Post-Quantum Secure Oblivious Transfers for Resource-Constrained Receivers

Abstract. We introduce two scalable OT variants: (1) Helix OT, a 1-out-of-n OT, and (2) Priority OT, a t-out-of-n OT. They provide unconditional security. Helix OT achieves a receiver-side download complexity of $O(1)$. In big data scenarios, where certain data may be more urgent or valuable, we propose Priority OT. With a receiver-side download complexity of $O(t)$, this scheme allows data to be received based on specified priorities. By prioritizing data transmission, Priority OT ensures that the most important data is received first, optimizing the use of bandwidth and processing resources. Performance evaluations indicate that Helix OT completes the transfer of 1 out of $n = 16,777,216$ messages in 9 seconds, and Priority OT handles $t = 1,048,576$ out of n selections in 30 seconds. Both outperform existing t-out-of-n OTs (when $t \geq 1$), underscoring their suitability for large-scale applications.

1 Introduction

Oblivious Transfer (OT) [48,24,53] is an important cryptographic primitive that enables a receiver to choose and learn exactly t of n messages held by a sender (where $t \geq 1$ and $n > t$). In this scenario, the sender must not be able to learn which specific messages were chosen and the receiver must not gain any information about the remaining $n - t$ messages. OT has applications in various domains, such as generic secure Multi-Party Computation [57,4,30], Private Set Intersection [22], contract signing [24], Federated Learning [56,49,55], and Zero-Knowledge proof systems [29].

In this paper, we introduce two scalable variants of OT: (1) Helix[1] OT, a 1-out-of-n OT, and (2) Priority OT, a t-out-of-n OT. Both schemes provide unconditional security, making them post-quantum secure. Additionally, they do not depend on any unconventional assumptions, such as noisy channels, trusted initialization, or the receiver's ability to store the sender's entire encrypted database. These two protocols are straightforward to understand, easy to analyze for potential vulnerabilities, and simple to implement, requiring only a single library (GMP [26]) for big integer arithmetic.

The receiver's download communication complexity in Helix OT is $O(1)$ while in Priority OT it is $O(t)$. In Priority OT, the receiver can sequentially obtain t out of n messages based on an initial preference, while maintaining the privacy of the chosen preferences. As we will discuss in Section 7.3, existing OT schemes

[1] The protocol is named "Helix" because its structure mirrors the layered complexity of a helix (shape), utilizing a Merkle tree where permutations may be applied to each level.

do not *efficiently* support ordering without imposing a download cost of at least $O(n)$ on the receiver.

Helix OT uses XOR-based secret sharing, one-time pads, a third-party helper (such as an SGX enclave within the sender's machine) that may be corrupted by a semi-honest adversary, and a new tool called a *tree-based controlled swap*, which could be of independent interest. Priority OT primarily utilizes random permutation, one-time pads, the third-party helper, and a simple tool called a *permutation map*.

We have implemented Helix OT and Priority OT, evaluated their performance, and compared them against state-of-the-art OTs. Our analysis shows that both Helix OT and Priority OT are highly scalable. For example, when n = 16,777,216 and $t = 1$, Helix OT completes in about 9 seconds (see Table 1). In comparison, for the same value of n but with t = 1,048,576, Priority OT takes around 30 seconds to complete (see Table 3). We also studied these two schemes' runtime when they are invoked up to 100,000,000 times, in the 1-out-of-2 setting. In this scenario, Helix OT and Priority OT complete in about 4.7 and 7 minutes respectively (see Table 2). Furthermore, they can be at least 411 times faster than existing efficient base OTs in the 1-out-of-2 setting (see Table 4), and 10 times faster than existing efficient t-out-of-n OTs, in the 12-out-of-16 setting (see Table 7). To the best of our knowledge, this is the first time the performance of OTs is studied for large values of n and t.

2 Preliminaries

2.1 Notations and Assumptions

We denote an empty string with ϵ. We denote a sender by S, a receiver by R, and a third-party helper by H. We consider the setting where these parties are corrupted by semi-honest (passive) adversaries. We assume parties interact with each other through a regular secure channel. U denotes a universe of messages $m_0, \ldots, m_l$. We define σ as the maximum bit size of messages in U, i.e., $\sigma = Max(|m_1|, \ldots, |m_l|)$. We define an algorithm $\texttt{Find}(\boldsymbol{v}, j) \rightarrow y$ that takes as input a vector $\boldsymbol{v}$ and a value j. If value j is in $\boldsymbol{v}$, it returns the index y of j in $\boldsymbol{v}$; otherwise, it returns ϵ. By $\mathcal{X} \equiv \mathcal{Y}$ we mean $\mathcal{X}$ and $\mathcal{Y}$ are unconditionally indistinguishable. We define $\texttt{Decompose}(e_1, e_2) \rightarrow b \in \{0,1\}^{e_2}$ as a mapping that takes integers e_1 and e_2 and decomposes e_1 into its e_2-bit binary representation. For a bit string b, by $b[i]$ we mean the i-th binary value of b, where $i \geq 0$. In this work, we require R to delineates its priorities using a vector called priority vector $\boldsymbol{p}$. A priority vector is defined as follows.

Definition 1 (Priority-Vector). *Let $\boldsymbol{m}$ be a vector of n messages and $\boldsymbol{p}$ be a vector of t indices, where $t \leq n$. Vector $\boldsymbol{p}$ is called priority vector if the elements of $\boldsymbol{p}$ are arranged such that $\boldsymbol{p}[0]$ corresponds to the index of a message in $\boldsymbol{m}$ deemed most critical, $\boldsymbol{p}[1]$ refers to the index of a message in $\boldsymbol{m}$ with the next highest level of importance, and this pattern continues in descending order of priority.*

2.2 Random Permutation

A random permutation, $\pi : S \to S$, is a bijective function chosen uniformly at random from the set of all possible permutations of the set S. This means that each permutation of the elements of S is equally likely. In practice, Fisher-Yates shuffle algorithm [37] can permute a set of m elements in time $O(m)$. Formal definitions of pseudorandom function and permutation can be found in [36].

2.3 Controlled Swap

Fredkin and Toffoli [25] initially introduced the idea of a controlled swap. It can be defined as function $\texttt{CS}(s, pair) \to pair'$ which takes two inputs: a binary value s and a pair $pair := (c_0, c_1)$. When $s = 0$, it returns the input pair $pair' := (c_0, c_1)$, i.e., it does not swap the elements. When $s = 1$, it returns $pair' := (c_1, c_0)$, i.e., it swaps the elements. If s is uniformly chosen at random, then $\texttt{CS}(.,.)$ represents a random permutation; therefore, the probability of swapping or not swapping is $\frac{1}{2}$ in this case.

2.4 Binary Tree

A binary tree is a data structure in which each node has at most two children, referred to as the left child and the right child.

The topmost node in the tree is called the root. It is the starting point for traversing the tree. Nodes that are at the lowest level of the tree and do not have any children are called leaf nodes or leaves. The height of a binary tree is the length of the longest path from the root to a leaf. In this paper, we consider a perfect binary tree, all internal nodes have two children and all leaves are at the same level. For a binary tree with n leaf nodes, the height of a binary tree is $e = \log_2(n)$. Let $pair_{_{f,h}}$ be f-th pair at h-th level. Each pair $pair_{_{f,h}}$ contains two nodes $pair_{_{f,h}} := (node_{_{2f,h}}, node_{_{2f+1,h}})$, where $1 \leq h \leq e$ and $0 \leq f \leq \frac{2^h}{2} - 1$.

In this work, for the sake of simplicity, we assume n is a power of 2 and construct a binary tree on top of a set of messages $m_0, \ldots, m_{_{n-1}}$. At the lowest level of the tree, each pair $pair_{_{f,e}}$ contains two nodes $pair_{_{f,h}} := (node_{_{2f,h}}, node_{_{2f+1,h}})$, such that $node_{_{2f,e}} = m_{_{2f}}$ and $node_{_{2f+1,e}} = m_{_{2f+1}}$.

2.5 Secret Sharing

A (threshold) secret sharing $\texttt{SS}^{(t,n)}$ scheme is a cryptographic protocol that enables a dealer to distribute a string s, known as the secret, among n parties in a way that the secret s can be recovered when at least a predefined number of shares, say t, are combined. If the number of shares in any subset is less than t, the secret remains unrecoverable and the shares divulge no information about s. This type of scheme is referred to as (n, t)-secret sharing or $\texttt{SS}^{(t,n)}$ for brevity.

In the case where $t = n$, there exists a highly efficient XOR-based secret sharing [7]. In this case, to share the secret s, the dealer first picks $n-1$ random bit strings $r_1, \ldots, r_{_{n-1}}$ of the same length as the secret. Then, it computes $r_{_n} =$

$r_1 \oplus \ldots \oplus r_n \oplus s$. It considers each $r_i \in \{r_1, \ldots, r_n\}$ as a share of the secret. To reconstruct the secret, one can easily compute $r_1 \oplus \ldots \oplus r_n$. Any subset of less than n shares reveal no information about the secret. We will use this scheme in this paper. A secret sharing scheme involves two main algorithms; namely, $\mathtt{SS}(1^\lambda, s, n, t) \rightarrow (r_1, \ldots, r_n)$: to share a secret and $\mathtt{RE}(r_1, \ldots, r_t, n, t) \rightarrow s$ to reconstruct the secret.

2.6 Security Model

In this paper, we rely on the simulation-based model of secure multi-party computation [27] to define and prove the proposed protocols. Below, we restate the formal security definition within this model.

Two-party Computation. A two-party protocol Γ problem is captured by specifying a random process that maps pairs of inputs to pairs of outputs, one for each party. Such process is referred to as a functionality denoted by $f : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^*$, where $f := (f_1, f_2)$. For every input pair (x, y), the output pair is a random variable $(f_1(x, y), f_2(x, y))$, such that the party with input x wishes to obtain $f_1(x, y)$ while the party with input y wishes to receive $f_2(x, y)$. In the setting where f is asymmetric and only one party (say the first one) receives the result, f is defined as $f := (f_1(x, y), \epsilon)$.

Security in the Presence of Passive Adversaries. In the passive adversarial model, the party corrupted by such an adversary correctly follows the protocol specification. Nonetheless, the adversary obtains the internal state of the corrupted party, including the transcript of all the messages received, and tries to use this to learn information that should remain private. Loosely speaking, a protocol is secure if whatever can be computed by a party in the protocol can be computed using its input and output only. In the simulation-based model, it is required that a party's view in a protocol's execution can be simulated given only its input and output.

This implies that the parties learn nothing from the protocol's execution. More formally, party i's view (during the execution of Γ) on input pair (x, y) is denoted by $\mathsf{View}_i^\Gamma(x, y)$ and equals $(w, r_i, m_1^i, \ldots, m_t^i)$, where $w \in \{x, y\}$ is the input of i^{th} party, r_i is the outcome of this party's internal random coin tosses, and m_j^i represents the j^{th} message this party receives. The output of the i^{th} party during the execution of Γ on (x, y) is denoted by $\mathsf{Output}_i^\Gamma(x, y)$ and can be generated from its own view of the execution.

Definition 2. *Let f be the deterministic functionality defined above. Protocol Γ securely computes f in the presence of a passive adversary if there exist polynomial-time algorithms $(\mathsf{Sim}^\Gamma{}_1, \mathsf{Sim}_2^\Gamma)$ such that:*

$$\{\mathsf{Sim}_1^\Gamma(x, f_1(x, y))\}_{x,y} \equiv \{\mathsf{View}_1^\Gamma(x, y)\}_{x,y}$$

$$\{\mathsf{Sim}_2^\Gamma(y, f_2(x, y))\}_{x,y} \equiv \{\mathsf{View}_2^\Gamma(x, y)\}_{x,y}$$

3 Related Work

The traditional 1-out-of-2 OT ($\mathcal{OT}^2_{1-2}$) is a protocol that involves two parties, a sender S and a receiver R. S has a pair of input messages (m_0, m_1) and R has an index s. The aim of $\mathcal{OT}^2_{1-2}$ is to allow R to obtain m_s, without revealing anything about s to S, and without allowing R to learn anything about m_{1-s}. The traditional $\mathcal{OT}^2_{1-2}$ functionality is defined as $\mathcal{F}_{\mathcal{OT}^2_{1-2}} : ((m_0, m_1), s) \rightarrow (\epsilon, m_s)$.

The notion of 1-out-of-2 OT was initially proposed by Rabin [48] which consequently was generalized by Even *et al.* [24]. Since then, numerous variants of OT have been proposed. For instance, (i) 1-out-of-2 OT: which enables the receiver to select 1 entry out of 2 entries held by S, e.g., see those schemes proposed in [6,1], (ii) t-out-of-n OT: which allows R to pick t entries out of n entries held by S, where $1 \leq t \leq n$; examples include the OTs proposed in [41,51,40] designed for the cases where $t = 1$ and the OTs introduced in [15,35,14] suitable for the scenarios where $t \geq 1$, (iii) OT extension, e.g., in [32,31,44,4]: that supports efficient executions of OT (that mainly relies on symmetric-key operations), in the case OT needs to be invoked many times, (iv) distributed OT, e.g., in [42,59,17]: that allows the database to be distributed among m servers/senders, and (v) correlated (or random) OTs, which are variants of OTs that considers specific scenarios where the inputs of the senders are correlated random values, rather than a set of messages (e.g., a private database) in the generic OT. The correlated OTs are often more efficient than generic OTs due to the certain structures that the input messages have. The schemes proposed in [18,45,12,11] are examples of correlated and random OTs.

In the remainder of this section, we discuss those variants of *generic* OTs that are closest to our proposed OTs.

3.1 Efficient t-out-of-n OT

To generalize the notion of 1-out-of-2 OT, t-out-of-n OTs have been proposed. These OTs are suitable for scenarios where $n > 2$ and $t \geq 1$.

Naor and Pinkas proposed two variants of OT in [41] one suitable when $t = 1$ and another one when $t \geq 1$. They rely on a pseudorandom function and any standard 1-out-of-2 OT. The former variant (when $t = 1$) involves $\log(n)$ invocations of a 1-out-of-2 OT, and the receiver obtains n ciphertexts from the sender. The latter, that supports the case when $t \geq 1$, requires $2 \cdot t \cdot \log(n)$ invocations of a 1-out-of-2 OT and operates under the constraint that $t \ll n$. In this variant the receiver obtains $t \cdot n$ ciphertexts from the sender.

Tzeng [51] proposed a 1-out-of-n OT, that relies on the Decisional Diffie-Hellman (DDH) assumption and involves public key operations. In this scheme, the receiver obtains n ciphertexts from the sender. Another t-out-of-n OT was proposed in [34], which relies on the Discrete logarithm problem (DLP), involves modular exponentiation linear with n and require the receiver to obtain messages linear with $n + t$. Wei *et al.* [52] proposed server-aided t-out-of-n OT, using the DDH assumption and involving modular exponentiation linear with t and n. In this scheme, the receiver obtains a response whose size is linear with n.

The efficient OT extensions, e.g., proposed in [32,31,44,4] have initially been designed for 1-out-of-2 OT setting, however, they can be called multiple times to meet the requirements of t-out-of-n OTs. Nevertheless, this approach will require the sender to obtain $t \cdot n$ messages and it will include a constant number of public operations to invoke base OT's to set up the initial system parameters.

To date, the fastest 1-out-of-n semi-honest and malicious secure OTs are the OT extensions proposed in [38] and [46] respectively, with a caveat. They have been designed to work efficiently when the input secret messages are *very short*, $log(n)$. For instance, the size of each input message is 4 when $n = 16$. Both schemes use a base OT (that often relies on a computationally hard problem) and a random oracle, while the latter also uses a pseudorandom generator. In both schemes, the receiver obtains a response of size $O(m \cdot (\lambda + n \cdot l))$, where m is the number of OT invocations, λ is a security parameter, and l is an input message's bit size. These schemes do not directly offer t-out-of-n OT. To achieve t-out-of-n OTs, one can simply set $m = t$ and invoke either of them t times.

We will propose the first t-out-of-n OTs that is unconditionally secure and efficiently work for arbitrary length inputs, e.g., 128 bits.

3.2 Unconditionally or Post-Quantum Secure OT

There have been efforts to design (both-sided) unconditionally secure OTs. Some schemes, such as those proposed in [42,10], rely on multiple servers/senders that maintain an identical copy of the database. Other ones, like the OTs introduced in [19,20,33], are based on a specific network structure, i.e., a noisy channel, to achieve unconditionally secure OT. There is also a scheme in [50] that achieves unconditionally secure OT using a fully trusted initializer.

The t-out-of-n OTs proposed in [54,16] achieves one-sided unconditional security, when only one of the parties is corrupt by unbounded adversary. These scheme still rely on computational hard problems, such as the DLP or DDH.

Moreover, there exist OT schemes developed to maintain security in the presence of adversaries equipped with quantum computers. Examples include those proposed in [9,8,47,39,23,5]. However, these schemes are not unconditionally secure. Instead, they rely on various assumptions and problems (such as short integer solution, learning with errors, multivariate quadratic, decoding random linear codes, or computing isogenies between supersingular elliptic curves) as well as primitives (such as AES, pseudorandom generator, lattice-based Chameleon hash function, multivariate quadratic cryptography, McEliece cryptosystem, or supersingular isogeny Diffie-Hellman key exchange) that are deemed valid and secure in the era of quantum computing based on current knowledge and assessment. Their security could be compromised if any of the underlying assumptions or problems are proven to be solvable efficiently by future advancements in quantum algorithms or other unforeseen developments.

Hence, there exists no (efficient) unconditionally secure OT that does not use noisy channels, multi-server, and fully trusted initializer.

3.3 OT with Constant Response Size

Researchers have proposed several OTs, e.g., those proposed in [13,28,58,16], that enable a receiver to obtain a constant-size response to its query. To achieve this level of communication efficiency, these protocols require the receiver to locally store the encryption of the *entire database*, in the initialization phase. During the transfer phase, the sender assists the receiver with locally decrypting the message that the receiver is interested in. The main limitation of these protocols is that a thin client with limited available storage space cannot use them, as it cannot locally store the encryption of the entire database.

4 Definition

Our OT schemes fall under the category of 3-party OT, initially defined in [21]. However, the original definition primarily addresses 1-out-of-2 OTs, lacks a download efficiency property[2], and does not offer the concept of ordering. In this section, we present a generalized definition for 3-party OT, to capture t-out-of-n OT scenarios. This definition will also include the download efficiency property, and the concept of ordering, which we refer to as *order-respecting*.

A 3-party t-out-of-n OT ($\mathcal{OT}^3_{t-n}$) involves a sender S, a receiver R, and a helper H. We assume each party can be corrupted by a passive non-colluding adversary. The functionality $\mathcal{F}_{\mathcal{OT}^3_{t-n}}$ that $\mathcal{OT}^3_{t-n}$ will compute is different from that of conventional OT in the sense that now an additional party H is introduced. We define the functionality as $\mathcal{F}_{\mathcal{OT}^3_{t-n}} : \big((m_0,\ldots,m_{n-1}),\epsilon,\boldsymbol{p}\big) \rightarrow (\epsilon,(n,t),\{m_j\}_{\forall j\in\boldsymbol{p}})$, which takes n messages from S, no input from H, and a vector $\boldsymbol{p}$ of t integers from R. It returns nothing to S, the total number of messages n and the total number of retrieved messaged t to H, and t messages to R.

Definition 3 (Security). *Let $\mathcal{F}_{\mathcal{OT}^3_{t-n}}$ be the OT functionality defined above. We assert that protocol Γ securely realizes $\mathcal{F}_{\mathcal{OT}^3_{t-n}}$ in the presence of passive adversaries, if for every adversary $\mathcal{A}$ in the real model, there is a simulator* $\mathtt{Sim}$ *in the ideal model, where:*

$$\begin{aligned}\Big\{\mathtt{Sim}^{\Gamma}_{S}\big((m_0,\ldots,m_{n-1}),\epsilon\big)\Big\}_{m_0,\ldots,m_{n-1},\boldsymbol{p}} \equiv\\ \Big\{\mathtt{View}^{\Gamma}_{S}\big((m_0,\ldots,m_{n-1}),\epsilon,\boldsymbol{p}\big)\Big\}_{m_0,\ldots,m_{n-1},\boldsymbol{p}}\end{aligned} \tag{1}$$

$$\Big\{\mathtt{Sim}^{\Gamma}_{H}(\epsilon,(n,t))\Big\}_{m_0,\ldots,m_{n-1},\boldsymbol{p}} \equiv \Big\{\mathtt{View}^{\Gamma}_{H}\big((m_0,\ldots,m_{n-1}),\epsilon,\boldsymbol{p}\big)\Big\}_{m_0,\ldots,m_{n-1},\boldsymbol{p}} \tag{2}$$

$$\begin{aligned}\Big\{\mathtt{Sim}^{\Gamma}_{R}\Big(\boldsymbol{p},\mathcal{F}_{\mathcal{OT}^3_{t-n}}\big((m_0,\ldots,m_{n-1}),\epsilon,\boldsymbol{p}\big)\Big)\Big\}_{m_0,\ldots,m_{n-1},\boldsymbol{p}} \equiv\\ \Big\{\mathtt{View}^{\Gamma}_{R}\big((m_0,\ldots,m_{n-1}),\epsilon,\boldsymbol{p}\big)\Big\}_{m_0,\ldots,m_{n-1},\boldsymbol{p}}\end{aligned} \tag{3}$$

[2] The download efficiency property was only informally discussed in [21]

Definition 4 (Download Efficiency). *An $\mathcal{OT}^3_{t-n}$ scheme is considered download efficient if the total number of messages k that receiver R obtains and the bit-size of each message that R receives are constant $O(1)$ concerning the total number of message n and are linear $O(t)$ with respect to t. Thus, the total complexity of R's received messages is $O(t)$. More formally, $\exists k \in \mathbb{N}$, such that the total complexity of the messages obtained by R is:*

$$O\big(t \cdot k \cdot Max(|m_0|, \ldots, |m_{n-1}|)\big) = O(t)$$

Informally, the order-respecting property ensures that the messages received by R are ordered according to a predefined priority of the indices, specified in $\boldsymbol{p}$.

Definition 5 (Order-Respecting). *Let $\boldsymbol{m} = [m_0, \ldots, m_{n-1}]$ be a vector of n messages, and $\boldsymbol{p}$ be the corresponding priority vector of size t, as defined in Definition 1. An $\mathcal{OT}^3_{t-n}$ is order-respecting if a receiver R obtains t messages in the order: $m_{\boldsymbol{p}[0]}, m_{\boldsymbol{p}[1]}, \ldots, m_{\boldsymbol{p}[t-1]}$.*

In the above definitions, in the case of 1-out-of-n OT (where $t = 1$), we simplify replace vector $\boldsymbol{p}$ with single value $indx$.

5 Helix OT: An Efficient 1-out-of-n OT

This section presents Helix OT, which relies on a novel combination of XOR-based secret sharing, one-time pads, a helper, and a tool called Tree-Based Controlled Swap (`TBCS`). We initially describe `TBCS` and then present this OT.

5.1 Tree-Based Controlled Swap

Tree-Based Controlled Swap (`TBCS`) is a new variant of traditional controlled swap [25] capable of handling the swap of more than two messages, utilizing a binary tree structure. In $\texttt{TBCS}(b, \boldsymbol{m}) \rightarrow \boldsymbol{w}$, we construct a binary tree on top of messages $\boldsymbol{m} = m_0, \ldots, m_{n-1}$ that we want to swap. This tree, along with the messages, is then permuted according to a predefined set of rules and an input bit-string b, where $|b| = \log_2(n)$. This bit-string b is a bit representation of one of the messages' index $indx$, i.e., a target message's index. Each bit in b determines the permutation of pairs of nodes at each corresponding level of the tree. Specifically, the least significant bit of b determines the permutation at the leaf node level, the next bit governs the level above the leaf nodes, and this pattern continues up the tree.

At a high level, $\texttt{TBCS}(.,.)$ works as follows. Beginning with the leaf nodes of the binary tree, we apply the controlled swap to each pair of nodes (i.e., messages) that share the same parent node. A swap occurs between these two nodes if the corresponding bit in the bit-string b is 1.

Next, we move up one level and apply the controlled swap to each pair of internal nodes sharing the same parent. If the related bit in b is 1, we swap these nodes along with their respective sub-trees. In this case, the order of the

descendant nodes remains unchanged, and the entire sub-trees are swapped. This process is repeated until we reach the root of the tree. By the end of this procedure, the leaf nodes of the tree will have been permuted according to the specific rules and the bits of the bit-string b. Figure 1 presents two examples of applying TBCS() to a vector of eight messages when $indx = 3$ and $indx = 7$.

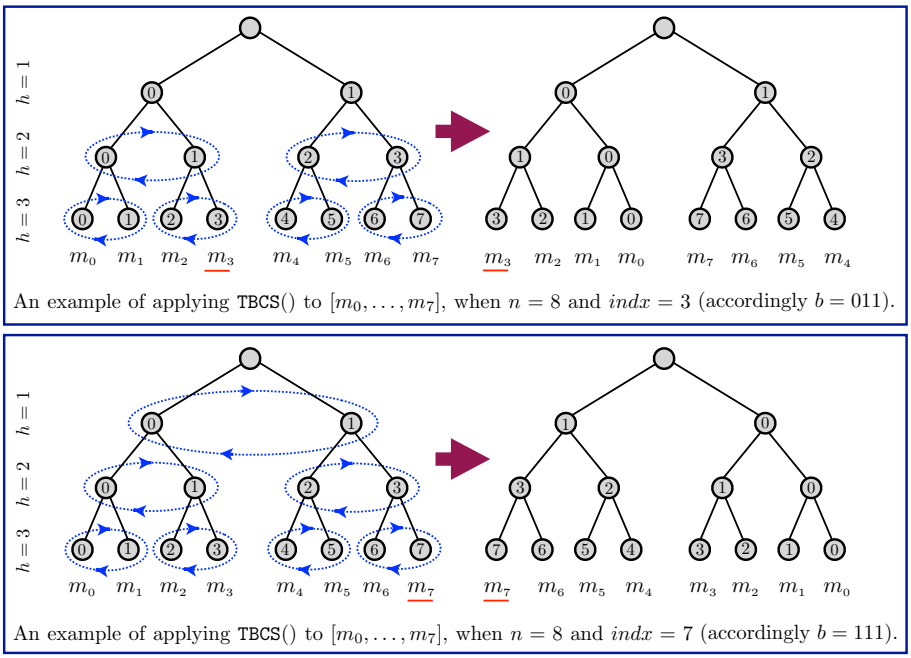


Fig. 1: Applying TBCS () to a vector of 8 messages, where the target index *indx* is 3 (top) and 7 (bottom). As shown, the first element in the output of TBCS () is the message at the target index.

Figure 2 presents TBCS in detail. TBCS(.,.) can be considered as a generalization of the original controlled swap CS(.,.) discussed in Section 2.3. TBCS(.,.) offers an interesting feature. Let b be a binary representation of an index *indx* of a (target) message in $\boldsymbol{m}$. Then, after applying TBCS(.,.) to $\boldsymbol{m}$ using b, the first element of the output of TBCS(.,.) is always m_{indx}. We formally state this feature below.

Claim 1. *Let* $\boldsymbol{m} = [m_0, \ldots, m_{n-1}]$ *be a vector of messages and* b *be a binary representation of an index indx of one of these messages, i.e.,* $0 \leq indx \leq n-1$ *and* $\texttt{Decompose}(indx, log_2(n)) \rightarrow b$. *Then, after swapping* $m_0, \ldots, m_{n-1}$ *using* TBCS(.,.) *and* b, *the first element of the resulting vector is* m_{indx}. *Formally,* $\boldsymbol{w}[0] = m_{indx}$, *where* $\texttt{TBCS}(b, \boldsymbol{m}) \rightarrow \boldsymbol{w}$.

Proof. The binary index b determines how the target node, m_{indx}, must be moved. We initially consider the leaf node level. The target node, after applying the swap rules at this level, always becomes the first child of its parent node, denoted as $node'$, for the following reasons. At the leaf node level, if the target node is originally the second child (or the right-hand side node) of its parent node $node'$, its corresponding bit (the least significant bit in b) is always 1, i.e., $b[log_2(n) - 1] = 1$. Because its decimal index is an odd value. According to the swap rule, the target node is swapped with its sibling node, resulting in the target node becoming the first child (or the left-hand side node) of $node'$. Otherwise, if the target node is already the first child of $node'$, it does not move at that level. Thus, after applying the swap rules at this level, the target node is always at the first position in the sub-tree with root node $node'$.

We move one level up the tree. If the parent node $node'$ of the target node, is the second child of its own parent node $node''$, then the corresponding bit in b is always 1, i.e., $b[log_2(n) - 2] = 1$. In this scenario, $node'$ and its sub-tree is always swapped with the sibling of $node'$ and its sub-tree. Otherwise, if the bit is 0, no swap takes place at this level. Hence, after applying the swap rules, the target node occupies the first position in the sub-tree with root node $node''$.

If we continue applying the same principle, we reach the two sub-tree root nodes rt_1 and rt_2, which are the left and right children of the entire tree's root node respectively. If the target node is in the sub-tree with root node rt_2, then the corresponding bit in b is always 1. As a result, rt_2 and its sub-tee is swapped with rt_1 and its sub-tee. Otherwise, if the target node is in the sub-tree with root node rt_1, no swap occurs at this level. Before the swap, the target node was already the first leaf node in the corresponding sub-tree. However, after applying the swap rules, the target node becomes the first node in the entire tree; specifically, it holds that $\boldsymbol{w}[0] = m_{indx}$. □

Note that $\texttt{TBCS}(.,.)$ offers an interesting feature in a distributed setting. It allows two non-colluding parties to collaboratively permute messages such that the leaf node initially at a target position always appears as the first leaf node in the final permuted tree if both parties follow the permutation rules. When combined with secret sharing, this method ensures secure and oblivious filtering of messages, enabling only one message to be sent to the recipient without disclosing this message's original index to the parties performing the permutation. This point becomes clearer when we explain Helix OT in detail.

5.2 An Overview of Helix OT

The primary idea behind the design of this OT is to require S to encrypt the messages it possesses, permute them using $\texttt{TBCS}(.,.)$, and send the result to H. Subsequently, H permutes the messages using $\texttt{TBCS}(.,.)$ and sends only the first node, containing a message, in the permuted tree to R. Upon receiving the encrypted message, R decrypts it to extract the desired plaintext message.

We proceed to provide more detail. Given the private index $indx$ that R possesses, it represents $indx$ into its binary representation b and splits each bit

$\underline{\texttt{TBCS}(b, \boldsymbol{m}) \rightarrow \boldsymbol{w}}$

- Inputs: (1) a vector of n messages $\boldsymbol{m} = [m_0, \ldots, m_{n-1}]$, and (2) a bit-string b of length $e = \log_2(n)$.
- Output: a vector $\boldsymbol{w}$ of all messages in $\boldsymbol{m}$, permuted.

1. construct a binary tree T on messages $m_0, \ldots, m_{n-1}$.
2. in each level (starting from the lowest one), apply controlled swap to each pair of nodes $pair_{f,h} := (node_{2f,h}, node_{2f+1,h})$ that share the same parents, using the bit-string b. Specifically, update the tree T as follows. $\forall h, e \geq h \geq 1, \quad \forall f, 0 \leq f \leq \frac{2^h}{2} - 1$:
   - if $b[h-1] = 1$ and $pair_{f,h}$ is an internal node: swap each node, along with its descendants, with the other node, i.e., swap subtrees with root nodes $node_{2f,h}$ and $node_{2f+1,h}$. In this setting, the order of the descendants of these two nodes does not change.
   - if $b[h-1] = 1$ and $pair_{f,h}$ is a leaf node: swap each node, i.e., swap $node_{2f,h}$ and $node_{2f+1,h}$.
   - if $b[h-1] = 0$: do not swap the nodes.

   Let T' be the resulting permuted tree, and $\boldsymbol{w}$ be the leaf nodes of T'.
3. return $\boldsymbol{w}$.

Fig. 2: Tree-Based Controlled Swap (TBCS).

of b into two shares, using XOR-based secret sharing. This yields two bit-strings q_S and q_H. Moreover, R generates some random values $\boldsymbol{v} = [v_0, \ldots, v_{n-1}]$. It sends $(q_S, \boldsymbol{v})$ to S and q_H to H. Sender S proceeds with encrypting the messages it holds, using the elements of $\boldsymbol{v}$. It permutes the encrypted messages using $\texttt{TBCS}(.,.)$ and the bits of q_S. It sends the permuted encrypted messages to H, which permutes them again using $\texttt{TBCS}(.,.)$ and the bits of q_H. Subsequently, H sends only the message corresponding to the first node (in the leaf node level) of the tree to R and discards the rest of the tree. R decrypts the message using $indx$ and an element of $\boldsymbol{r}$, yielding its desired plaintext message.

5.3 Detailed Description of Helix OT

Below, we present the protocol in more detail.

1. *R-side Setup:* $\texttt{Setup}(1^\lambda, indx) \rightarrow \boldsymbol{r}$
   This algorithm is executed every time R wants to send a query.
   (a) picks n random values $(r_0, \ldots, r_{n-1}) \xleftarrow{\$} \{0,1\}^\sigma$. Let vector $\boldsymbol{r}$ be defined as $\boldsymbol{r} = [r_0, \ldots, r_{n-1}]$.
   (b) sends $\boldsymbol{r}$ to S.

(c) locally stores $r_{indx} \in \boldsymbol{r}$ and discards the rest of the elements in $\boldsymbol{r}$.

2. *R-side Query Generation:* $\texttt{GenQuery}(1^\lambda, indx) \rightarrow q = (q_S, q_H)$
   (a) decompose $indx$ into its binary representation:
   $$\texttt{Decompose}(indx, e) \rightarrow b$$
   where $e = \log_2(n)$.
   (b) splits every bit of the private bit-string b into two shares as follows:
   $$\forall j, 0 \leq j \leq e-1: \quad \texttt{SS}(1^\lambda, b[j], 2, 2) \rightarrow (s_{S,j}, s_{H,j})$$
   (c) sets bit strings q_S and q_H as follows:
   $$q_S \leftarrow s_{S,0} || \ldots || s_{S,e-1}$$
   $$q_H \leftarrow s_{H,0} || \ldots || s_{H,e-1}$$
   (d) sends q_S to S and q_H to H.
3. *S-side Response Generation:* $\texttt{GenRes}(m_0, \ldots, m_{n-1}, \boldsymbol{r}, q_S) \rightarrow res_H$
   (a) encrypts each message as follows.
   $$\forall g, 0 \leq g \leq n-1: \quad m'_g \leftarrow m_g \oplus r_g$$
   (b) constructs a vector $\boldsymbol{z}$ of the encrypted messages as follows:
   $$\boldsymbol{z} = \left[(m'_0, m'_1), (m'_2, m'_3), \ldots, (m'_{n-2}, m'_{n-1})\right]$$
   (c) permutes the elements of vector $\boldsymbol{z}$ using the tree-based controlled swap $\texttt{TBCS}(.,.)$ and bit string q_S, as:
   $$\texttt{TBCS}(q_S, \boldsymbol{z}) \rightarrow \boldsymbol{w}$$
   (d) sets $res_H \leftarrow \boldsymbol{w}$ and sends res_H to H.
4. *H-side Oblivious Filtering:* $\texttt{OblFilter}(res_H, q_H) \rightarrow res_R$
   (a) permutes the elements of $\boldsymbol{w}$ (in res_H) using $\texttt{TBCS}(.,.)$ and bit string q_H:
   $$\texttt{TBCS}(q_H, \boldsymbol{w}) \rightarrow \boldsymbol{w}'$$
   (b) sets res_R always to the first element, say e, of the first pair in $\boldsymbol{w}'$ and discards the rest of the elements in $\boldsymbol{w}'$. It sends res_R to R.
5. *R-side Message Extraction:* $\texttt{Retreive}(res_R, \boldsymbol{r}, indx) \rightarrow m_{indx}$
   (a) retrieves the final related message m_{indx} by decrypting res_R:
   $$m_{indx} = res_R \oplus r_{indx}$$
   (b) returns m_{indx}.

Theorem 1. *Let $\mathcal{F}_{\mathcal{OT}^3_{t-n}}$ be the functionality defined in Section 4. Then, Helix OT securely computes $\mathcal{F}_{\mathcal{OT}^3_{t-n}}$ in the case where $t = 1$, in the presence of semi-honest adversaries, with regard to Definition 3.*

5.4 Proof of Theorem 1

In this section, we prove the security of Helix, i.e, Theorem 1.

Proof. We will prove the security of the protocol when each party is corrupt.
Corrupt Sender S. The view of S during the real execution of Helix includes: $\texttt{View}_S^{Helix}\big((m_0,\ldots,m_{n-1}),\epsilon,indx\big) = \{r_S, \boldsymbol{r}, q_S\}$, where r_S is the outcome of the internal random coin of S, $\boldsymbol{r} = [r_0,\ldots,r_{n-1}]$ is a vector of random value sent to S by R, $q_S = s_{S,0}||\ldots||s_{S,e-1}$ is a query that R sends to S, and each $s_{S,j}$ is a share, i.e., an output of $\texttt{SS}()$. We construct an ideal-model simulator $\texttt{Sim}_S^{Helix}$ which receives $m_0,\ldots,m_{n-1}$ and the security parameter σ from S and outputs a view which has identical distribution to the view of S in the real model. Figure 3 shows how $\texttt{Sim}_S^{Helix}$ works.

$\texttt{Sim}_S^{Helix}(m_0,\ldots,m_{n-1},\sigma) \rightarrow \texttt{View}_{\texttt{Sim}_S}^{Helix}$

1. Initiates an empty view $\texttt{View}_{\texttt{Sim}_S}^{Helix}$ and appends a uniformly random coin r'_S to the view.
2. Selects n random values $(d_0,\ldots,d_{n-1}) \xleftarrow{\$} \{0,1\}^\sigma$. Let $\boldsymbol{d}$ contain these random values.
3. Selects $e = \log_2(n)$ random bits $s'_{S,0},\ldots,s'_{S,e-1}$, where $s'_{H,j} \xleftarrow{\$} \{0,1\}$.
4. Sets q'_S to $s'_{S,0}||\ldots||s'_{S,e-1}$.
5. Appends $\boldsymbol{d}$ and q'_S to $\texttt{View}_{\texttt{Sim}_S}^{Helix}$ and outputs $\texttt{View}_{\texttt{Sim}_S}^{Helix} = \{r'_S, \boldsymbol{d}, q'_S\}$.

Fig. 3: Simulator $\texttt{Sim}_S^{Helix}$ for sender S in Helix.

We will argue that the views of an adversary in the real and ideal models have identical distributions. Random coins r_S in the real model and r'_S in the ideal model have identical distributions. Furthermore, vectors $\boldsymbol{r}$ in the real model and $\boldsymbol{d}$ in the ideal model have identical distributions, as each element of $\boldsymbol{r}$ and $\boldsymbol{d}$ is picked uniformly at random from a domain of size σ by R and $\texttt{Sim}_S^{Helix}$ respectively. The bit strings q_S in the real model and q'_S in the ideal model have identical distributions as well, for the following reason. Each bit $s_{S,j} \in q_S$ is an output of the XOR-based secret sharing scheme $\texttt{SS}()$, while each bit $s'_{S,j} \in q'_S$ is selected uniformly at random. Due to the security of $\texttt{SS}()$, the output of $\texttt{SS}()$ has an identical distribution to a bit selected uniformly at random. Thus, each $s_{S,j}$ has an identical distribution to $s'_{S,j}$.
Corrupt Helper H. The view of H during the real execution of Helix includes: $\texttt{View}_H^{Helix}\big((m_0,\ldots,m_{n-1}),\epsilon,indx\big) = \{r_H, q_H, res_H\}$, where r_H is the outcome of the internal random coin of H, $q_H = s_{H,0}||\ldots||s_{H,e-1}$ is a query that R sends to H, and each $s_{H,j}$ is a an output of $\texttt{SS}()$, and $res_H = \boldsymbol{w}$ is the output of $\texttt{TBCS}()$ which is sent to H by S as a response.

Next, we construct an ideal-model simulator $\texttt{Sim}_H^{Helix}$, given the real-model view of H. The simulator $\texttt{Sim}_H^{Helix}$ receives the total number of messages n and the security parameter σ. It outputs a view that has an identical distribution to the view of H in the real model. Figure 4 shows how $\texttt{Sim}_H^{Helix}$ works.

$\underline{\texttt{Sim}_H^{Helix}(n, \sigma) \rightarrow \texttt{View}_{\texttt{Sim}_H}^{Helix}}$

1. Initiates an empty view $\texttt{View}_{\texttt{Sim}_H}^{Helix}$ and appends a uniformly random coin r'_H to the view.
2. Selects $e = \log_2(n)$ bits $s'_{H,0}, \ldots, s'_{H,e-1}$, where each $s'_{H,j} \xleftarrow{\$} \{0,1\}$.
3. Sets q'_H to $s'_{H,0}||\ldots||s'_{H,e-1}$.
4. Constructs a vector of n random values: $\boldsymbol{b} = [a_0, \ldots, a_{n-1}]$, where $a_j \xleftarrow{\$} \{0,1\}^\sigma$.
5. Randomly permutes $\boldsymbol{b}$ as: $\pi(\boldsymbol{b}) \rightarrow \boldsymbol{e}$.
6. Sets res'_H to $\boldsymbol{e}$.
7. Appends q'_H and res'_H to $\texttt{View}_{\texttt{Sim}_H}^{Helix}$ and outputs $\texttt{View}_{\texttt{Sim}_H}^{Helix} = \{r'_S, q'_H, res'_H\}$.

Fig. 4: Simulator $\texttt{Sim}_H^{Helix}$ for helper H in Helix.

Now, we explain why the views of an adversary in the real and ideal models have identical distributions. In the real model, random coin r_H and in the ideal model random coin r'_H have identical distributions. Furthermore, the bit strings q_H in the real model and q'_H in the ideal model have identical distributions, for the same reason provided above for indistinguishably of q_S and q'_S.

In the real model, $\boldsymbol{w} \in res_H$ is a vector of encrypted elements, where each element is encrypted using a fresh one-time pad. Because of the security of one-time pads, each element in $\boldsymbol{w}$ has an identical distribution to an element, of the same size, selected uniformly at random. In the ideal model, $\boldsymbol{e} \in res'_H$ is a vector of random elements.

In the real model, the elements of $\boldsymbol{w}$ have been swapped using $\texttt{TBCS}(q_S, .)$. As previously discussed, due to the security of $\texttt{SS}()$, each bit of q_S can be considered as a uniformly random value. Hence, each controlled swap at any level of the tree occurs with a probability $\frac{1}{2}$. At the lowest level, the elements of each pair of leaf nodes are swapped with a probability of $\frac{1}{2}$. As we move up the tree, the sub-trees are swapped as whole units, but the probability of any specific element being swapped at each level remains $\frac{1}{2}$. Each level of the tree contributes to the overall permutation independently. Given that there are $\log_2(n)$ levels, the movement of any specific element through each level is equally probable.

For a specific element to end up in a certain position, it must pass through all the controlled swaps to reach that position. Each level contributes a swap decision independently, leading to a uniform distribution of the elements across

the final positions. Therefore, for a vector $\boldsymbol{w}$ of length n, the probability that an element moves to a certain position is $\frac{1}{n}$.

In the ideal model, $\boldsymbol{e} \in res'_{H}$ has been randomly shuffled. As a result, the probability that an element falls in a certain position in $\boldsymbol{e}$ is $\frac{1}{n}$. Hence, res_{H} and res'_{H} have identical distributions.

Corrupt Receiver R**.** The view of R during the real execution of Helix includes: $\texttt{View}_{R}^{Helix}\big((m_0, \ldots, m_{n-1}), \epsilon, indx\big) = \{r_{R}, indx, res_{R}, m_{indx}\}$, where r_{R} is the outcome of the internal random coin of R, $indx$ is the secret index of R, res_{R} is a single encrypted message that S sends to R, and m_{indx} is the output of the protocol which is the desirable message that R interested in. We will construct an ideal-model simulator $\texttt{Sim}_{R}^{Helix}$, using the real-model view of R. This simulator receives n, the input of R, which is private index $indx$, the output m_{indx}, and the security parameter σ. It outputs a view that has an identical distribution to the view of R in the real model. Figure 8 demonstrates how $\texttt{Sim}_{R}^{Helix}$ operates.

$\texttt{Sim}_{R}^{Helix}(n, indx, m_{indx}, \sigma) \rightarrow \texttt{View}_{\texttt{Sim}_{R}}^{Helix}$

1. Initiates an empty view $\texttt{View}_{\texttt{Sim}_{R}}^{Helix}$ and appends a uniformly random coin r'_{R} to the view.
2. Selects a random value $r_{indx} \xleftarrow{\$} \{0,1\}^{\sigma}$ using r'_{R}.
3. Encrypts m_{indx} using r_{indx} as follows: $res'_{R} = m_{indx} \oplus r_{indx}$.
4. Appends $indx$, res'_{R}, and m_{indx} to $\texttt{View}_{\texttt{Sim}_{R}}^{Helix}$ and returns $\texttt{View}_{\texttt{Sim}_{R}}^{Helix} = \{r'_{R}, indx, res'_{R}, m_{indx}\}$.

Fig. 5: Simulator $\texttt{Sim}_{S}^{Helix}$ for receiver R in Helix.

Now, we explain why the two views are identical. As before, in the real model, random coin r_{R} and in the ideal model random coin r'_{R} have identical distributions. Moreover, values $indx$ and m_{indx} are identical in both models. Also, res_{R} in the real model and res'_{R} in the ideal model have identical distributions because both have been encrypted using a fresh one-time pad, selected uniformly at random and both ciphertexts are decrypted to m_{indx}. □

6 Priority OT: An Efficient Construction of $\mathcal{OT}^{3}_{t-n}$

This section presents Priority OT, a fast t-out-of-n OT, which allows a receiver to obtain its t preferred messages in the order that it initially specifies.

6.1 An Overview

Priority OT primarily relies on random permutation, one-time pads, a helper, and a tool called a *permutation map*. A permutation map is a vector indicating

the new position of each element of a vector $\boldsymbol{v}$ of n elements after $\boldsymbol{v}$ is randomly permuted. In our protocol, the permutation map allows a receiver R to fetch t messages from sender S and helper H without disclosing the original indices of these t messages to them. At a high level, Priority OT operates as follows. Receiver R possesses a list $\boldsymbol{p}$ containing t indices of n messages held by S. The list $\boldsymbol{p}$ is organized according to R's priority. For instance, if $\boldsymbol{p} = [4, 0, 1]$, with $t = 3$ and $n = 5$, then $\boldsymbol{p}[0]$ holds the highest priority, while $\boldsymbol{p}[2]$ holds the lowest priority. Initially, R sends n random values to S. These values will be used by S to encrypt its outgoing messages. Additionally, R computes a permutation map for a vector of n elements. R asks S to encrypt and then randomly permute the n plaintext messages it holds, according to the permutation map. Consequently, S sends the result to H. Moreover, utilizing the permutation map, R instructs H to (i) retrieve t elements from the messages sent by S, and (ii) transmit each element to R in a specific order. Upon receiving each message from H, R decrypts it to obtain one of its prioritized messages.

Informally, S does not learn anything, because R does not reveal to it, its preferred indices. Moreover, H does not learn anything due to its lack of knowledge regarding (a) the original indices (or order) of the permuted messages and (b) the secret values used for encrypting the messages. One might consider replacing the random permutation with a pseudorandom permutation [36] to achieve the same goal. However, using a pseudorandom permutation does not allow us to achieve unconditionally secure OT.

6.2 Detailed Description of Priority OT

1. *R-side Setup:* $\texttt{Setup}(1^\lambda) \rightarrow \boldsymbol{r}$
   This algorithm is executed every time R wants to send a query.
   (a) picks n random values $(r_0, \ldots, r_{n-1}) \xleftarrow{\$} \{0,1\}^\sigma$. Let vector $\boldsymbol{r}$ be defined as $\boldsymbol{r} = [r_0, \ldots, r_{n-1}]$. These elements will be used as a one-time pad by S to encrypt each message that S holds.
   (b) sends $\boldsymbol{r}$ to S.
2. *R-side Query Generation:* $\texttt{GenQuery}(1^\lambda, \boldsymbol{p}) \rightarrow q := (q_S, q_H)$
   (a) determines to which position, each index in a vector $\boldsymbol{v}$ of size n is moved, if $\boldsymbol{v}$ is randomly permuted once. To do that, it takes the following steps.
      i. initiates a vector $\boldsymbol{v}$, such that its i-th element is set to i:
      $$\forall i, 0 \leq i \leq n-1 : \quad \boldsymbol{v}[i] \leftarrow i$$
      ii. randomly permutes $\boldsymbol{v}$:
      $$\pi(\boldsymbol{v}) \rightarrow \boldsymbol{w}$$
      Vector $\boldsymbol{w}$ can be considered as a permutation map which determines the position of each element $\boldsymbol{v}[i]$ after this element in $\boldsymbol{v}$ is permuted.
   (b) finds the index of each element of its priority vector $\boldsymbol{p}$ in $\boldsymbol{w}$.
   To do that, it initiates an empty vector $\boldsymbol{y}$ of size t and then takes the following steps.

$$\forall j, 0 \leq j \leq t-1: \quad \texttt{Find}(\boldsymbol{w}, \boldsymbol{p}[j]) \rightarrow y_j, \quad \boldsymbol{y}[j] \leftarrow y_j$$

Recall that the original priority vector $\boldsymbol{p}$ contains the priority-based ordered indices of R's t preferred elements in $[1, \ldots, n]$, while $\boldsymbol{y}$ determines the position of these indices in $\boldsymbol{p}$ after they are permuted according to the permutation map $\boldsymbol{w}$. Moreover, note that vector $\boldsymbol{y}$ still maintains the order of R's t preferred indices. For instance, $\boldsymbol{y}[0]$-th element in $\boldsymbol{w}$ is the index of the highest priority message while $\boldsymbol{y}[t-1]$-th element in $\boldsymbol{w}$ is that of the lowest priority message.

(c) sets $q_S \leftarrow \boldsymbol{w}$ and $q_H \leftarrow \boldsymbol{y}$. It sends q_S to S and q_H to H.

3. *S-side Response Generation:* $\texttt{GenRes}(m_0, \ldots, m_{n-1}, \boldsymbol{r}, q_S) \rightarrow res_H$

(a) encrypts each message using the elements of $\boldsymbol{r}$ and then appends the result to a vector $\boldsymbol{z}$ initially empty:

$$\forall i, 0 \leq i \leq n-1: \quad m'_i \leftarrow m_i \oplus r_i, \quad \boldsymbol{z}[i] \leftarrow m'_i$$

(b) permutes vector $\boldsymbol{z}$ according the permutation map $\boldsymbol{w} \in q_S$. To do that, it initiates an empty vector $\boldsymbol{x}$ of size n. It finds the position of each value i in the permuted vector $\boldsymbol{w}$, let y_i denote that position. It inserts i-th element from $\boldsymbol{z}$ into y_i-th position in $\boldsymbol{x}$. Specifically,

$$\forall i, 0 \leq i \leq n-1: \quad \texttt{Find}(\boldsymbol{w}, i) \rightarrow y_i, \quad \boldsymbol{x}[y_i] \leftarrow \boldsymbol{z}[i]$$

(c) sets $res_H \leftarrow \boldsymbol{x}$ and sends res_H to H.

4. *H-side Oblivious Filtering:* $\texttt{OblFilter}(res_H, q_H) \rightarrow res_R$

(a) uses elements of $\boldsymbol{y} \in q_H$ (which are priority-ordered indices of R's preferences) to retrieve R's preferred encrypted messages in the permuted vector $\boldsymbol{x} \in res_H$ and append them to an empty vector $\boldsymbol{u}$. Specifically, it takes the following steps.

$$\forall j, 0 \leq j \leq t-1: \quad \boldsymbol{u}[j] \leftarrow \boldsymbol{x}\Big[\boldsymbol{y}[j]\Big]$$

(b) sends each element of $\boldsymbol{u}$ in streaming fashion to R, based on their level of priority, starting from the highest one. Specifically, for every j (where $0 \leq j \leq t-1$), it sets $res_{R,j}$ to $(\boldsymbol{u}[j], j)$ and sends $res_{R,j}$ to R.

5. *R-side Message Extraction:* $\texttt{Retrieve}(res_{R,j}, \boldsymbol{r}, \boldsymbol{p}) \rightarrow m_p$

This algorithm is called each time R receives an encrypted element from H.

- retrieves the related message m_p with priority j, by decrypting first element u of pair $res_{R,j} := (u, j)$ as:

$$m_p = u \oplus r_p$$

where $p = \boldsymbol{p}[j]$.

Theorem 2. *Let $\mathcal{F}_{\mathcal{OT}^3_{t-n}}$ be the functionality defined in Section 4. Then, Priority OT securely computes $\mathcal{F}_{\mathcal{OT}^3_{t-n}}$ in the presence of semi-honest adversaries, regarding Definition 3.*

Theorem 3. *Priority OT satisfies download efficiency and order-respecting, with regard to Definitions 4 and 5 respectively.*

We prove Theorems 2 and 3 in Section 6.3 and Appendix A respectively.

6.3 Proof of Security

In this section, we prove the security of Priority OT, i.e., Theorem 2.

Proof. We will analyze the security of the protocol when each party is compromised by an adversary.
Corrupt Sender S**.** The view of S during the real execution of the protocol includes: $\texttt{View}_S^{Priority}\big((m_0,\ldots,m_{n-1}),\epsilon,\boldsymbol{p}\big) = \{r_S,\boldsymbol{r},q_S\}$, where r_S is the outcome of the internal random coin of S, used to generate its random values, $\boldsymbol{r}=[r_0,\ldots,r_{n-1}]$ is a vector of random value sent to S by R, and $q_S=\boldsymbol{w}$ is a query that R sends to S. Given the real-model view of S, we can construct an ideal-model simulator $\texttt{Sim}_S^{Priority}$ which receives $m_0,\ldots,m_{n-1}$ and the security parameter σ from S and outputs a view which has identical distribution to the view of S in the real model. Figure 6 demonstrates how $\texttt{Sim}_S^{Priority}$ works.

$\texttt{Sim}_S^{Priority}(m_0,\ldots,m_{n-1},\sigma)\rightarrow \texttt{View}_{\texttt{Sim}_S}^{Priority}$

1. Initiates an empty view $\texttt{View}_{\texttt{Sim}_S}^{Priority}$ and appends a uniformly random coin r'_S to the view.
2. Selects n random values $(d_0,\ldots,d_{n-1})\xleftarrow{\$}\{0,1\}^\sigma$. Let $\boldsymbol{d}$ contain these random values.
3. Constructs a permutation map $\boldsymbol{e}$ as follows.
   (a) Initiates a vector $\boldsymbol{x}$, such that its i-th element is set to i:
   $$\forall i, 0\leq i\leq n-1: \quad \boldsymbol{x}[i]\leftarrow i$$
   (b) Randomly permutes $\boldsymbol{x}$ as: $\pi(\boldsymbol{x})\rightarrow\boldsymbol{e}$.
4. Sets q'_S to $\boldsymbol{e}$.
5. Appends $\boldsymbol{d}$ and q'_S to $\texttt{View}_{\texttt{Sim}_S}^{Priority}$ and outputs $\texttt{View}_{\texttt{Sim}_S}^{Priority} = \{r'_S,\boldsymbol{d},q'_S\}$.

Fig. 6: Simulator $\texttt{Sim}_S^{Priority}$ for sender S in Priority OT.

We discuss that the views of an adversary in the real and ideal models have identical distributions. Random coin r_S in the real model and r'_S in the ideal model have identical distributions as the one in the real model has been selected uniformly at random by a semi-honest adversary while the one in the ideal model has been chosen uniformly at random by $\texttt{Sim}_S^{Priority}$. The same argument holds for $\boldsymbol{r}$ in the real model and $\boldsymbol{d}$ in the ideal model. The reason is that each element of $\boldsymbol{r}$ is picked uniformly at random by a R from a domain of size σ while each element of $\boldsymbol{d}$ is selected uniformly at random by $\texttt{Sim}_S^{Priority}$ from a domain of the same size. Moreover, vectors $\boldsymbol{w}\in q_S$ (in the real model) and $\boldsymbol{e}\in q'_S$ (in the ideal model) have identical distributions as they both have been permuted randomly.

Corrupt Helper H. The view of H during the real execution of Priority OT includes: $\texttt{View}_H^{Priority}\big((m_0,\ldots,m_{n-1}),\epsilon,\boldsymbol{p}\big) = \{r_H, q_H, res_H\}$, where r_H is the outcome of the internal random coin of H, $q_H = \boldsymbol{y}$ is a query that R sends to H, and $res_H = \boldsymbol{x}$ is a response that S sends to H.

Next, we construct an ideal-model simulator $\texttt{Sim}_H^{Priority}$, given the real-model view of H. $\texttt{Sim}_H^{Priority}$ receives n, t, and the security parameter σ. It outputs a view that has an identical distribution to the view of H in the real model. Figure 7 shows how $\texttt{Sim}_H^{Priority}$ operates.

$\texttt{Sim}_H^{Priority}(n,t,\sigma) \rightarrow \texttt{View}_{\texttt{Sim}_H}^{Priority}$

1. Initiates an empty view $\texttt{View}_{\texttt{Sim}_H}^{Priority}$ and appends a uniformly random coin r'_H to the view.
2. Selects n values $[b_0,\ldots,b_{n-1}]$, where each value is chosen uniformly at random, i.e., $b_i \xleftarrow{\$} \{0,1\}^\sigma$. Let $\boldsymbol{b}$ contain these values.
3. Randomly permutes $\boldsymbol{b}$ as: $\pi(\boldsymbol{b}) \rightarrow \boldsymbol{e}$.
4. Selects uniformly at random t indices of the elements in $\boldsymbol{e}$. Let $\boldsymbol{c}$ contain these indices.
5. Sets q'_H to $\boldsymbol{c}$ and res'_H to $\boldsymbol{e}$.
6. Appends q'_H and res'_H to $\texttt{View}_{\texttt{Sim}_H}^{Priority}$ and outputs $\texttt{View}_{\texttt{Sim}_H}^{Priority} = \{r'_S, q'_H, res'_H\}$.

Fig. 7: Simulator $\texttt{Sim}_H^{Priority}$ for helper H in Priority OT.

We proceed to explain why the views of an adversary in the real and ideal models are identical. In the real model, random coin r_H and in the ideal model random coin r'_H have identical distributions. Because r_H has been honestly selected uniformly at random by a semi-honest adversary while r'_H has been chosen uniformly at random by $\texttt{Sim}_H$. In the real model, $\boldsymbol{x} \in res_H$ is a permuted vector of elements, where each element is encrypted using a fresh one-time pad. Due to the security of one-time pads, each element in $\boldsymbol{x}$ has an identical distribution to an element (of the same size) selected uniformly at random. In the real model, the probability that an adversary can correctly guess a correct index of an element in $\boldsymbol{x}$ is $\frac{1}{n}$, because the vector $\boldsymbol{x}$ has been randomly permuted; therefore, the probability that an element ends up in a specific position in the final permutation is $\frac{1}{n}$.

In the ideal model, $\boldsymbol{e} \in res'_H$ is a vector of random elements that have been randomly permuted. In this case, the probability that a specific element ends up in a specific position is $\frac{1}{n}$. Hence, $\boldsymbol{x}$ and $\boldsymbol{e}$ (and accordingly res_H and res'_H) have identical distributions. In the real model, each element of vector $\boldsymbol{y}$ referees to a position in a permuted vector $\boldsymbol{x} \in res_H$, while in the ideal model, each element of vector $\boldsymbol{c}$ is an index picked uniformly at random from one of the elements'

indices in the permuted vector $\boldsymbol{e} \in res'_H$. Both vectors $\boldsymbol{x}$ and $\boldsymbol{e}$ contain random elements and the probability that an element is moved to a specific location is $\frac{1}{n}$, according to the above discussion. Hence, in the real model, from H's view, the probability of receiving any element in $\boldsymbol{y}$ is the same as the probability of receiving any other index in $\boldsymbol{x}$. In the ideal model, since each element of $\boldsymbol{c}$ has been picked uniformly at random, the probability of receiving any element in $\boldsymbol{c}$ is the same as the probability of receiving any other index in $\boldsymbol{e}$. Therefore, $\boldsymbol{y}$ and $\boldsymbol{c}$ (and accordingly q_H and q'_H) have identical distributions.

Corrupt Receiver R**.** The view of R within the real execution of Priority OT includes: $\texttt{View}_R^{Priority}\big((m_0,\ldots,m_{n-1}),\epsilon,\boldsymbol{p}\big) = \{r_R,\boldsymbol{p},\{res_{R,j}\}_{\forall j,0\le j\le t-1},\{m_l\}_{\forall l\in\boldsymbol{p}}\}$, where r_R is the outcome of the internal random coin of R, $\boldsymbol{p}$ is the priority vector of R, $\{res_{R,j}\}_{\forall j,0\le j\le t-1}$ is a vector of encrypted messages that S sends to R, and $\{m_l\}_{\forall l\in\boldsymbol{p}}$ is the output of the protocol which includes the desirable messages that R is interested in. We will construct an ideal-model simulator $\texttt{Sim}_R^{Priority}$, using the real-model view of R. This simulator receives n, the input $\boldsymbol{p}$ of R, the output $\{m_l\}_{\forall l\in\boldsymbol{p}}$, and the security parameter σ. It outputs a view that has an identical distribution to the view of R in the real model. Figure 8 demonstrates how $\texttt{Sim}_R^{Priority}$ works.

$\texttt{Sim}_R^{Priority}(n,\boldsymbol{p},\{m_l\}_{\forall l\in\boldsymbol{p}},\sigma)\rightarrow\texttt{View}_{\texttt{Sim}_R}^{Priority}$

1. Initiates an empty view $\texttt{View}_{\texttt{Sim}_R}^{Priority}$ and appends a uniformly random coin r'_R to the view.
2. Selects n random values $\boldsymbol{b} = [b_0,\ldots,b_{n-1}]$, using r'_R, where $b_j \xleftarrow{\$} \{0,1\}^\sigma$.
3. Encrypts each m_l in $\{m_l\}_{\forall l\in\boldsymbol{p}}$ using an element of $\boldsymbol{b}$ as follows, $\forall j, 0 \le j \le t-1 : res'_{R,j} = m_p \oplus b_p$, where $p = \boldsymbol{p}[j]$.
4. Appends $\boldsymbol{p}$, $\{res'_{R,j}\}_{\forall j,0\le j\le t-1}$, and $\{m_l\}_{\forall l\in\boldsymbol{p}}$ to $\texttt{View}_{\texttt{Sim}_R}^{Priority}$ and returns $\texttt{View}_{\texttt{Sim}_R}^{Priority} = \big\{r'_R,\boldsymbol{p},\{res'_{R,j}\}_{\forall j,0\le j\le t-1},\{m_l\}_{\forall l\in\boldsymbol{p}}\big\}$.

Fig. 8: Simulator $\texttt{Sim}_S^{Priority}$ for receiver R in Priority OT.

Now, we are ready to explain why the two views are identical. In the real model, random coin r_R and in the ideal model random coin r'_R have identical distributions. Furthermore, in the real and ideal models, vector $\boldsymbol{p}$ is identical, the same applies to the output set $\{m_l\}_{\forall l\in\boldsymbol{p}}$. Moreover, $\{res_{R,j}\}_{\forall j,0\le j\le t-1}$ in the real model and $\{res'_{R,j}\}_{\forall j,0\le j\le t-1}$ in the ideal model have identical distributions because the elements of both sets have been encrypted using fresh one-time pads and they are decrypted to identical values. □

7 Evaluation

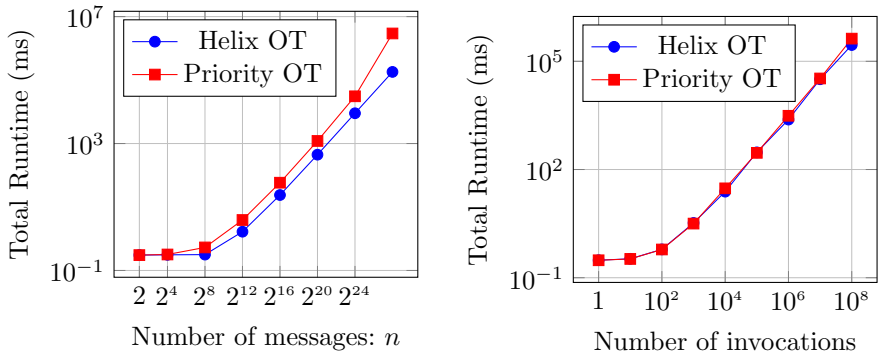
We implemented Helix OT and Priority OT in C++ and evaluated their concrete runtime. The source code for the implementation is publicly available in [2,3]. We used a MacBook Pro with an Apple M3 Pro CPU and 36 GB of RAM for the experiment. No parallelization or other optimizations were applied. The experiment was repeated an average of 20 times. All charts in this paper are on a logarithmic scale. We utilized the GMP library [26] for big integer arithmetic. The security parameter of all schemes studied in this section is 128 bits. Accordingly, in Helix OT and Priority OT, we set the size of each message that the sender holds to 128 bits. We analyze the runtime of various phases of Helix OT and Priority OT across different parameters and schemes. Specifically, we:

- analyze the runtime of Helix OT and Priority OT for different number of messages n held by the sender, ranging from 2 to approximately 268 million. Table 1 provides a summary of the results.
- analyze the runtime of Helix OT and Priority OT for different invocation frequencies, ranging from 1 to 100 million times. Table 2 outlines the results.
- analyze the runtime of Priority OT for various values of t (from 2 to about 16 million) and n (from 2 to 268 million). Table 3 shows the results.
- compare the runtime of Helix OT and Priority OT with base OTs proposed in [4,43,1] for 1-out-of-2 setting. Table 4 presents a summary of the results.
- compare the runtime of Helix OT and the most efficient 1-out-of-n OT proposed in [38], for different number of OT invocations (from 125,000 to 1,250,000), when $n = 16$. Table 5 outlines the outcomes.
- compare the runtime of Helix OT and the efficient OT in [38], for different values of n (from 8 to 128) when the number of OT invocations is 50,000 and $t = 1$. Table 6 shows the results.
- compare the runtime of Priority OT and the OT in [38], for different number of OT invocations (from 125,000 to 1,250,000) and values of t (from 2 to 12) when $n = 16$. Table 7 summarizes the results.
- compare the features of Helix OT and Priority OT with several state-of-the-art OTs. Table 8 shows the results.

We obtained the code from [1] and ran it in our machine to estimate its runtime and conduct the analysis. For the runtime of STD–OT in [4], STD–OT in [43], RO–OT in [43], we derived the reported figures from [4], specifically from Table 3, where the GMP library was employed. For the runtime of [38], we derived the figures from [46]. All experiments for the aforementioned schemes, including ours, were conducted on laptops. We acknowledge that variations in experimental environments (e.g., hardware and network delay) can influence the runtime. However, these factors typically impact the runtime by no more than a factor of 2 or 3. As we will demonstrate, in certain cases, the performance improvements achieved by our schemes far exceed these variations.

7.1 Runtime of Helix OT and Priority OT

As Table 1 indicates, when the number of messages increases from 2 to $2^8 = 268{,}435{,}456$, the runtime of Helix OT increases from 0.3 to 180,806 milliseconds (ms) or 3 minutes, while the runtime of Priority grows from 0.3 to 2,943,085 ms or 49 minutes. As Figure 9a shows, for small values of n (up to around 2^8), the runtime of both protocols is similar. As n increases, the runtime of Priority OT grows faster than Helix OT. This is especially apparent for $n \geq 2^{12}$. At the largest values of n (2^{28}), Priority OT's runtime is higher than that of Helix OT. Therefore, as the number of messages scales up, Helix OT becomes more efficient, demonstrating that it is more suitable than Priority OT for the 1-out-of-n setting, i.e., when $t = 1$. According to Table 1, in both schemes, (a) Phase 5 (receiver-side message retrieval) imposes very low computation cost across all values of n and (b) Phase 1 (the receiver-side setup) and Phase 2 (query generation) impose low cost (at most 391 ms combined) when n is in the range $[2, 2^{20}]$. This attribute indicates that these protocols can be effectively employed by resource-constrained devices without significantly depleting their battery when n is within the above range.



(a) Runtime vs Number of Messages (b) Runtime vs Number of Invocations

Fig. 9: Comparison of total runtime for Helix OT and Priority OT across (a) different numbers of messages and (b) different numbers of invocations.

As Table 2 demonstrates, when the number of OT invocations increases from 1 to 100,000,000, the total runtime of Helix OT increases from 0.3 to 282,454 ms (or 4 minutes) while the runtime of Priority OT increases from 0.3 to 424,795 ms (or 7 minutes). As Figure 9b indicates, both protocols demonstrate a gradual increase in runtime for smaller numbers of invocations (1 to 10^2). In this case, both schemes' runtime is almost identical. However, the growth accelerates as the number of invocations reaches 10^4 and beyond. Priority OT shows a faster increase in runtime compared to Helix OT. According to Table 2, in both schemes, the receiver-related phases (Phases 1, 2, and 5) impose low cost (at most about 12 ms), when the number of invocations is between 1 and 10,000.

Table 1: The table presents the runtime (in ms) of Helix OT and Priority OT for different n, across various phases, when $t = 1$.

Protocol	Phase	Number of messages: n							
		2	2^4	2^8	2^{12}	2^{16}	2^{20}	2^{24}	2^{28}
Helix OT	1	0.299	0.3	0.301	0.496	3.073	44.856	761.959	16343.5
	2	0.005	0.006	0.007	0.009	0.011	0.017	0.034	0.054
	3	0.001	0.003	0.005	0.698	11.854	230.745	4827.7	97006.5
	4	0.001	0.002	0.003	0.463	8.908	169.811	3411.47	67456.1
	5	0.0003	0.0003	0.0004	0.0004	0.0005	0.002	0.003	0.036
	Total	0.306	0.311	0.316	1.666	23.846	445.431	9001.166	180806.19
Priority OT	1	0.3	0.3	0.31	0.459	2.865	42.884	789.428	19475.6
	2	0.002	0.007	0.085	1.322	21.066	349.881	8388.79	171850
	3	0.002	0.011	0.136	2.107	34.365	812.245	21650.4	2751760
	4	0.0008	0.0008	0.0008	0.0009	0.001	0.004	0.006	0.027
	5	0.0004	0.0004	0.0004	0.0004	0.0004	0.001	0.002	0.086
	Total	0.305	0.319	0.532	3.889	58.297	1205.015	30828.626	2943085.713

Table 2: The table presents the runtime (in ms) of Helix OT and Priority OT for different number of invocations, when $t = 1$ and $n = 2$.

Protocol	Phase	Number of invocations								
		1	10	10^2	10^3	10^4	10^5	10^6	10^7	10^8
Helix OT	1	0.299	0.303	0.347	0.585	2.97	26.688	279.483	2871.42	30000
	2	0.005	0.013	0.101	0.80457	8.04	80.1101	804.913	8275.36	89317.2
	3	0.001	0.011	0.098	1.123	8.34	112.96	842.173	12036.8	96365.1
	4	0.001	0.007	0.056	0.705	4.13	70.989	411.538	7583.49	50769.8
	5	0.0003	0.0013	0.009	0.081	0.858	8.315	83.383	999.895	16002.5
	Total	0.306	0.335	0.611	3.29857	24.338	299.062	2421.49	31766.965	282454.6
Priority OT	1	0.3	0.302	0.328	0.567	2.96	26.714	281.152	2805.7	32327.6
	2	0.002	0.008	0.073	0.667	7.01	66.784	723.319	10394.3	121818
	3	0.002	0.018	0.175	1.661	17.232	168.108	1787.01	17280.7	231383
	4	0.0008	0.003	0.024	0.221	2.277	22.444	235.671	2293.88	28963.8
	5	0.0004	0.001	0.007	0.057	0.587	5.791	59.161	616.539	10302.9
	Total	0.305	0.332	0.607	3.173	30.066	289.841	3086.313	33391.119	424795.3

According to Table 3, Priority OT scales well even for large values of n and t. For instance, when $n = 2^{24} = 16{,}777{,}216$, and $t = 2^{20} = 1{,}048{,}576$, it will take about 31,905 ms (or half a minute) to complete. For significantly larger parameters, the runtime may increase substantially, requiring more powerful devices to handle the increased computation. For instance, when $t = 2^{24}$ and $n = 2^{28}$, in total, it will take about 52 minutes to terminate.

7.2 Runtime Comparison

As Table 4 indicates, Helix OT and Priority OT have runtime of 0.62 ms and 0.7 ms respectively, indicating that they are much faster than the base OTs in [4,43]. For Helix OT, the enhancement rates compared to STD-OTs are very high

Table 3: The table presents the runtime (in ms) of Priority OT for different values of n and t.

Protocol	t	Number of messages: n						
		2^4	2^8	2^{12}	2^{16}	2^{20}	2^{24}	2^{28}
Priority OT	2	0.3257	0.5747	4.2262	64.2984	1270.43	30892.3	2961580
	2^4	–	0.5777	4.2354	65.131	1280.34	30902	2966150
	2^8	–	–	4.2925	65.5668	1309.79	30919.6	3139240
	2^{12}	–	–	–	66.157	1328.63	30938.1	3141950
	2^{16}	–	–	–	–	1356.14	30964.7	3142570
	2^{20}		–	–	–	–	31905.5	3143390
	2^{24}	–	–	–	–	–	–	3149780

(1,962 and 2,711 times faster), indicating substantial efficiency gains. However, it has a moderate improvement over RO-OT (464 times faster).

Table 4: The table compares the runtime (in ms) of Helix OT and Priority OT with the following "base" OTs: standard OT (STD–OT) in [4], STD–OT in [43], the random oracle OT (RO–OT) in [43], and Supersonic OT in [1] for 1-out-of-2 setting. The runtime is based on 128 invocations of each scheme. The text in blue shows the improvement rate attained by our schemes, while the text in red shows the overhead rate of our schemes.

Protocol	Runtime (in ms)	Rate	
		Helix OT	Priority OT
STD–OT in [4]	1217	1962.9	1738.5
STD–OT in [43]	1681	2711.2	2401.4
RO–OT in [43]	288	464.5	411.4
Supersonic OT in [1]	0.36	0.58	0.51
Helix OT	0.62	1	0.88
Priority OT	0.7	1.12	1

Priority OT follows a similar trend but with slightly lower enhancement rates (1,738 and 2,401 for STD-OTs and 411 for RO-OT). When compared to Supersonic OT [1], both Helix OT and Priority OT have overhead rates (0.58 and 0.51), suggesting that they are slower than 1-out-of-2 Supersonic OT but still quite efficient. As Figure 10 (in Appendix B) demonstrates, STD-OT in [43] has the highest runtime whereas Supersonic OT [1] has the lowest. As we discussed in Section 3.1, the 1-out-of-n OT proposed in [38] is the most efficient 1-out-of-n OT secure against semi-honest adversaries. On the other hand, as Tables 1 and 2 show, Helix OT is faster than Priority OT, when $t = 1$. Thus, we compared Helix OT with the efficient OT in [38] in Table 5. For all invocation counts, Helix OT demonstrates a lower runtime than the OT from [38]. Overall, Helix OT offers a factor of 2.1 improvement over the OT in [38].

Table 5: The table compares the runtime (in ms) of Helix OT and the most efficient 1-out-of-16 OT proposed in [38], for different number of invocations. For the OT in [38] the message size is only 4 bits. The text in blue shows the average improvement rate achieved by our scheme.

Protocol	Number of invocations				Average
	1.25×10^5	2.5×10^5	5×10^5	1.25×10^6	Rate
OT in [38]	2160	4230	8500	21680	2.1
Helix OT	982.45	1962.83	3945.84	10113.07	1

Furthermore, we compared the runtime of Helix OT and the OT in [38], for different values of n when $t = 1$. As Table 6 illustrates, Helix OT shows lower runtime, approximately half the time of OT in [38] for most values of n. However, when $n = 2^7$, the runtime of Helix OT is approximately 1.3 times longer than that of OT in [38]. Thus, on average Helix OT offers about a factor of 2 improvement over the OT in [38]. We note that the OT in [38] operates efficiently on a much smaller message size that Helix OT, such as 7 bits compared to 128 bits.

Table 6: The table compares the runtime (in ms) of Helix OT and the efficient OT proposed in [38], for different values of n, when the number of invocations is 5×10^4. For the OT in [38] the message size is at most 7 bits, i.e., $\log_2(n)$ bits. The text in blue shows the average improvement rate attained by our scheme.

Protocol	Number of messages: n					Average
	2^3	2^4	2^5	2^6	2^7	Rate
OT in [38]	700	960	1220	1330	1500	2
Helix OT	299.48	415.8	690.92	1119.26	2033.84	1

We aimed to compare the runtime of our (t-out-of-n) Priority OT with the state-of-the-art t-out-of-n OT. However, to the best of our knowledge, there is no efficient (implementation of) t-out-of-n OT. It is known that a t-out-of-n OT can be derived by sequentially executing any 1-out-of-n OT t times. Therefore, we apply this approach to the most efficient 1-out-of-n OT proposed in [38] to estimate the performance of state-of-the-art efficient t-out-of-n and compare it with our Priority OT. Table 7 presents the comparison results.

As the table shows, Priority OT demonstrates lower runtime across all configurations. For smaller values of t (e.g., $t = 2$), the improvement rate is 1.8, indicating that Priority OT is about 1.8 times faster. As t increases, the improvement rate becomes more significant, reaching a peak of 10 times faster for $t = 12$. This improvement across all values of t shows that Priority OT outperforms OT in [38], especially as t increases. As Figure 11 (in Appendix C) shows, both schemes' runtime linearly grows as the number of invocations increases.

Table 7: The table compares the runtime (in ms) of Priority OT and the OT proposed in [38], for different values of t and various number of OT invocations, when $n = 16$. For the OT in [38] the message size is only 4 bits. The text in blue shows the highest average improvement rate attained by our scheme.

Protocol	t	Number of invocations				Average
		1.25×10^5	2.5×10^5	5×10^5	1.25×10^6	Rate
OT in [38]	2	4320	8460	17000	43360	1.8
	4	8640	16920	34000	86720	3.6
	6	12960	25380	51000	130080	5.2
	8	17280	33840	68000	173440	6.9
	10	21600	42300	85000	216800	8.5
	12	25920	50760	102000	260160	10
Priority OT	2	2228.56	4529.27	9290.04	23495.6	1
	4	2346.3	4713.24	9554.61	23963.7	1
	6	2400.08	4897.03	9823.35	24541.6	1
	8	2408.6	4985.46	9934.83	24569	1
	10	2448.86	4993.86	9999.01	25249.2	1
	12	2478.03	5136.36	10323.4	26076.2	1

7.3 Communication Cost

We initially focus on Helix OT. A receiver R sends n messages to the sender S (in step 1b) and transmits two strings (in step 2d), where the size of each string is $\log_2(n)$. Thus, its overall communication complexity is $O(n)$. However, in total, R downloads and receives a single message of size 128-bit, which happens in step 4b. The sender's overall complexity is $O(n)$, as it sends n encrypted messages to H in step 3d. The communication complexity of H is $O(1)$ as it always sends a single message to R, in step 4b. Note that the size of each message in this scheme is relatively short, e.g., 128 bits.

We proceed to analyze the communication cost of Priority OT. A receiver R sends n messages to the sender S (in step 1b). It also sends n messages to S and t messages to H in step 2c. Therefore, the communication complexity of R is $O(n)$. Nevertheless, R needs to download only t messages from H, which occurs in step 4b. R obtains these t messages sequentially, one-by-one, based on their priority level. The communication complexity of S is $O(n)$ as it sends n messages to H, in step 3c. The total communication complexity of H is $O(t)$ as it (sequentially) sends t messages to R in step 4b.

Communication Cost Comparison. The efficient 1-out-of-n OT extensions (including the one proposed in [38]) have an overall communication complexity of $O(n)$. However, in all these schemes, a receiver's download complexity is at least $O(n)$. By sequentially invoking any of these efficient 1-out-of-n OT extensions t times, the receiver can obtain the desired messages in its preferred order. However, for each invocation, the receiver must download all n messages. If one opts to use existing t-out-of-n OT, such as those proposed in [34,52,41], the messages will not be received in order. In this case, the receiver would need

to wait until all n messages are downloaded (in the worst-case scenario), decrypt them, and then arrange the plaintext messages based on their priority. As discussed in Section 3.3, there exists OTs (e.g., those in [13,28,58,16]) that support constant response size; however, (i) they require the receiver to locally store the encryption of the *entire database*, and (ii) they are not based on OT extensions, consequently are not efficient. In contrast, our Priority OT ensures that the receiver obtains messages according to its preferences (e.g., the highest priority message is received first), receiving only one message at a time.

7.4 Features Comparison

As discussed in Section 3.2, there are unconditionally or post-quantum secure OTs, such as those schemes proposed in [42,10,19,20,33,50,9,47,39,23,5]. However, briefly, they either rely on exotic assumptions or not unconditionally secure. Specifically, the unconditionally secure OTs proposed in [42,10] use multiple servers that maintain an identical copy of the database. Other unconditionally secure OTs like the one proposed in [19,20,33] rely on noisy channels. Moreover, the OT in [50] that achieves unconditionally secure OT by using a fully trusted initializer. There have been efforts to develop post-quantum secure OTs, like those proposed in [9,47,39,23,5], but they still rely on various computational assumptions, accordingly are not unconditionally secure. In contrast, Helix OT and Priority are unconditionally secure. Table 8 compares features of Helix OT and Priority OT with several state-of-the-art OTs.

Table 8: Feature Comparison of OTs.

Protocol	Unconditional security	Post-quantum security	Constant size response	No database replications	No noisy channel	No trusted initialization	Type
STD–OT [4]	×	×	×	✓	✓	✓	1-2
STD–OT [43]	×	×	×	✓	✓	✓	1-2
RO–OT [43]	×	×	×	✓	✓	✓	1-2
[38]	×	×	×	✓	✓	✓	1-n
[42]	✓	✓	✓	×	✓	✓	1-2
[10]	✓	✓	✓	×	✓	✓	1-n
[19]	✓	✓	×	✓	×	✓	1-2
[20]	✓	✓	×	✓	×	✓	1-2
[33]	✓	✓	×	✓	×	✓	1-2
[50]	✓	✓	✓	✓	✓	×	1-2
[9]	×	✓	×	✓	✓	✓	1-n
[47]	×	✓	×	✓	✓	✓	1-2
[39]	×	✓	×	✓	✓	✓	1-2
[23]	×	✓	×	✓	✓	✓	1-2
[5]	×	✓	×	✓	✓	✓	1-2
[1]	✓	✓	✓	✓	✓	✓	1-2
Helix OT	✓	✓	✓	✓	✓	✓	1-n
Priority OT	✓	✓	✓	✓	✓	✓	t-n

References

1. Abadi, A., Desmedt, Y.: Supersonic OT: fast unconditionally secure oblivious transfer. IACR Cryptol. ePrint Arch. (2024)
2. anonymous: Source code of Helix OT (2024), https://github.com/anonymous2012000/Code-OT-/blob/main/Helix-OT--1-out-of-n-OT/main.cpp
3. anonymous: Source code of Priority OT (2024), https://github.com/anonymous2012000/Code-OT-/blob/main/Priority-OT--ordered-t-out-of-n-OT/main.cpp
4. Asharov, G., Lindell, Y., Schneider, T., Zohner, M.: More efficient oblivious transfer and extensions for faster secure computation. In: CCS'13 (2013)
5. Barreto, P., Oliveira, G., Benits, W.: Supersingular isogeny oblivious transfer. IACR Cryptol. ePrint Arch. (2018)
6. Berti, F., Hazay, C., Levi, I.: LR-OT: leakage-resilient oblivious transfer. IACR Cryptol. ePrint Arch. (2024)
7. Blakley, G.R.: One time pads are key safegaurding schemes, not cryptosystems. fast key safeguarding schemes (threshold schemes) exist. In: IEEE S&P (1980)
8. Blazy, O., Chevalier, C.: Generic construction of uc-secure oblivious transfer. In: ACNS (2015)
9. Blazy, O., Chevalier, C., Vu, Q.: Post-quantum uc-secure oblivious transfer in the standard model with adaptive corruptions. In: ARES (2019)
10. Blundo, C., D'Arco, P., Santis, A.D., Stinson, D.R.: On unconditionally secure distributed oblivious transfer. J. Cryptol. (2007)
11. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Resch, N., Scholl, P.: Oblivious transfer with constant computational overhead. In: Advances in Cryptology - EUROCRYPT (2023)
12. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Rindal, P., Scholl, P.: Efficient two-round OT extension and silent non-interactive secure computation. In: CCS (2019)
13. Camenisch, J., Neven, G., Shelat, A.: Simulatable adaptive oblivious transfer. In: Advances in Cryptology - EUROCRYPT (2007)
14. Chen, Y., Chou, J., Hou, X.: A novel k-out-of-n oblivious transfer protocols based on bilinear pairings. IACR Cryptol. ePrint Arch. (2010)
15. Chu, C., Tzeng, W.: Efficient k-out-of-n oblivious transfer schemes with adaptive and non-adaptive queries. In: PKC (2005)
16. Chu, C., Tzeng, W.: Efficient k-out-of-n oblivious transfer schemes. J. Univers. Comput. Sci. (2008)
17. Corniaux, C.L.F., Ghodosi, H.: A verifiable 1-out-of-n distributed oblivious transfer protocol. IACR Cryptol. ePrint Arch. (2013)
18. Couteau, G., Devadas, L., Devadas, S., Koch, A., Servan-Schreiber, S.: Quietot: Lightweight oblivious transfer with a public-key setup. IACR Cryptol. ePrint Arch. (2024)
19. Crépeau, C., Kilian, J.: Achieving oblivious transfer using weakened security assumptions (extended abstract). In: FoCS (1988)
20. Crépeau, C., Morozov, K., Wolf, S.: Efficient unconditional oblivious transfer from almost any noisy channel. In: Security in Communication Networks, 4th International Conference, SCN (2004)
21. Desmedt, Y., Abadi, A.: Delegated-query oblivious transfer and its practical applications. CoRR (2024)

22. Dong, C., Chen, L., Wen, Z.: When private set intersection meets big data: an efficient and scalable protocol. In: CCS (2013)
23. Dowsley, R., van de Graaf, J., Müller-Quade, J., Nascimento, A.C.A.: Oblivious transfer based on the mceliece assumptions. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. (2012)
24. Even, S., Goldreich, O., Lempel, A.: A randomized protocol for signing contracts. Commun. ACM (1985)
25. Fredkin, E., Toffoli, T.: Conservative logic. In: International Journal of Theoretical Physics (1982)
26. GNU Project: The gnu multiple precision arithmetic library (1991), https://gmplib.org
27. Goldreich, O.: The Foundations of Cryptography - Volume 2, Basic Applications. Cambridge University Press (2004)
28. Green, M., Hohenberger, S.: Universally composable adaptive oblivious transfer. In: ASIACRYPT (2008)
29. Gunupudi, V., Tate, S.R.: Generalized non-interactive oblivious transfer using count-limited objects with applications to secure mobile agents. In: FC (2008)
30. Harnik, D., Ishai, Y., Kushilevitz, E.: How many oblivious transfers are needed for secure multiparty computation? In: CRYPTO (2007)
31. Henecka, W., Schneider, T.: Faster secure two-party computation with less memory. In: CCS (2013)
32. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending oblivious transfers efficiently. In: CRYPTO, (2003)
33. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Prabhakaran, M., Sahai, A., Wullschleger, J.: Constant-rate oblivious transfer from noisy channels. In: CRYPTO (2011)
34. Jain, A., Hari, C.: A new efficient protocol for k-out-of-n oblivious transfer. Cryptologia (2010)
35. Jarecki, S., Liu, X.: Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection. In: TCC (2009)
36. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. CRC Press (2007)
37. Knuth, D.E.: The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition. Addison-Wesley (1981)
38. Kolesnikov, V., Kumaresan, R.: Improved OT extension for transferring short secrets. In: CRYPTO (2013)
39. Kundu, N., Debnath, S.K., Mishra, D.: 1-out-of-2: post-quantum oblivious transfer protocols based on multivariate public key cryptography. Sādhanā (2020)
40. Liu, M., Hu, Y.: Universally composable oblivious transfer from ideal lattice. Frontiers Comput. Sci. (2019)
41. Naor, M., Pinkas, B.: Oblivious transfer and polynomial evaluation. In: STOC (1999)
42. Naor, M., Pinkas, B.: Distributed oblivious transfer. In: ASIACRYPT. Springer (2000)
43. Naor, M., Pinkas, B.: Efficient oblivious transfer protocols. SODA (2001)
44. Nielsen, J.B.: Extending oblivious transfers efficiently - how to get robustness almost for free. IACR Cryptol. ePrint Arch. (2007)
45. Orlandi, C., Scholl, P., Yakoubov, S.: The rise of paillier: Homomorphic secret sharing and public-key silent OT. In: EUROCRYPT (2021)
46. Patra, A., Sarkar, P., Suresh, A.: Fast actively secure OT extension for short secrets. In: NDSS (2017)
47. Peikert, C., Vaikuntanathan, V., Waters, B.: A framework for efficient and composable oblivious transfer. In: CRYPTO (2008)

48. Rabin, M.O.: How to exchange secrets with oblivious transfer (1981)
49. Ren, Z., Yang, L., Chen, K.: Improving availability of vertical federated learning: Relaxing inference on non-overlapping data. ACM Trans. Intell. Syst. (2022)
50. Rivest, R.: Unconditionally secure commitment and oblivious transfer schemes using private channels and a trusted initializer. technical report (1999)
51. Tzeng, W.: Efficient 1-out-n oblivious transfer schemes. In: Naccache, D., Paillier, P. (eds.) PKC (2002)
52. Wei, X., Zhao, C., Jiang, H., Xu, Q., Wang, H.: Practical server-aided k-out-of-n oblivious transfer protocol. In: GPC. Lecture Notes in Computer Science (2016)
53. Wiesner, S.: Conjugate coding. SIGACT News **15**(1), 78–88 (1983)
54. Wu, Q., Zhang, J., Wang, Y.: Practical t-out-n oblivious transfer and its applications. In: ICICS (2003)
55. Xu, G., Li, H., Zhang, Y., Xu, S., Ning, J., Deng, R.H.: Privacy-preserving federated deep learning with irregular users. IEEE Trans. Dependable Secur. Comput. (2022)
56. Yang, Q., Liu, Y., Chen, T., Tong, Y.: Federated machine learning: Concept and applications. ACM Trans. Intell. Syst. Technol. (2019)
57. Yao, A.C.: Protocols for secure computations (extended abstract). In: 23rd Annual Symposium on Foundations of Computer Science (1982)
58. Zhang, B., Lipmaa, H., Wang, C., Ren, K.: Practical fully simulatable oblivious transfer with sublinear communication. In: FC (2013)
59. Zhao, S., Song, X., Jiang, H., Ma, M., Zheng, Z., Xu, Q.: An efficient outsourced oblivious transfer extension protocol and its applications. Secur. Commun. Networks (2020)

A Proof of Theorem 3

Proof. Initially, we discuss why Priority OT satisfies the download efficiency property, w.r.t. Definition 4. In this protocol, the size of each message that the receiver R obtains (in step 4b) can be arbitrary and is upper-bounded by the security parameter σ. The size of this message, depending on the security parameter, can be set to 128-bit. This size is $O(1)$ with respect to the total number of messages n held by the sender S. Furthermore, in total, R obtains exactly t messages, all of which are sent by H (in step 4b). Hence, the total complexity of messages obtained by R is $O(t)$.

We proceed to argue that Priority OT meets the order-respecting property, w.r.t. Definition 5. In the protocol, the original priority vector $\boldsymbol{p}$ contains the priority-based ordered indices of R's t preferred elements in $[1, \ldots, n]$. The vector $\boldsymbol{y}$ that R constructs in step 2b using $\boldsymbol{p}$, determines the position of these indices in $\boldsymbol{p}$ after they are permuted according to the permutation map $\boldsymbol{w}$. This vector $\boldsymbol{y}$ still maintains the order of R's t preferred indices, in the sense that $\boldsymbol{y}[0]$-th element in $\boldsymbol{w}$ is the index of the highest priority message, $\boldsymbol{y}[1]$-th element in $\boldsymbol{w}$ is the index of the second highest priority message, up to $\boldsymbol{y}[t-1]$ which is $\boldsymbol{y}[t-1]$-th element in $\boldsymbol{w}$, referring to the lowest priority message.

In step 4a, H retrives and appends to $\boldsymbol{u}$ each encrypted value from $\boldsymbol{x}$ one-by-one based on the order determined by $\boldsymbol{y}$. Therefore, the order of elements in $\boldsymbol{u}$ is ultimately determined by the order that $\boldsymbol{p}$ specifies. Since in step 4b H

sends to R each element of $\boldsymbol{u}$ sequentially starting from 0-th element, R receives its preferred messages sequentially according to the priority vector $\boldsymbol{p}$ it initially defined. Thus, Priority OT is order-respecting, w.r.t. Definition 5. □

B Runtime Comparison of Base OTs

As Figure 10 shows, both STD-OT schemes implementations (blue and orange bars) have the highest runtime, around 10^3 ms, with slight differences between them. The RO-OT implementation (green bar) also has a high runtime of 2.46 ms, but is slightly faster than the STD-OT schemes. Supersonic OT (red bar) is much faster than the other protocols. Both Helix OT (black bar) and Priority OT (light blue bar) show very low runtime. They are much more efficient compared to the other protocols except for Supersonic OT. Note that the negative values in the figure are a result of the logarithmic scale used in the chart.

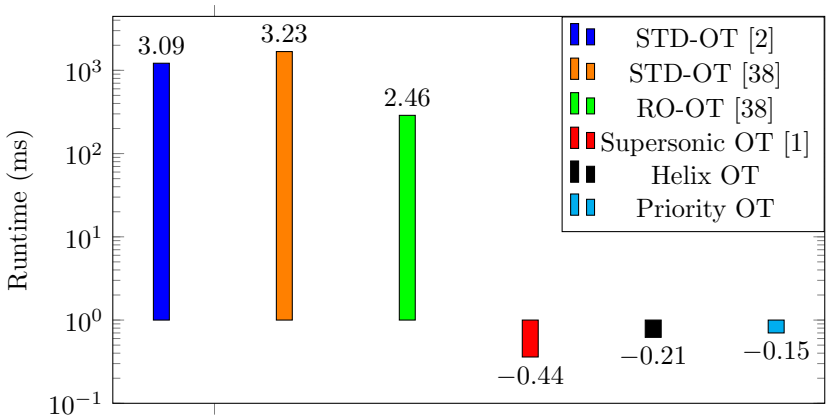


Fig. 10: Runtime comparison of Helix OT, Priority OT, STD–OT in [4], STD–OT [43], RO–OT [43], and Supersonic OT [1], shown on a logarithmic scale.

C Runtime Comparison of Priority OT and OT in [38]

As Figure 11 demonstrates, for both protocols, the runtime increases linearly as the number of invocations grows. Priority OT consistently shows a lower runtime compared to the OT in [38] for all values of t. For smaller values of t (e.g., $t = 2$), the difference in runtime between Priority OT and OT in [38] is noticeable but not as large as for higher values of t. As t increases (e.g., $t = 10$, or $t = 12$), the gap in runtime between the two protocols becomes more pronounced, highlighting the scalability advantage of Priority OT.

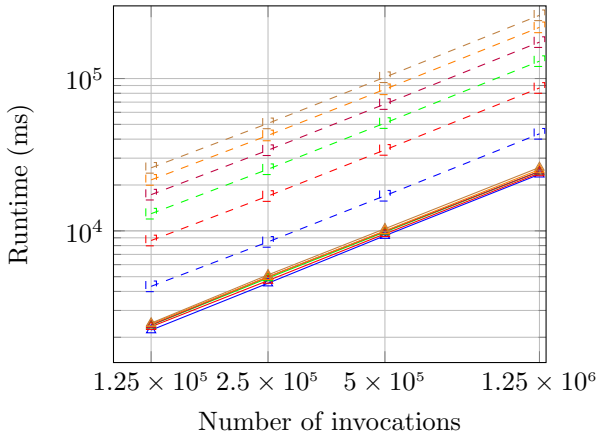


Fig. 11: Runtime comparison of OT in [38] and Priority OT for different values of t.