Achieving Obliviousness in Compressed Data Structures

Mariagiovanna Rotundo[1][a] and Paolo Ferragina[1,2][b]
[1]*Department of Computer Science, University of Pisa, Pisa, Italy*
[2]*Department of L'EMbeDS, Sant'Anna School of Advanced Studies, Pisa, Italy*

Keywords: Compressed Data Structures, Oblivious RAM, Rank and Select, Privacy.

Abstract: Oblivious RAM (ORAM) is a general-purpose storage technique that hides memory access patterns, protecting data, queries, and results from untrusted honest-but-curious servers. While traditional ORAM supports only basic reads and writes, recent works have addressed the problem of enabling richer queries by designing oblivious versions of simple data structures (e.g., maps, sets, queues). In this paper, we take a distinct step by introducing *two building blocks* that make it easy to construct many advanced oblivious and compressed data structures efficiently and securely. Our approach builds on recent results showing that most (compressed) data structures can be implemented using two fundamental primitives: Rank and Select, which count and locate symbol occurrences within a sequence. We present the first oblivious implementations of Rank and Select that require only a single ORAM access per operation, use O(1) client-side space, and add negligible client computation overhead. Thanks to their simplicity and efficiency, our oblivious Rank and Select provide a foundation for building a wide range of other oblivious compressed data structures, enabling privacy-preserving (compressed) data storage, indexing, and search in the cloud.

1 INTRODUCTION

As systems increasingly handle vast volumes of diverse and rapidly evolving information, the ability to organize, access, and manipulate data effectively becomes a cornerstone of performance and scalability. Well-chosen data structures can dramatically reduce memory usage, improve response times, and enable real-time analytics — essential qualities in modern applications ranging from search engines to recommendation systems and AI-driven platforms.

Increasing concerns about data management in the cloud have made privacy guarantees for storage and queries an additional computational feature to be optimized in data structure design. To achieve this, existing classes of privacy-preserving solutions can be leveraged. Among them, the class with the most challenging scenario is referred to as Oblivious RAM (ORAM) and aims to protect stored data, queries issued by a client on those data, and their results by ensuring that an untrusted or honest-but-curious server, responsible for storing the data and responding to queries, cannot learn anything about them.

While the literature about ORAM solutions is extensive (Stefanov et al., 2018; Kushilevitz et al., 2012; Patel et al., 2018; Devadas et al., 2015), most existing works focus on storage systems limited to supporting basic read and write operations. Some recent studies (Wang et al., 2014; Keller and Scholl, 2014; Sadakane, 2025) have investigated the design of oblivious data structures offering some sort of query operations — such as arrays, maps, sets, stacks, and suffix arrays. These data structures can be made oblivious with a trivial mapping of their input data and pointer-based structure on the nodes of an ORAM, and then supporting any of its operations by just accessing the corresponding ORAM cells. However, this approach inherits the performance of the underlying data structure, turning its time complexity into bandwidth complexity. Namely, if an operation takes T data accesses to be supported on the investigated data structure when executed in the classic RAM, its oblivious version would take T ORAM accesses. Each ORAM access involves several random shuffling and decryption and re-encryption operations (at the client side, see Section 2), as well as T block-wise data accesses and exchanges over the cloud, making the overall cost impractical.

[a] https://orcid.org/0009-0001-1671-7407
[b] https://orcid.org/0000-0003-1353-360X

Rotundo, M. and Ferragina, P.
Achieving Obliviousness in Compressed Data Structures.
DOI: 10.5220/0015059800004103

In *Proceedings of the 23rd International Conference on Security and Cryptography (SECRYPT 2026) - Volume 2*, pages 989-1001
ISBN: 978-989-758-858-7; ISSN: 2184-7711

1.1 Our Contribution

This paper provides one of the first focused analyses on the oblivious design of compressed[1] data structures, which usually improve pointer-based data structures both in time and space. In ORAM scenarios, using compressed data structures rather than their pointer-based alternative may significantly reduce the communication cost, the encryption/decryption cost, and the data reshuffling time introduced by the ORAM protocol. This improvement can result from both the smaller total amount of data stored on the server, thus the smaller ORAM tree, and from the ability to store more (compressed) data per each blocks stored in the ORAM tree.

We address the above problem focusing on a solution that is *not pointwise* — namely by choosing some data structures and turning them into oblivious ones —, but we aim at providing a novel design scheme that allows us to *uniformly and easily turn many known compressed data structures into ORAM-based ones*. The specialty of our design scheme is that it is simple, offers efficient theoretical guarantees, and above all, is practically usable.

A key insight behind our approach is drawn from a modern trend in data structure design: building efficient representations upon two fundamental primitives, Rank and Select (Mäkinen and Navarro, 2007; González et al., 2005; Navarro and Providel, 2012; Navarro, 2016; Ferragina, 2023). These primitives respectively count and locate symbol occurrences within a sequence and serve as the foundation for a broad range of state-of-the-art compressed data structures, including arrays, hash tables, trees, and graphs, that match the performance of their classical counterparts while achieving significantly reduced space usage (Google, 2007; Ferragina, 2023; Navarro, 2016).

As said above, Rank and Select can be easily made oblivious by mapping existing non-oblivious solutions into an ORAM or by storing, for each valid index i, the precomputed values $rank(i)$ and $select(i)$ inside the ORAM. The former approach requires multiple ORAM operations per query, while the latter requires fewer but significantly more expensive ORAM accesses due to the larger amount of data to be stored.

However, in this paper, we do not propose an implementation of Rank and Select based on trivial mapping. Instead, we introduce a new design for Rank and Select data structures that is both time and space efficient, and and achieves the optimal cost of a single ORAM access per operation, using an ORAM tree that is as small as possible.

Our experimental evaluation shows that the proposed design minimizes client-side storage, keeps server space usage close to that of non-oblivious solutions, and incurs only negligible in-block computation overhead (on the order of a few microseconds), effectively hidden by the ORAM communication cost.

Finally, building on these primitives, we demonstrate that a wide range of modern (and possibly compressed) data structures, such as trees, predecessor/successor structures, and Google's sparse hash, can be made oblivious without redesigning each of them from scratch by easily deploying our oblivious Rank and Select and what it is already known in the data-structure literature.

We believe that our work paves the way for the development of a software library of privacy-preserving compressed data structures for both static and dynamic scenarios (supporting updates). In the dynamic setting, our approach can efficiently handle updates leveraging a periodic reconstruction of the static data structures, a strategy widely used in many dynamic solutions, such as the Log-Structured Merge-tree (LSM-tree) (O'Neil et al., 1996; Luo and Carey, 2020) and RocksDB (Platforms, 2022).

In this work, we adopt the Ring ORAM protocol. Nevertheless, our approach is not limited to this choice; other ORAM schemes based on fixed-length blocks can also be employed, as our new design is independent of the underlying ORAM protocol.

2 RING ORAM

Ring ORAM (Ren et al., 2015) is a tree-based oblivious RAM protocol that organises the server storage as a perfect binary tree of height $L = O(\log n)$, where n is the number of blocks in which the input data are stored.

Each node in this tree, known as a *bucket*, holds S fixed-length dummy blocks (with dummy content) and Z fixed-length blocks that can contain either real or dummy data. Real data blocks are randomly and logically assigned to leaves in the tree but are physically stored in a block residing in a bucket along the path from the root to their assigned leaf. The mapping between real blocks and assigned leaves is kept

[1] In the current literature, researchers (Navarro, 2016) use two different terms to refer to data structures that use *reduced* space occupancy: succinct and compressed. The term *succinct* denotes data structures that store n objects using only $o(n)$ bits of extra space. In contrast, the term *compressed* denotes data structures that occupy space proportional to the entropy of the input data. For ease of reading, throughout the rest of this paper, we use the term "compressed" to refer to both classes of data structures, as their respective space complexities will make it clear which class is being discussed in each context.

in a so-called *position map*, which assumes the form of an array of integers, one for each real data block. To guarantee privacy, all the blocks, both with real and dummy content, are encrypted with randomized ciphers, which makes the encryption of a block different every time it is encrypted.

In our cloud-based scenario, the Ring ORAM is assumed to be stored in a honest-but-curious server, whereas the position map is stored in a thin client.

To access a block, the client first queries its local position map to determine the block's current leaf assignment and then asks the server to retrieve some metadata for the corresponding path in its Ring ORAM. This metadata allows the client to locate the queried block precisely and to distinguish between real and dummy blocks. As shown in (Ren et al., 2015), the client fetches from the Ring ORAM, present in the server, one dummy block in each bucket along the queried path, except for the one containing the data of interest.

To preserve obliviousness, Ring ORAM *reshuffles* a path every A accesses, by redistributing its real blocks across the same path, and ensures that no block is accessed more than once between two shuffles.

In this paper, we adopt Ring ORAM as the reference protocol for evaluating our oblivious Rank/Select design on practical examples. However, the approach we propose is not limited to Ring ORAM and can be combined with any general-purpose ORAM protocol.

3 THE RANK PRIMITIVE

Rank (Mäkinen and Navarro, 2007; González et al., 2005; Navarro and Providel, 2012; Navarro, 2016; Ferragina, 2023) is a primitive that, given a sequence of length n over an alphabet Σ and an index i, returns the **number of occurrences** of a symbol $\sigma \in \Sigma$ up to position i in the sequence. Following common practice in the literature, we restrict our attention in this paper to binary sequences, hence $\Sigma = 0,1$, and denote by $rank_1$ the Rank operation for $\sigma = 1$, and by $rank_0$ the Rank operation for $\sigma = 0$. In the following, we restrict our attention to $rank_1$ since $rank_0$ can be computed as $i - rank_1(i)$.

A trivial approach to answering $rank_1(i)$ is to scan the sequence up to the target position i and count the occurrences of 1s. However, this method is highly inefficient for long sequences, as it may require scanning the entire sequence in the worst case. More time-efficient solutions do exist in the literature. Among them, we refer to the one described in (Navarro, 2016, §4.2.2), (Ferragina, 2023, §15.1.1), which speeds up the Rank operation by dividing the sequence into blocks and sub-blocks, and storing some partial information about them in an auxiliary two-level data structure, described below.

Take two values K and k such that $k < K < n$. We divide the input binary sequence into fixed-size *big blocks* of K bits each and store, for each big block, its *absolute rank*, i.e., the number of 1s that precede the block in the sequence. Each big block is further divided into $\lceil K/k \rceil$ *small blocks* of size k, and for each small block, we store its *relative rank*, i.e., the number of 1s that precede it within the same big block. Finally, a lookup table with 2^k rows, one for each possible combination of k bits, and k columns, one for each position within a combination, is constructed. This table stores, for each combination, the rank of 1s at each of its positions.

With this information, we can efficiently answer $rank_1(i)$ in $O(1)$ time and exactly 3 memory accesses by (i) identifying the correct big and small blocks, (ii) retrieving their corresponding absolute and relative ranks, (iii) using the content of the small block to access the appropriate position in the lookup table, and (iv) summing all the retrieved partial information.

Provided that the values of K and k are appropriately chosen (Ferragina, 2023; Navarro, 2016), this solution require $o(n)$ additional space.

Lemma 1. *We can support constant-time Rank operations on a binary sequence of length n by using only $o(n)$ bits of additional storage. The input sequence is not modified, and thus it is assumed to be read-only.*

4 THE SELECT PRIMITIVE

Select (Mäkinen and Navarro, 2007; González et al., 2005; Navarro and Providel, 2012; Navarro, 2016; Ferragina, 2023) is a primitive that, given a sequence of length n over an alphabet Σ and an index x, returns the **position of the x^{th} occurrence** of a symbol $\sigma \in \Sigma$ in the sequence. As done for the Rank, in the following, we consider binary sequences, hence $\Sigma = 0,1$, and denote by $select_1$ the Select operation for $\sigma = 1$, and by $select_0$ the Select operation for $\sigma = 0$.

We notice that $select_1$ and $select_0$ cannot be derived one from the other: rather, they need a specific implementation that we detail below just for $select_1$, for the sake of presentation (for $select_0$ we follow the same construction but looking at 0s instead of 1s).

A trivial approach to answering $select_1$ is again to scan the sequence up to the x^{th} occurrence of the symbol of interest, but possibly scanning of the whole sequence. We can use more time-efficient solutions like the one described in (Navarro, 2016, §4.3.3), (Ferrag-

ina, 2023, §15.1.1), which speeds up the Select by dividing the sequence again into blocks and sub-blocks, whose lengths now depend on their contents, in a sophisticated way that we recall below (and refer to the literature for further details).

Let us assume that the input sequence of n bits is divided into variable-size *big blocks*, each containing K 1s (where K is a parameter to be set next). Big blocks whose length exceeds K^2 bits are called *sparse*, and for these, we store the *absolute positions* of the 1s contained in them. In contrast, big blocks whose length is at most K^2 bits are called *dense* and are further subdivided into *small blocks*, each containing k 1s (where k is a parameter to be set next). Each small block, in turn, may be sparse or dense depending on its total length: if it exceeds k^2, it is called sparse; otherwise, it is called dense. For sparse small blocks, we store the *relative positions* of the 1s within their respective big block. For the dense small blocks, we use a lookup table with 2^{k^2} rows, one for each possible combination of k^2 bits (which is the maximum size of a small dense block), and k columns storing the relative positions of the 1s within the small block.

With this information, we can efficiently answer a $select_1(x)$ in $O(1)$ time, performing up to 3 memory accesses at most: to the absolute positions, to the relative positions, and to the lookup table, summing all the retrieved partial information of interest.

Provided that the values of K and k are appropriately chosen (Ferragina, 2023; Navarro, 2016), this solution require $o(n)$ additional space.

Lemma 2. *We can support constant-time Select operations on a binary sequence of length n by using only $o(n)$ bits of additional storage. The input sequence is not modified, and thus it is assumed to be read-only.*

5 OUR CONTRIBUTION

In this paper, we present a surprisingly simple approach to implement oblivious Rank and Select based on two ideas: (i) partitioning the input binary sequence into properly chosen blocks, so that the partial information per block takes negligible space, and (ii) mapping the blocks into a Ring ORAM answering each *rank* and *select* with exactly one ORAM access

Specifically, starting from the ideas described in Section 3 and Section 4, we propose a novel construction of a Rank/Select block, which we refer to as an *ORAM block*. It encapsulates all the information necessary to answer a *rank*/*select* operation on its bits: the absolute and relative ranks/positions, and the block content when required. These blocks are then encrypted and stored on the server within a Ring ORAM, and can be efficiently retrieved by the client.

Throughout the paper, we assume that indices and positions start from 0.

5.1 Oblivious Rank

To obliviously support *rank* operations, we create one ORAM block for each big block (as described in Section 3). We store in each ORAM block (i) the absolute rank of the big block, (ii) the relative ranks of all its small blocks, and (iii) the binary sequence in the big block. Different from the idea shown in Section 3, we do not store a direct-access table, and we do not have a two-level block scheme.

If n is not a multiple of K, the last big block may contain fewer than $\lceil K/k \rceil$ small blocks and K bits. In such cases, we pad the relative ranks and the binary sequence stored in the corresponding ORAM block with zeros, ensuring that each block has the same size and contains exactly $\lceil K/k \rceil - 1$ relative ranks and a sequence of K bits. Note, however, that the space overhead introduced by padding is negligible, as it affects at most one ORAM block: the final one.

Using this structure, we can efficiently answer $rank_1(i)$ by performing the following steps:

1. we identify the big and small blocks containing position i by computing $\lfloor i/K \rfloor$ and $\lfloor (i \mod K)/k \rfloor$, respectively;
2. we determine the starting position of the small block within the binary sequence stored in the ORAM block by computing $\lfloor (i \mod K)/k \rfloor \cdot k$;
3. we scan the next $((i \mod K) \mod k) + 1$ bits of that binary sequence to reach the bit in position i;
4. we compute $rank_1(i)$ by summing the absolute rank, the relative rank, and the number of 1s encountered during the scan.

Assume we want to answer $rank_1(13)$ on the sequence `0110011000011110 10`, with $K = 9$ and $k = 3$, from which we construct the two ORAM blocks `<0, 2, 3, 011001100>` and `<4, 1, 4, 001111010>`. Here, the absolute rank of the big block if followed by $\lceil K/k \rceil - 1$ relative ranks of the small blocks, and the content of the big block. For each big block, the relative rank of the first small block is always 0 and thus not stored. Position 13 lies within the big block B_p where $p = \lfloor 13/9 \rfloor = 1$. Therefore, we access the second ORAM block, with absolute rank 4. The small block b_q containing position 13 has index $q = \lfloor (13 \mod 9)/3 \rfloor = 1$, and relative rank 1 (relative rank of the small block with index $q = 0$ is omitted). Then, we locate the start of the small block, which is at position $\lfloor (13 \mod 9)/3 \rfloor \cdot 3 = 3$ within the binary sequence.

From there, we scan $((13 \mod 9) \mod 3) + 1 = 2$ bits. During this scan, we encounter 2 1s. Finally, summing the absolute rank (4), the relative rank (1), and the number of 1s found during the scan (2), we compute $rank(13) = 4 + 1 + 2 = 7$.

We can support $rank_0$ operations by computing them as $rank_0(i) = i - rank_1(i)$.

5.1.1 Time and Space Complexity

Answering a *rank* requires three values: the big block's absolute rank, the small block's relative rank, and the result of the scan. The first two are retrieved in $O(1)$ time from the ORAM block, while the scan involves at most k bits. Hence, the worst-case time complexity is $O(k)$.

About the space complexity, each ORAM block has the following size, measured in bits:

$$\log n + \left(\left\lceil \frac{K}{k} \right\rceil - 1 \right) \log K + K \tag{1}$$

where $\log n$ bits represent the absolute rank, $(\lceil K/k \rceil - 1) \log K$ bits the relative ranks, and K bits the binary subsequence. Being the big blocks $\lceil n/K \rceil$, our approach requires the following space (in bits):

$$\left\lceil \frac{n}{K} \right\rceil \left(\log n + \left(\left\lceil \frac{K}{k} \right\rceil - 1 \right) \log K + K \right) \tag{2}$$

From (2) we observe that increasing K reduces the number of ORAM blocks, at the cost of larger blocks. Likewise, increasing k reduces the number of small blocks, and thus the space needed for relative ranks, but increases the *rank* time, which grows with k.

By choosing $K = \log^2 n$ and $k = (\log n)/2$ we can support *rank* operations in $O(\log n)$ time using $n + o(n)$ bits of space, matching the space of Lemma 1. In fact, by substituting these values into (2), we get the following total space (in bits):

$$\frac{n}{\log n} + \frac{4n \cdot \log\log n}{\log n} + n \tag{3}$$

Lemma 3. *We can support a Rank operation on a binary sequence of length n in $O(\log n)$ worst-case time, and occupying a total of $n + o(n)$ bits.*

Our approach performs worse than the best non-oblivious solution in terms of runtime (see Section 3), but its design requires *a single Ring ORAM access* to a block of size $O(\log^2 n)$ per *rank* operation.

5.1.2 Integration with Ring ORAM

To make *rank* oblivious, we store the ORAM blocks constructed as described above in a Ring ORAM tree on a server. Then, to perform an oblivious *rank* operation, we follow the same algorithm outlined in Section 5.1, but once the correct big block is identified, the corresponding ORAM block is retrieved from the server via Ring ORAM protocol. Therefore, each $rank(i)$ operation accesses only one ORAM block, since all the necessary partial information related to position i is contained within that block. This proves the following via an extension of Lemma 3:

Lemma 4 (Oblivious rank)**.** *We can support an oblivious Rank operation on a binary sequence of length n in $O(\log n)$ worst-case time, executing one single Ring ORAM access to a block of size $O(\log^2 n)$ bits, and occupying a total of $n + o(n)$ bits.*

Although our approach performs an optimal single Ring ORAM access per *rank* operation, its runtime must include the cost of fetching the corresponding ORAM block from the server. To better understand the impact of using Ring ORAM on Rank performance, we provide a brief analysis of ORAM blocks size here and defer a deeper study to Section 6.

For example, consider a sequence of length $n = 2^{64}$, with parameters $K = \log^2 n = 4096$ bits and $k = (\log n)/2 = 32$ bits. From (1), we know that each ORAM block requires 5684 bits (approximately 711 bytes), an amount of data that can be retrieved efficiently from the server.

5.2 Oblivious Select

As for the *rank*, we support oblivious *select* by creating one ORAM block for each big block, defined as in Section 4, but storing sparse and dense big blocks by using different representations that accommodate their structural differences, and be able to squeeze everything in one single ORAM block.

Assuming we want to support $select_1$ operations, we proceed as follows ($select_0$ is symmetric). For each sparse big block, we create an ORAM block whose first bit (referred to as *marking bit*) indicates that the block is sparse, followed by the K absolute positions of all the occurrences of 1 within the block. For each dense big block, we store a marking bit indicating that the block is dense, then the starting position of the big block in the binary sequence, and the following partial information about its small blocks: (i) a binary sequence that, in order, marks each small block as sparse or dense (also referred to as marking bits); (ii) the k relative positions of the 1s in the sparse small blocks; (iii) the binary subsequences of the dense small blocks. If a sparse small block contains fewer than k bits, its relative positions are padded with dummy values equal to 0.

Using this structure, we can efficiently answer $select_1(x)$ by performing the following steps:

1. we identify the big block containing the x^{th} occurrence of 1 by computing $\lfloor (x-1)/K \rfloor$. Then, we check the marking bit of the corresponding ORAM block to determine whether the big block is sparse or dense;
2. if the big block is sparse, we retrieve the absolute position of the x^{th} occurrence of 1 by accessing the value at index $(x-1) \mod K$;
3. if the big block is dense, we identify its small block containing the x^{th} occurrence by computing $\lfloor (x-1 \mod K)/k \rfloor$. We then check the marking bit of the small block and proceed as follows:
   (a) if the small block is sparse, we count the number of sparse small blocks before it by scanning the subsequence. Let s_x be the number of sparse small blocks preceding the one that contains the x^{th} occurrence of 1. We then access the value $s_x \cdot k + ((x-1 \mod K) \mod k)$, which gives the correct relative position. Finally, we compute $select_1(x)$ by adding this value to the absolute starting position of the big block.
   (b) if the small block is dense, we proceed as follows. First, we scan the subsequence to determine (i) the total number of sparse small blocks in the big block, denoted by s, (ii) the number of sparse and dense small blocks that precede the current dense small block, denoted s_x and d_x, respectively, and (iii) the number of dense small blocks that precede the sparse small blocks counted in s_x, denoted as d_{s_x}. We use s to locate in the ORAM block the binary subsequence storing the contents of the dense small blocks by skipping the $s \cdot k$ stored relative positions. Then, we use s_x to retrieve the last relative position in the big block that appears immediately before the current dense block. Finally, we scan the binary subsequence by skipping $d_{s_x} \cdot k$ occurrences of 1s. From the bit reached, we then count the number of scanned bits needed to reach the target occurrence computed as $((d_x - d_{s_x}) \cdot k + ((x-1 \mod K) \mod k) + 1$. The value of $select_1(x)$ is given by the sum of three components: the absolute position of the big block, the last relative position among the small blocks that precede the current dense small block, and the number of bits counted during the final scan.

To better understand how *select* works, let's look to an example.

Assume we want to support $select_1$ operations on the sequence `1000010001011000 01`, with parameters $K = 3$ and $k = 2$. The sequence is divided into the two big blocks `1000010001` and `01100001`. The first big block is sparse, containing 10 bits, while the second block is dense, with 8 bits. For the sparse big block, we create the ORAM block `<1, 0, 5, 9>`. Here, the first bit is set to 1 to mark the block as sparse. It is followed by the absolute positions where the 1s occur. For the dense big block, we create the ORAM block `<0, 10, 01, 7, 0, 011>`. This block starts with a bit set to 0, indicating it is dense. Next is the absolute starting position of the block, which is 10, followed by the bitmask `01` indicating the density of the small blocks (the first small block is dense, the second is sparse). Then, we store the relative position 7, which is the only position in the sparse small block where a 1 occurs. Since the sparse small block contains fewer than k occurrences of 1, we pad the list of relative positions with dummy entries set to 0. Finally, we store the subsequence `011` from the dense small block.

Assume now, we aim to answer $select_1(4)$ having the blocks above. We first identify the big block B_p, where $p = \lfloor (4-1)/3 \rfloor = 1$. This block is dense, as indicated by its marking bit being 0. We compute the index $q = \lfloor (4-1 \mod 3)/2 \rfloor = 0$ of the small block b_q that contains the 4^{th} occurrence of 1. The first bit of the sequence 01 indicates that b_0 is dense. Next, we scan the sequence 01 and count the total number of sparse small blocks ($s = 1$), the number of sparse and dense small blocks before b_0 ($s_x = 0$ and $d_x = 0$, respectively), and the number of dense small blocks that precede the first s_x sparse small blocks ($d_{s_x} = 0$). Then, we skip the first $s \cdot k = 2$ values representing relative positions of 1s in sparse blocks. We start scanning the subsequence 011, skipping no bits since $d_{s_x} = 0$, and stop when we find $(0 + ((4-1 \mod K) \mod k) + 1 = 1$ occurrence of 1, which appears at relative position 1. Finally, we sum the absolute starting position of the big block (10), the relative position from the sparse small blocks (0, since there are none), and the count of scanned bits (1), yielding $select_1(4) = 10 + 0 + 1 = 11$.

Note that in our scheme, we do not need to store the entire binary sequence on which the *select* operations are performed.

5.2.1 Time and Space Complexity

Answering $select_1(x)$ depends on the density of the block containing the x^{th} 1: (i) for sparse blocks, we retrieve the absolute position in $O(1)$ time; (ii) for sparse small blocks, we scan the $\lceil K/k \rceil$ marking bits and then retrieve the absolute and relative positions in $O(1)$ time; (iii) for dense small blocks, we scan the $\lceil K/k \rceil$ marking bits and up to K^2 bits, and sum the result with the absolute and relative positions retrieved

in $O(1)$ time. Thus, the worst-case time is $O(K^2)$.

Unlike the *rank* approach, the *select* shows blocks size that depend on the content of the big blocks.

Sparse big blocks have all the same size (in bits in the following), which depends only on K and n:

$$1 + K \cdot \log n \tag{4}$$

Here, 1 bit is reserved for the initial marking bit, and $K \log n$ bits are allocated to store the absolute positions of the K considered occurrences in each block. Under the assumption that all big blocks are sparse, they are at most $\lceil n/K^2 \rceil$, resulting in the following upper bound (in bits):

$$\left\lceil \frac{n}{K^2} \right\rceil (1 + K \cdot \log n) \tag{5}$$

Dense big blocks have different sizes that depend on their content. Here, 1 bit is reserved for the initial marking bit, $\log n$ bits are allocated for the absolute position of the big block in the input sequence, and $\lceil K/k \rceil$ bits are used for the marking bit of each of its small blocks. Assuming that all $\lceil K/k \rceil$ small blocks are sparse, we need $\lceil K/k \rceil \cdot k \cdot \log K^2$ bits to store their relative positions. On the other hand, if all small blocks are dense, we use at most K^2 bits to store their content. Hence, dense big blocks take no more than the following amounts of bits:

$$1 + \log n + \left\lceil \frac{K}{k} \right\rceil + \left\lceil \frac{K}{k} \right\rceil \cdot k \cdot \log K^2 + K^2 \tag{6}$$

Note that this upper bound is quite large, as it accounts for both the cases where all small blocks are sparse and all are dense. Moreover, while we assume K^2 bits for dense small blocks, in practice, the larger the big block size approaches K^2, the more bits belong to sparse small blocks. Conversely, when the size is closer to K, most bits are part of dense small blocks.

Under the assumption that all big blocks are dense, we can derive the following upper bound (in bits) on the total space:

$$\left\lceil \frac{n}{K} \right\rceil \left(1 + \log n + \left\lceil \frac{K}{k} \right\rceil\right) + \left\lceil \frac{n}{k^2} \right\rceil \cdot k \cdot \log K^2 + n \tag{7}$$

By setting $K = \log^2 n$ and $k = (\log\log n)^2$, we can support *select* operations in $O(\log^4 n)$ worst-case time using at most $n + o(n)$ bits of space, thus matching the space of Lemma 2. Assume for simplicity that n is a multiple of K and k^2, by substituting them into (5) we obtain the following upper bound (in bits) on the space needed to store sparse big blocks:

$$\frac{n}{\log^4 n} + \frac{n}{\log n} \tag{8}$$

Instead, substituting $K = \log^2 n$ and $k = (\log\log n)^2$ into (7), we obtain the following upper bound (in bits) on the space required to store dense big blocks:

$$\frac{n}{\log^2 n} + \frac{n}{\log n} + \frac{n}{(\log\log n)^2} + \frac{4n}{\log\log n} + n \tag{9}$$

Lemma 5. *We can support a Select operation on a binary sequence of length n in $O(\log^4 n)$ worst-case time, and using at most $n + o(n)$ bits of space.*

Unlike the proposed *rank* data structure, the $n + o(n)$ bits for the *select* data structure represents an upper bound to the space rather than the exact space.

Our approach shows a worse time than the best non-oblivious solution (see Section 4), but its design will require *a single Ring ORAM access* to a block of size at most $O(\log^4 n)$ per *select* operation.

5.2.2 Integration with Ring ORAM

To make *select* oblivious we store the ORAM blocks as constructed above in a Ring ORAM tree on a server. Then, to support an oblivious *select*, we follow the same algorithm illustrated in Section 5.2, but once the correct big block is identified, we retrieve the corresponding ORAM block from the server via Ring ORAM. Therefore, each *select*(x) operation accesses only one ORAM block, which includes all partial information needed to compute the correct result. This proves the following via an extension of Lemma 5:

Lemma 6 (Oblivious select)**.** *We can support an oblivious Select operation on a binary sequence of length n in $O(\log^4 n)$ worst-case time, executing one single Ring ORAM access to a block of size $O(\log^4 n)$ bits, and occupying at most $n + o(n)$ bits.*

Although our approach uses a single Ring ORAM access per *select* operation, its runtime includes the cost of fetching the corresponding block from the server. Moreover, since the *select* constructs variable-length ORAM blocks while Ring ORAM uses fixed-size blocks, we must determine how to represent and store them efficiently. We provide a brief analysis of ORAM blocks size here and defer a deeper study to Section 6.

For example, consider a sequence of length $n = 2^{64}$, with parameters $K = \log^2 n = 4096$, and $k = (\log\log n)^2 = 36$. From above, we know that each sparse and dense big block requires approximately 32.0 KiB and up to 2.0 MiB, respectively. These could be too much in some applicative scenarios, but we can reduce the block sizes by decreasing K. For instance, by setting $K = (\log^2 n)/4 = 1024$, and $k = (\log\log n)^2 = 36$, we still maintain a total space usage

of $n + o(n)$ bits, but the size of sparse and dense big blocks drops, respectively, to 8.0 KiB and 130.6 KiB.

We point out that the block size discussed above is assumed to be equal to the theoretical upper bound on dense block size, as given in (6), but this upper bound is quite large and cannot be reached in practice, as we will show in the experimental setting of Section 6.2.

Before storing the blocks in Ring ORAM, we can enforce a fixed block size by padding all variable-length block to the size of the largest one, but this may waste substantial space. A more space-efficient approach is to compact multiple consecutive blocks of the same type (dense or sparse) into a single ORAM block when their sizes are significantly smaller than that of the largest block of the opposite type. This approach reduces both the padding and the number of ORAM blocks just using a few additional bits per ORAM block that would otherwise be wasted for padding. To be more precise, as additional metadata, each ORAM block stores one bit that indicates whether it is a compacted block or not. Then, each compacted ORAM blocks stores the number of compacted big blocks stored, and a pointer to their starting position (except for the first one) to support fast random access. In each ORAM block, the marking bit, which indicates whether a big block is dense or sparse, needs to be stored only once, since we only compact big blocks of the same type.

This approach shrinks the position map, which now stores a single entry per (compacted) ORAM block, at the cost of an auxiliary bitvector with one bit per big block indicating whether it is the first big block within its ORAM block. This bitvector lets the client identify which ORAM block to fetch: for a target big block x, the corresponding ORAM block `id` is obtained by counting how many "first" marks occur before x in the bitvector.

6 EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the space needed to store the ORAM blocks built as in Section 5.1 and Section 5.2, and the time required to compute *rank* and *select* from them. We do not include the Ring ORAM protocol overhead, whose performance has been analyzed in prior work (Ren et al., 2015).

For the experiments, we consider randomly generated binary sequences with $n = 2^{32}$ bits, and three different densities of 1s: 50%, 10%, and 1%.

We measure *rank* and *select* using two implementations: (i) a baseline that follows the algorithms in Section 5.1 and Section 5.2, and (ii) an optimized version that accelerates scans with the `POPCOUNT` instruction (Wikipedia, 2025). Execution times are measured over one million queries. Queries are randomly generated among valid inputs: $rank(i)$ with $i \in [0, n-1]$, and $select_b(j)$ with $j \in [1, \#b]$.

Table 1: Oblivious rank space: ORAM block size and total space on sequences with length $n = 2^{32}$ bits.

K	k	block size (bytes)	total space (MiB)
1024	16	211	843
	128	141	563
	512	134	533
2048	16	435	870
	128	281	562
	512	265	529
32768	16	7939	993
	128	4579	573
	512	4219	528

6.1 Rank Evaluation

We separately analyze the space usage and execution time of *rank* in our ORAM-block design, and discuss its integration with Ring ORAM to demonstrate the practicality of our oblivious *rank*.

6.1.1 Space Evaluation

The block size is independent of the distribution of 1s and 0s in the input sequence and depends only on n, K, and k, as in (1). In all experiments, we fix $n = 2^{32}$. We evaluate space usage under three configurations for both big blocks and small blocks (see Table 1). Specifically, we use the baseline $K = \log^2 n = 1024$ bits and $k = (\log n)/2 = 16$ bits, and also test larger values: $K \in \{2048, 32768\}$ and $k \in \{128, 512\}$.

As predicted by (1), larger K yields larger ORAM blocks, while larger k reduces ORAM block size by decreasing the number of stored relative ranks per ORAM block. In particular, blocks stay below 500 bytes for $K \leq 2048$ and $k \geq 16$, and remain under 8 KiB even for $K = 32768$.

For total space, increasing K reduces the number of ORAM blocks but does not save space unless k also increases. Setting $k = 512$ cuts total space by $58.1\% - 88.2\%$ compared to $k = 16$. Moreover, with $k = 512$, across all tested K, our construction uses only 3–4% additional space beyond the original binary sequence while supporting *rank* in $O(k)$ time.

6.1.2 Time Evaluation

As expected from Section 5.1.1, the time required to answer a *rank* increases with k, but remains unaffected by the value of K and the distribution of bits

Table 2: Oblivious rank time: time to answer a rank operation on a binary sequence of length $n = 2^{32}$ bits.

k	scan (μs)	POPCOUNT (μs)
16	0.25	0.24
128	0.35	0.27
512	0.65	0.30
1024	1.05	0.33
2048	1.75	0.38

in the sequence. For this reason, we report in Table 2 the *rank* times at varying values of k, omitting K and the input distribution, since all configurations exhibit similar performance for the same k.

We observe that in the implementation based on a linear scan of the bits in an ORAM block, the *rank* time increases with k, ranging from 0.25 μs for $k = 16$ to 1.75 μs for $k = 2048$. In contrast, the implementation that leverages the `POPCOUNT` instruction shows the same increasing trend, but with a significantly reduced impact. Specifically, this method achieves *rank* times below 0.40 μs for all the considered values of k, being 2.1× and 4.6× faster than the linear scan implementation for $k = 512$ and $k = 2048$, respectively. For smaller $k = 16$ and $k = 128$, the `POPCOUNT` implementation exhibits results similar to those of the linear scan.

6.1.3 Integration with Ring ORAM

From (Ren et al., 2015, §5), we know that the amortized bandwidth overhead can be estimated, given L, Z, S, and A, as $1 + (L+1)(2Z+S)/A \cdot (1 + Poiss_{cdf}(S,A))$ blocks when using the XOR approach (Dautrich et al., 2014), where $Poiss_{cdf}$ denotes the cumulative distribution function of the Poisson distribution according to the parameters S and A.

Consider a Ring ORAM tree with height $L = \log(2n/A)$, where n represents the number of blocks stored in the Ring ORAM, computed as the total data size divided by the block size. Let us use the configuration $Z = 32, A = 46, S = 61$ that is proposed in (Ren et al., 2015, §5), where Z and S are, respectively, the number of real and dummy blocks per bucket, and A is the path reshuffling frequency (see Section 2). The estimated bandwidth overhead is approximately 37, 34, and 26 blocks when K is 1024, 2048, and 32768, respectively. However, since the block size increases with K, the amortized bandwidth overhead also grows when measured in bytes, ranging from about 4.8 KiB to 14.4 KiB as K increases from 1024 to 2048, and reaching up to 199.0 KiB when $K = 32768$.

Therefore, if we limit ourself to the case $K \leq 2048$, these data volumes are small and can be transferred quickly over modern networks, typically within a few milliseconds or even less. For comparison, a short email or an html-page usually occupy a few kilobytes, indicating that the communication costs of our proposal are fully affordable in practice.

6.1.4 Take-Home Message

Based on these experimental results, we conclude that the additional $o(n)$ server-side space of the data structure stated in Lemma 3 ranges from 3% to 12% of the input sequence size when $k \geq 128$, being closer to the 3% when k increases with K. A suggested setting is $K = 2048$ and $k = 512$, which makes the additional space equal to 3.3% of the input binary sequence.

The time to support a *rank* operation is on the order of a few microseconds, independently of the experimented k and K, and thus negligible compared to the other operations required by Ring ORAM (Ren et al., 2015). Furthermore, communication time is also negligible due to the small amount of data to exchange in our experiments.

Overall, this makes our oblivious *rank* implementation relevant in practice.

6.2 Select Evaluation

We separately analyze the space usage and execution time of *select* in our ORAM-block design, and discuss its integration with Ring ORAM to demonstrate the practicality of our oblivious *select*.

In our experimental analysis, we assume that each ORAM block includes one big block, thus not applying the block-compaction technique described at the end of Section 5.2. We evaluate only $select_1$, since the design of $select_0$ is symmetric.

6.2.1 Space Evaluation

We recall from Section 5.2.1 that the ORAM block size of the *select* depends not only on n, K, and k, but also on the distribution of 1s and 0s in the input sequence, because this affects the classification of blocks as sparse or dense. In all the following experiments, the input binary sequence has length $n = 2^{32}$. We analyse the space usage of our approach by considering three distributions of 1s in the input sequence: 50%, 10%, and 1%.[2] For this analysis, we consider two values for K: the base value

[2]We do not consider cases where the occurrences of 1s exceed 50% of the sequence because, as emerges from the following results, all big blocks would be dense and get smaller and smaller, thus inducing a reduction of both the average and maximum block size, and of the *select* time, at the cost of a slight increase in the total space usage due to the larger number of blocks.

Table 3: Oblivious select space: average and maximum ORAM block size, and total space on sequences with length $n = 2^{32}$ bits. * indicates that the size depends on sparse big blocks.

%1s	K	avg block size (bytes)	max block size (bytes)	max total space (MiB)
50%	512	132	159	539
	1024	263	297	531
10%	512	655	2049*	524
	1024	1290	1486	516
1%	512	1187	2049*	96
	1024	2571	4097*	103

$K = \log^2 n = 1024$ and the smaller $K = 512$. We fix $k = (\log\log n)^2 = 25$.

Table 3 reports the results about the average and the maximum sizes of ORAM blocks, along with the maximum total space required by the $select_1$ implementation. Here, as expected, the average block size increases linearly with K (i.e., it doubles when K doubles). Additionally, the average size grows as the number of occurrences of 1s decreases, reflecting the transition to sparser blocks.

Regarding the maximum block size observed in the experiments, we note that when 50% of bits in the sequence are 1s, the maximum block size is 159 bytes for $K = 512$ and 297 bytes for $K = 1024$. Thus, it is close to the average block size (132 bytes and 263 bytes, respectively) and it is determined by big dense blocks. But, when the number of 1s is reduced to 10%, the maximum block size is determined by the sparse blocks (marked with '*' in Table 3) for $K = 512$, but not for $K = 1024$. And, when the number of 1s decreases further to 1%, the maximum block size is only determined by the sparse blocks (for both $K = 512$ and $K = 1024$). We recall that, unlike dense blocks, sparse block sizes depend only on K and n, and not on the bit distribution. For this reason, the sparse block sizes observed at 10% and 1% with $K = 512$ are identical. Consequently, the more 1s in the input binary sequence we have, the greater the gap between the sizes of sparse and dense big blocks.

Let us now move to discuss the total space used to support $select_1$ operations, still referring to Table 3. We observe that increasing K slightly reduces the overall space for sequences with 50% and 10% of 1s, but slightly increases the total space when the proportion of 1s drops to 1%. For sequences with 50% and 10% of 1s, we need between 0.7% and 5.3% additional space compared to simply storing the 2^{32} bits (512 MiB) of the input sequence. In contrast, for sequences containing only 1% of 1s, the total space used is significantly smaller: between 79.9% and 81.4% smaller than the input binary sequence, thus resulting in just 96 MiB and 103 MiB when $K = 512$ and $K = 1024$, respectively. This substantial space saving derives from the fact that, with only 1% of 1s, most big and small blocks are sparse and directly store the positions of the few 1s present in the input. This means that for very sparse input arrays, the space occupancy of the proposed solution is even smaller than plainly storing the input array.

Table 4: Oblivious select time: time to answer a rank operation on a binary sequence of length $n = 2^{32}$ bits.

%1s	K	scan (μs)	POPCOUNT (μs)
50%	512	1.36	0.55
	1024	2.26	0.67
10%	512	5.30	0.84
	1024	10.39	1.30
1%	512	0.23	0.22
	1024	0.25	0.24

6.2.2 Time Evaluation

As expected from Section 5.2.1, the time to answer a $select_1$ operation, shown in Table 4, increases with K. In this evaluation, we compare two implementation strategies: the first one involves a linear scan up to K^2 bits in the dense small blocks, while the second one leverages the `POPCOUNT` instruction to count the number of 0s and 1s within a dense small block.

When we scan the bits, increasing K from 512 to 1024 results in *select* times approximately 1.7× and 1.9× larger, which passes from 1.4 μs and 5.3 μs to 2.3 μs and 10.4 μs for sequences containing 50% and 10% of 1s, respectively. In the case where 1s constitute only 1% of the input sequence, blocks tend to be sparse rather than dense, leading to significantly lower query times, and this means that the time increase is by only a few nanoseconds as K doubles.

A similar trend is observed when using the `POPCOUNT`, although overall times are considerably reduced for sequences with 50% and 10% of 1s, being from 2.5× up to 8.0× smaller than the ones experienced with the scanning. As expected, for highly sparse sequences with 1% of 1s, `POPCOUNT` offers no particular advantage over scanning, since the sparse nature of the blocks allows for direct access to the relevant bit positions.

6.2.3 Integration with Ring ORAM

As done for the *rank*, we estimate the amortized bandwidth overhead as $1 + (L+1)(2Z+S)/A \cdot (1 + Poiss_{cdf}(S,A))$ blocks by considering a Ring ORAM

tree with height $L = \log(2n/A)$, where n is computed as the total space divided by the average block size, and using the same configuration for $Z = 32, A = 46, S = 61$ proposed in (Ren et al., 2015, §5).

The estimated bandwidth overhead ranges between 37 and 26 blocks for $K = 512$, and between 34 and 23 blocks for $K = 1024$, as the density of 1s decreases from 50% to 1%. However, as the number of 1s decreases, the block size increases; thus, the amortized bandwidth overhead also grows with the sparsity of the binary sequence. Specifically, the overhead rises from 4.8 KiB to 56.7 KiB as the symbol occurrences decrease from 50% to 1%.

Although the blocks used for *select* are larger than those for *rank*, the resulting data volumes remain small enough to be transferred quickly over modern networks, typically within a few milliseconds. Therefore the communication costs introduced by integrating our ORAM blocks for *select* within Ring ORAM are affordable in practical scenarios.

6.2.4 Take-Home Message

Based on these experimental results, we observe that the additional $o(n)$ server-side space of the proposed data structure ranges from 0.8% to 5.3% on sequences with at least 10% of 1s. In contrast, for sparse input sequences with only 1% of 1s, the proposed data structure achieves up to 80% space savings compared to storing the plain binary sequence, thus requiring at most 20.1% of the sequence length.

The time to support a *select* operation is on the order of a few microseconds on all the considered distributions of the input sequences, thus negligible compared to the other operations required by Ring ORAM (Ren et al., 2015). Finally, communication time is also negligible due to the small amount of data to exchange in our experiments.

Overall, this makes our oblivious *select* implementation practically relevant.

A final comment on the *select* implementation is in order at this point. The overall space occupancy for the *select* assumes variable-length blocks; whereas, as often recalled in this paper, Ring ORAM requires fixed-length blocks. To estimate the wasted space due to padding, we assume that all blocks are padded to match the length of the longest block reported in Table 3. The number of blocks can be approximated by dividing the total space used by the average block size. Therefore, across all the above considered percentages of 1s and values of K, the estimated wasted space due to padding ranges from 61.1 MiB to 111.1 MiB, except in the case of 10% 1s with $K = 512$. In this latter configuration, the maximum block size is approximately 3.1× larger than the average block size, resulting in a padding overhead of about 1.1 GiB. This means that padding could increase significantly the space occupancy of our oblivious *select* over input binary sequence (of 2^{32} bits, hence 512 MiB) by an additional 222% space. But, as discussed at the end of Section 5.2, we can mitigate this space increase due to padding by compacting multiple dense big blocks into one single ORAM block. This strategy allows us to store information from three dense big blocks within a single ORAM block, reducing the padding overhead from approximately 1.1 GiB to just 22.4 MiB.[3] Therefore, the overhead in the space occupancy would be reduced to 6.8% (for input binary sequence of 512 MiB).

7 DYNAMIC SCENARIOS

In this paper, we focused on oblivious Rank and Select over static binary sequences. However, our approach can naturally extend to dynamic scenarios where bits may flip, while insertions and deletions are excluded, as ORAM typically assumes only data updates.

Many dynamic structures, such as the Log-Structured Merge-tree (LSM-tree) (Luo and Carey, 2020; O'Neil et al., 1996) and RocksDB (Platforms, 2022), rely on periodically reconstructing static data structures to incorporate updates efficiently. Following the same algorithm approaches, oblivious Rank and Select can be supported dynamically by reconstructing ORAM blocks. This reconstruction remains hidden from the server, since neither the number nor the size of the real blocks changes, and thus does not affect the Ring ORAM tree.

This approach is particularly suitable for Rank, as the reconstruction of ORAM blocks preserves both their size and count, given that only bit flips occur and the binary sequence length remains unchanged. On the other hand, reconstructing the ORAM blocks for Select requires special care, as both the number and size of these blocks depend on the sparsity of the target symbol (being it 1 or 0). Nonetheless, we can manage this reconstruction using the following strategy. First, the ORAM block size is fixed to the largest possible value in practice, determined by the size of the sparse big blocks. Next, after compacting dense big blocks into the same ORAM block using the technique described in Section 5.2, we estimate the maximum number of ORAM blocks needed to store all the big blocks by considering the case of a very dense binary sequence, which represents the scenario

[3]Note that this last padding overhead considers a number of ORAM blocks reduced by a factor of 3.

requiring the most server storage. Hence, we can determine both the size and number of ORAM blocks that should be used for building the Ring ORAM tree and this ensures that the reconstruction of our ORAM blocks is completely hidden from the server and does not impact the failure probability of Ring ORAM.

On the client side, even when the Rank is considered in the dynamic setting, the size of the position map (Section 2) remains unchanged, since it depends only on the number of blocks, which, as noted above, is unaffected by bit flips. For the Select, instead, the size of the position map may vary when bits are flipped, because the number of blocks can change since it depends on the density of blocks. However, as noted above, we can estimate the maximum number of ORAM blocks and therefore the maximum size of the position map.

8 APPLICATION

Our interest in oblivious Rank and Select stems from the fact that they enable simple and efficient oblivious implementations of many other sophisticated algorithms and data structures (see (Navarro, 2016) and refs therein). Below, we analyze how Rank and Select support oblivious tree navigation, discussing main advantages and limitations. Then, we sketch a few applications of Rank and Select to compressed oblivious data structures, but omitting pros and cons, since the same considerations of the trees apply.

Trees. A standard way to make tree navigation oblivious is to store the pointer-based tree in an ORAM: each block contains a node plus pointers to its parent and children, and navigation follows these pointers. Although this needs only one ORAM access per step, it has three drawbacks: (i) pointers, typically 64 bits each, make the blocks rather large; (ii) padding is expensive because variable-length blocks (because of variable number of children) must be converted to fixed-size blocks; and (iii) storing one block per node yields many real blocks, resulting in a large ORAM tree and costly accesses.

By contrast, a succinct encoding such as LOUDS (Navarro, 2016) supports navigation using one *rank* and one *select*. With our oblivious *rank*/*select* design, each navigation step requires up to two ORAM accesses. However, padding is negligible in practice (Section 6); pointers are eliminated, since the target block is computed arithmetically; and nodes information are tightly packed so as to reduce the number of ORAM accesses, by enabling multiple navigation steps from information stored in the same ORAM block, as well as the total number of ORAM blocks, this the resulting ORAM tree size.

Predecessor and Successor Searches. Assume an array A composed of n sorted numbers $x_i < x_{i+1}$ for each $i \in [0, n-2]$. We are interested in supporting two operations, i.e., *predecessor* and *successor*, over the array A. The predecessor operation, given a number y, returns the position i such that $x_i \leq y < x_{i+1}$; while the successor operation, given a number y, returns the position i such that $x_{i-1} < y \leq x_i$. We can leverage the approach described in (Navarro, 2016, §4.5.2), which encodes the numbers of A into a binary vector B by setting the bits $B[x_i] = 1$, for $i \in [0, n-1]$; and then use our oblivious *rank* and *select* to support oblivious predecessor and successor operations in exactly two Ring ORAM access.[4]

Google's Sparse Hash. Google's sparse hash (Google, 2007) stores a hash table T by avoiding the usage of empty cells via an array H, which stores only the non-empty cells of T, and a binary vector B, which indicates which cells of T are empty or not. This approach clearly saves significant space because of the removal from T of the empty cells, at the cost of one bit per cell (empty or not) of T. Still, this storage enables fast access to T: in fact, retrieving the cell $T[i]$ requires only one $rank(i)$ operation over B and then an access to that position in H.

We can easily make this solution oblivious by replacing the standard *rank* with an oblivious *rank* over the array B, and storing the array H in a separate Ring ORAM. This way, each query requires only two Ring ORAM accesses: one to perform the oblivious *rank* and one to access the hash table content.

9 CONCLUSIONS

This work addressed the challenge of efficiently supporting sophisticated queries over data stored in Oblivious RAM (ORAM), in a way that memory access patterns are hidden from untrusted or potentially honest-but-curious servers. While naive integration of standard data structures into ORAM is conceptually straightforward, it incurs high time and communication overheads due to the multiple and costly accesses to ORAM. We tackle this by focusing on two

[4]Given y, $predecessor(y) = select_1(rank_1(y))$ while $successor(y) = select_1(rank_1(y)+1)$. If $y = x_i \in A$, we have to perform anyway the *select* operation to avoid leakage of such information, even if we discover that x_i is predecessor/successor of y after the *rank* operation.

fundamental primitives, Rank and Select, which have proven pivotal in the design of a wide range of sophisticated (non-oblivious) data structures (including hash tables, trees, and graphs), and present their first ORAM-aware implementation.

Our design extends block-based strategies, used in non-oblivious settings, by introducing carefully structured and succinct metadata. This ensures each Rank/Select operation requires just a single ORAM access, uses $O(1)$ additional client-side space, and achieves space efficiency comparable to non-oblivious counterparts, thus minimizing the time required for each ORAM access. Furthermore, experimental results show negligible in-block search times (at most $1.3\mu s$ on 512 MiB sequences), confirming the practical efficiency of our design scheme.

Finally, our oblivious Rank and Select implementations enable compressed oblivious data structures with lower server storage and padding overhead, and with faster ORAM accesses (smaller communication, encryption/decryption, and reshuffling costs) than pointer-based alternatives. Moreover, the simplicity and generality of our approach pave the way for a software library of oblivious data structures, making oblivious-by-design applications more accessible to software engineers.

ACKNOWLEDGEMENTS

The work of Paolo Ferragina was partially supported by the Alfred P. Sloan Foundation with the grant #G-2025-25193 (sloan.org).

REFERENCES

Dautrich, J., Stefanov, E., and Shi, E. (2014). Burst {ORAM}: Minimizing {ORAM} response times for bursty access patterns. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 749–764.

Devadas, S., Van Dijk, M., Fletcher, C. W., Ren, L., Shi, E., and Wichs, D. (2015). Onion oram: A constant bandwidth blowup oblivious ram. In *Theory of Cryptography Conference*, pages 145–174.

Ferragina, P. (2023). *Pearls of Algorithm Engineering*. Cambridge University Press.

González, R., Grabowski, S., Mäkinen, V., and Navarro, G. (2005). Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38.

Google (2007). Google sparsehash library.

Keller, M. and Scholl, P. (2014). Efficient, oblivious data structures for mpc. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 506–525.

Kushilevitz, E., Lu, S., and Ostrovsky, R. (2012). On the (in) security of hash-based oblivious ram and a new balancing scheme. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 143–156.

Luo, C. and Carey, M. J. (2020). Lsm-based storage techniques: a survey. *The VLDB Journal*, 29(1):393–418.

Mäkinen, V. and Navarro, G. (2007). Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347.

Navarro, G. (2016). *Compact data structures: A practical approach*. Cambridge University Press.

Navarro, G. and Providel, E. (2012). Fast, small, simple rank/select on bitmaps. In *International Symposium on Experimental Algorithms*, pages 295–306.

O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E. (1996). The log-structured merge-tree (lsm-tree). *Acta informatica*, 33(4):351–385.

Patel, S., Persiano, G., Raykova, M., and Yeo, K. (2018). Panorama: Oblivious ram with logarithmic overhead. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 871–882.

Platforms, M. (2022). Rocksdb.

Ren, L., Fletcher, C., Kwon, A., Stefanov, E., Shi, E., Van Dijk, M., and Devadas, S. (2015). Constants count: Practical improvements to oblivious ram. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 415–430.

Sadakane, K. (2025). Secure Compressed Suffix Arrays. In *From Strings to Graphs, and Back Again: A Festschrift for Roberto Grossi's 60th Birthday*, volume 132, pages 13:1–13:8.

Stefanov, E., Dijk, M. v., Shi, E., Chan, T.-H. H., Fletcher, C., Ren, L., Yu, X., and Devadas, S. (2018). Path oram: an extremely simple oblivious ram protocol. *Journal of the ACM (JACM)*, 65(4):1–26.

Wang, X. S., Nayak, K., Liu, C., Chan, T. H., Shi, E., Stefanov, E., and Huang, Y. (2014). Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 215–226.

Wikipedia (2025). Hamming weight.