$, shown in step (d).

At line 14, any element in $Q.\tilde{D}$ that has positions above 2^{0+1} should be moved to buffer at lower level, $Q.B_1 \leftarrow Q.\tilde{D}[2^{0+1} + 1 : |Q.\tilde{D}|]$, and since no such element(s) exist(s), nothing will be moved. Finally, at line 15, the algorithms returns the content of $Q.\tilde{D}[1 : 2^0]$ which is element 2, as shown in step (e). If the temporary buffer $Q.\tilde{D}$ were not full at line 12, the algorithm would execute the line 17 and return the whole content of it. It is important to note that the queue invariants are still maintained after the execution of Extract-Front.

C EFFICIENCY ANALYSIS

The potential method, in computational complexity theory, is a method to analyze the amortized time and space complexity of a data structure, a measure of its performance over sequences of operations that eliminates the cost of infrequent but expensive operations. Usually the change in potential should be measured as positive for low-cost operations and negative for high-cost operations. There is an interesting analogy between the potential function and a bank account: if we can take our cheap operations (those whose cost is less than our bound) and put our savings from them in a bank account, using coins, then use our savings to pay for expensive operations (those whose cost is greater than our bound), and somehow guarantee that our account will never go negative, then we will have proven an amortized bound for our procedure.

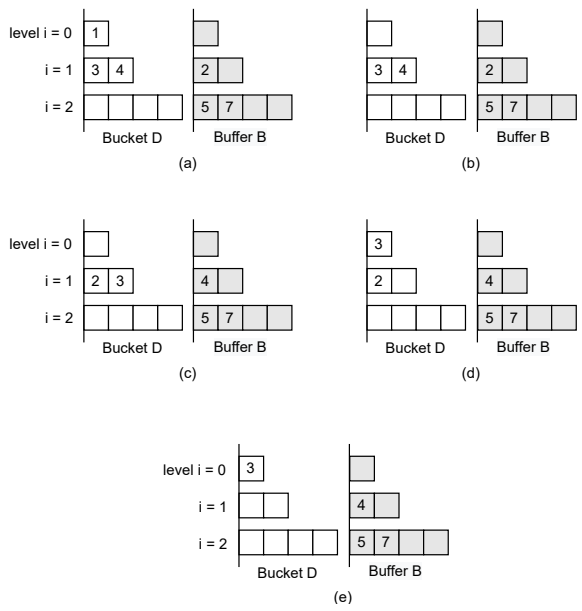

Figure 5: An example of Q.Extract-Front(i) operation in Toft [Tof11] queue, (a) shows the initial state of the queue when it has some elements; state (b) shows the content of the queue after front-most element at level $i = 0$ is extracted. steps (c) to (e) shows how to extract the next front-most element after the previous extraction, by calling Q.Extract-Front(0) again.

We use the potential function and coin method to analyze the amortized time complexity of our data-independent priority queue and its operations.

C.1 Proof of Theorem 3.1

Theorem C.1. *Consider an arbitrary sequence of intermixed queue operations, and denote by N an upper bound on the capacity attained by the queue throughout the sequence of operations. Then there exists a potential function for which Q.Insert(x) and Q.Extract-Front() have amortized costs of $O(\log^2 N)$ and $O(1)$, respectively, measured in comparator modules. Q.Front() has a worst-case cost of $O(1)$.*

Proof. We first describe our potential function, Φ, which controls the cost of both insertions and deletions; each unit of potential funds one comparator module. We set Φ equal to:

$$\sum_{i=0}^{q-1} \left(|Q.B_i| \cdot (2 \cdot \log N - i) + |Q.D_i| \cdot i + 2^i \right) \cdot \Theta(\log N).$$

(In this setting, the implicit constant in $\Theta(\log N)$ is fixed and independent of N). Put differently, each element of each buffer $Q.B_i$ has at least $(2 \cdot \log N - i) \cdot \Theta(\log N)$ coins, while each element of each bucket $Q.D_i$ has at least $i \cdot \Theta(\log N)$ coins. Finally, each non-empty level itself has $2^i \cdot \Theta(\log N)$ coins.

We first remark upon Q.Flush(i, t, e)'s movement of elements from $Q.B_i$ to $Q.D_i$, i.e. at line 4 (possible only when $i = |Q.D| - 1 = q - 1$). Because each element of the last level buffer $Q.B_i$ loses *at least* $2 \cdot \Theta(\log N)$ in potential when it is moved to $Q.D_i$ (because $(2 \cdot \log N - i) - i \geq 2$ for each level i), the algorithm may place exactly $2 \cdot \Theta(\log N)$ coins on the next level—i.e., at level $|Q.D| = q$—for each such element moved (without increasing overall potential). By the time the algorithm executes lines 6 and 7, at least 2^{q-1} elements have necessarily been moved in this way, so that level q has $2 \cdot 2^{q-1} \cdot \Theta(\log N) = 2^q \cdot \Theta(\log N)$ coins, and q can be incremented safely. An exceptional case occurs when q is incremented from 0 to 1 by the external routine Q.Insert(x); this act increases the queue's potential by $\Theta(\log N)$. On the other hand, it occurs only during insertions, which already cost $O(\log^2 N)$ in any case (as we discuss below). The insertion of each new element x into $Q.B_0$ increases the queue's potential by $2 \cdot \log N \cdot \Theta(\log N) = O(\log^2 N)$. We now argue, essentially as in [Tof11, §5.5], that each call to Q.Flush($i, t,$ Regular)—regardless of which routine calls it—can be funded by an adequate drop in the queue's potential. Indeed, this routine is *only* called when $|Q.B_i| \geq 2^i$, and moreover empties the contents of $Q.B_i$ into $Q.B_{i+1}$; by consequence, it decreases the queue's potential by least $2^i \cdot \Theta(\log N)$. On the other hand, Q.Flush($i, t,$ Regular) itself costs at most $\Theta(2^i \log 2^i)$; indeed, the calls Merge-Split($Q.D_i, Q.B_i$) and Merge($Q.B_i, Q.B_{i+1}$) involve lists whose sizes exceed 2^i by at most a constant factor.

We now analyze Q.Flush($i, t,$ Extract). The essential idea is that the routine funds its cost by moving "sufficiently many" elements from $Q.D_i$ into higher level buckets. We distinguish between *typical* and *degenerate* executions of Q.Extract-Front(). We call an execution *typical* if and only if there exists some i for which line 10 of Q.Flush($i, t,$ Extract) executes; otherwise, we call the execution *degenerate*. In a typical execution, we write i^* for the minimal such i (i.e., at which e is first flipped to Regular).

In a typical execution, Q.Extract-Front() proceeds by executing Q.Flush($i, t,$ Extract) for each i up to and including i^*. (Further executions of Q.Flush($i, t,$ Regular), if they take place—i.e., for $i > i^*$—may be funded in the manner described above). By definition of i^*, upon the termination of Q.Flush($0, 0,$ Extract), each $i < i^*$ satisfies $\sum_{j \leq i} |Q.D_j| \leq 2^i$ (in fact, the inequality is strict for positive i); on the other hand, $\sum_{j \leq i^*} |Q.D_j| \geq 2^{i^*}$. Moreover, $|Q.B_i| = 0$ for each $i \leq i^*$. By consequence, for each positive $i \leq i^*$, the input v passed into Q.Retrieve(i, v) has length $|v| \leq 2^{i-1}$, whereas Q.Retrieve(i, v) necessarily returns a vector of length exactly 2^i. It follows that, for $0 < i \leq i^*$, Q.Retrieve(i, v) single-handedly moves at least 2^{i-1} elements from $Q.D_i$ into those buffers $Q.D_j$ for $j < i$. The resulting drop in potential—of at least $2^{i-1} \cdot \Theta(\log N)$—suffices to fund the cost of Q.Flush($i, t,$ Extract), itself at most $\Theta(2^i \log 2^i)$; indeed, both Merge-Split($Q.D_i, Q.B_i$) and Merge($Q.B_i, Q.B_{i+1}$) involve lists whose sizes exceed 2^{i-1} by at most a constant factor. Q.Retrieve($0, v$) does not change the queue's potential; on the other hand, Q.Flush($0, t,$ Extract) imposes no cost. Finally, additional calls Q.Retrieve(i, v)—that is, for $i > i^*$—can only further decrease potential. We note that no work is exerted at these levels.

In a degenerate execution, upon the termination of flush operation Q.Flush($0, 0,$ Extract), each i satisfies $\sum_{j \leq i} |Q.D_j| \leq 2^i$ (again with strict inequalities for positive i). We observe that, for each i, Q.Retrieve(i, v, e) *either* returns exactly 2^i elements *or* removes the level i altogether (or both, in case $|v| = 2^i$ exactly before

line 7 begins). In the former case, the reasoning given above demonstrates that Q.Retrieve(i, v, e) sends at least 2^{i-1} elements from $Q.D_i$ into the higher buckets, and hence reduces the queue's potential by $2^{i-1} \cdot \Theta(\log N)$. In the latter case, the removal of level i reduces the queue's potential by $2^i \cdot \Theta(\log N)$. Both reductions suffice to fund the cost of Q.Flush$(i, t,$ Extract), itself at most $\Theta(2^i \log 2^i)$. In other words, reducing the cost of some queue's operations such as insertions and deletion, can cover the high cost of Flush operation. □

D EXPERIMENTAL RESULTS

Opened Sell Orders	Computation Time (S)	Variance
2	0.008200467	1.26272E-08
4	0.015145233	2.81649E-08
8	0.030181933	3.03158E-07
16	0.0578681	7.29808E-07
32	0.113870333	8.36937E-07
64	0.224520333	9.07922E-06
128	0.449528	1.79212E-05
256	0.923293	6.19776E-05
512	1.950453333	0.000315694
1024	4.006068386	0.001091776

Table 5: Performance of our Privacy-Preserving Dark pool with higher number of open sell orders. $n = m = 1024$

Protocol	Opened Sell Orders	Matching Latency (S)	Total Latency (S)
[CSA19]	0	0.7526	1.567-1.584
	1	0.7565	1.571-1.588
	2	0.7608	1.575-1.592
	3	0.7649	1.579-1.597

Table 6: The total latency (i.e. run time) of CDA method [CSA19] and its two phases, Matching and Insertion, are all in seconds (S). The insertion latency which measures the time to insert the remaining volume of incoming buy order to the buy list, and depending where on the list it will be inserted (top, middle or bottom) can vary between [0.8150 - 0.83214] seconds. The size of the book is chosen to be $n = m = 40$**.**

No. Incoming Orders	Computation Time (S)	Communication Cost (MB)
10	1.34675	1.51988
20	1.37065	1.72196
30	1.38397	1.78091
40	1.40503	2.036708
50	1.41043	2.27555
60	1.43180	2.3816225

Table 7: Performance of our Privacy-Preserving Dark pool in the honest majority three-party setting tolerating one malicious adversary, with random incoming orders. The order book is initially empty, and the number of incoming orders are $n + m = \{10, 20, 30, 40, 50, 60\}$**. Computation time is the total time it takes to process all the incoming orders, while communication cost is the transmitted data per party.**