Oblivious Complex Queries on Variable-Length Strings

Mariagiovanna Rotundo[1][a], Giuseppe Persiano[2][b] and Paolo Ferragina[1,3][c]
[1]*Department of Computer Science, University of Pisa, Pisa, Italy*
[2]*Dipartimento di Scienze Aziendali - Management and Innovation Systems, University of Salerno, Salerno, Italy*
[3]*Department of L'EMbeDS, Sant'Anna School of Advanced Studies, Pisa, Italy*

Keywords: Oblivious RAM, Variable-Length Data, Complex Queries.

Abstract: In this paper, we study the problem of storing and searching in datasets of variable-length strings, a core primitive in key-value stores, (graph) DBs, and search engines. However, enabling such search capabilities in ORAM scenarios, where data are stored on an honest-but-curious server, remains challenging. We address this problem by proposing a practical design that combines Ring ORAM (Ren et al., 2015) to hide access patterns to outsourced data, with a Patricia trie (Ferragina and Grossi, 1999; Ferragina et al., 2025) for space-efficient search over variable-length strings. The resulting scheme supports search over variable-length string datasets in an ORAM scenario, while retaining efficient storage and access both on the client and the server. We evaluated our scheme on datasets having size up to 273 GB, showing that it supports complex string queries, with only 2 Ring ORAM accesses on the server, incurring a client-server communication cost below 3 MiB, a client memory footprint of at most 200 MB, and negligible client computation time per query. Although we assume bounded-length strings, the bound is high enough to handle most practical use cases.

1 INTRODUCTION

Cloud outsourcing has become the default for storing and processing large-scale data. Still, it also raises a central privacy challenge: even when data are encrypted, the server can often infer sensitive information from the client's access patterns. This issue is particularly relevant in domains such as healthcare, justice and public administration, where both the dataset and the query sequence may reveal highly sensitive attributes (Roche et al., 2016; Liu et al., 2018).

Several cryptographic primitives address privacy in outsourced settings. Oblivious Transfer (OT) addresses the scenario in which a trusted server stores and protects data, typically encrypted, from untrusted clients, who can only access portions of data while hiding their queries and results from the server. PIR and ORAM address settings where clients want to hide queries and results from the server storing the data. PIR typically assumes the data are public and available in the clear to the server. ORAM, instead, considers a stricter setting where the data belong to the client and are stored encrypted on an untrusted honest-but-curious server, which tries to infer information about the data, queries, and results by observing the memory access pattern.

In this paper, we consider the ORAM setting, aiming for space and bandwidth-efficient data management while hiding from the server the memory access pattern through cryptographic techniques and continuous reshuffling. At the same time, we go beyond merely supporting oblivious reads and writes: our goal is to enable efficient and sophisticated search functionality directly over the outsourced, encrypted, variable-length string dictionary (e.g., membership, prefix, and lexicographic queries), while protecting the access pattern and hiding dictionary content, query, and results.

1.1 Related Works

First of all, we want to efficiently store variable-length data on an honest-but-curious server, and obfuscate its content and the memory access pattern.

Although many ORAM protocols have been proposed for storing data hiding the memory access patterns, most of them, including Path ORAM and Ring ORAM, assume fixed-size data items. A straightfor-

[a] https://orcid.org/0009-0001-1671-7407
[b] https://orcid.org/0000-0001-6579-4807
[c] https://orcid.org/0000-0003-1353-360X

Rotundo, M., Persiano, G. and Ferragina, P.
Oblivious Complex Queries on Variable-Length Strings.
DOI: 10.5220/0015001100004103

In *Proceedings of the 23rd International Conference on Security and Cryptography (SECRYPT 2026) - Volume 1*, pages 971-983
ISBN: 978-989-758-858-7; ISSN: 2184-7711

ward scheme for storing variable-length data is to pad all elements to the longest item in the dataset, thereby possibly wasting a significant amount of space. To the best of our knowledge, we are aware of only three ORAM schemes that specifically target the storage of variable-length strings, reducing padding overhead and minimizing wasted space while still preserving access-pattern privacy: DivORAM, vORAM, and weighted ORAM (wORAM).

DivORAM. DivORAM (Liu et al., 2018) stores variable-length data by splitting each element into fixed-size pieces, called *splinters*, and placing all splinters of the same element along a single root-to-leaf path. The ORAM tree uses non-uniform bucket sizes, where each bucket is twice the size of its child buckets. An access retrieves all splinters on the path with a single ORAM operation and reconstructs the element locally.

DivORAM reduces client bandwidth by introducing a *trusted proxy* that performs most interactions with the server (including eviction, reshuffling, and re-encryption). However, proxy-server communication affects the end-to-end access time, while the original analysis accounts only for client-proxy traffic when reporting bandwidth overhead. Moreover, to avoid overflow, the DivORAM algorithm writes back the stash whenever it exceeds $\log n$ blocks (Assouline and Minaud, 2023), which may leak information because some elements fill the stash faster than others.

vORAM. The vORAM (Roche et al., 2016) approach is based on Path ORAM but stores variable-length blocks in fixed-length buckets. A block can be split into multiple variable-length pieces and stored in more buckets along the same root-to-leaf path, and to reconstruct an element, vORAM retrieves from the same path all its variable-length pieces. vORAM relies on choosing bucket sizes based on an assumed distribution of element lengths, and its efficiency is demonstrated mainly for geometric distributions (Roche et al., 2016), rather than as a general-purpose solution.

wORAM. Among existing proposals, weighted ORAM (wORAM) (Assouline and Minaud, 2023) is a state-of-the-art general-purpose ORAM designed to handle variable-length data. wORAM assumes that each element has a *weight* (its length) bounded by an upper bound B, and stores each element in a single variable-length block whose size matches the element. Blocks are then stored in fixed-size buckets.

To locate elements, wORAM uses a position map with one fixed-size entry per element/block. This can make the position map up to B times larger than in fixed-block ORAMs: with fixed-size blocks, the number of blocks is roughly $n \approx N/B$, where N is the total data size, whereas with variable-length blocks the number of blocks can be much larger, and potentially as large as N.

By design, wORAM inherits the communication cost of the underlying fixed-block ORAM it instantiates. For example, when built on Path ORAM, each access reads and writes a full root-to-leaf path. Since wORAM hides lengths by retrieving and reshuffling entire paths, it cannot directly leverage optimizations from state-of-the-art fixed-block ORAMs such as Ring ORAM, whose techniques rely on uniform block sizes. Finally, variable-length blocks also complicate reshuffling/eviction, because the protocol must ensure that the total length of blocks stored in each bucket never exceeds its capacity, increasing algorithmic complexity.

If we aim to support privacy-preserving searches, then there exist Symmetric Searchable Encryption (SSE) schemes that obfuscate memory access patterns.

Privacy-Preserving SSE. In (Garg et al., 2016), an SSE based on Path ORAM and TWORAM protocols have been proposed. We can use this ORAM-based solution to, given a keyword k, return the list of files containing k.

For each keyword k, this approach stores in TWORAM (i) k, (ii) the number of files containing it, and (iii) the number of accesses to the corresponding list of files, while storing in Path ORAM one block for each file containing k (i.e., each file is replicated once per contained keyword).

The information stored in TWORAM is used to locate in Path ORAM the blocks with the files containing k, and each file is then retrieved via a Path ORAM access. Since, given a keyword k, the number of Path ORAM accesses depends on the number of files containing k, the scheme incurs *volume leakage*. Note that both TWORAM and Path ORAM store fixed-length data. Here, only the lists of files have variable length.

1.2 Our Contribution

In this paper, we show (i) how to efficiently store bounded variable-length strings by leveraging state-of-the-art ORAM protocols for fixed-size blocks, outperforming existing ORAM protocols tailored to variable-length data; and (ii) how to support complex queries over these strings while keeping the client memory footprint small. Specifically, we combine

Ring ORAM (Ren et al., 2015) for secure storage with a Patricia trie (Ferragina and Grossi, 1999; Ferragina et al., 2025) for space-efficient indexing, obtaining a secure scheme that supports not only point accesses but also richer queries such as lexicographic searches over strings stored on an honest-but-curious server.

Our design preserves the theoretical guarantees of the underlying components and is validated on the URLs (273 GB, 4 billion strings) and Filenames (70 GB, 2 billion strings) dictionaries from (Ferragina et al., 2025), which are the largest datasets currently used to evaluate indexing solutions. In our evaluation, lexicographic queries require only 2 Ring ORAM accesses, with a position map and Patricia trie of at most 60 MB and 140 MB, negligible traversal/scanning time, and at most 3 MiB client-server communication cost per query, improving over prior variable-length solutions.

The clear separation between string storage and indexing makes the scheme flexible: Ring ORAM and the Patricia trie can be replaced independently as better components emerge. While we focus on static dictionaries of bounded-length strings, this restriction is non-limiting in practice, since the scheme supports strings up to 32 KiB, a common scenario in key-value stores, search engines, and bioinformatics workloads (Khan et al., 2022; Pibiri, 2023; Pibiri et al., 2023; Fan et al., 2024). With additional engineering, the same architecture can be extended to dynamic dictionaries through dynamic succinct trie representations, periodic rebuilding, or the allocation of additional space on the server. Exploring these extensions and evaluating their practical trade-offs are left for future work.

2 BACKGROUND

In this section, we review the main schemes and solutions used in the rest of the paper.

2.1 Ring ORAM

Among existing ORAM protocols, Ring ORAM (Ren et al., 2015) is among the most efficient constructions for fixed-size blocks. Ring ORAM on n blocks of data of size B organizes the server storage as a perfect binary tree of height L with $2^{L+1}-1$ nodes, called *buckets*. Each bucket contains up to Z *real* blocks (i.e., blocks of data) and S *dummy* blocks (i.e., with dummy content), all of fixed size B. If fewer than Z real blocks are allocated to a cluster, then more dummy blocks are added so to reach a total of $Z+S$ blocks. The $Z+S$ blocks are randomly permuted and the bucket contains some extra information, called the *metadata*, that keeps track of which of the blocks are real and of its index and which are dummy. Each real block is mapped uniformly at random to a leaf and is stored in a cluster found in the path from the root to the leaf assigned to it. The client storage includes the *position map* that keeps track of the block to leaf assignment and a *stash* for blocks read from the server but not yet written back.

Ring ORAM improves bandwidth overhead by avoiding full-path downloads: to access a block mapped to a given root-to-leaf path, the client retrieves only one block per bucket (the target block in its bucket and a dummy block in all other buckets along the path). When dummy blocks have known content, bandwidth overhead can be further reduced by asking the server to return only the XOR of the requested blocks (Dautrich et al., 2014). Finally, reshuffling is performed periodically after A accesses, where A is a fixed protocol parameter, on a path selected deterministically.

Based on (Ren et al., 2015) and assuming $A \leq S$, the communication and encryption costs of Ring ORAM over every A access operations are summarised in the following two lemmas.

Lemma 1 (Download/upload costs)**.** *During every A Ring ORAM access operations, the client downloads $(A+1)(L+1)$ and uploads $L+1$ metadata blocks (Lemma 3 below). In terms of data blocks, the client downloads $A+Z(L+1)$ and uploads $(Z+S)(L+1)$ blocks, each of size B.*

Lemma 2 (Encryption/decription costs)**.** *During every A Ring ORAM access operations, the client decrypts $(A+1)(L+1)$ and encrypts $L+1$ metadata blocks. In terms of data blocks, the client decrypts up to $A+Z(L+1)$ and encrypts up to $AL+(Z+S)(L+1)$ blocks.*

All metadata blocks share the same size (Ren et al., 2015), which depends on Z, S, n (we recall that $L=O(\log n)$), and λ, which is the size of the encryption seed of the bucket:

Lemma 3. *Each metadata block has size $\log S+(Z+S)+Z\log n+ZL+Z\log(Z+S)+\lambda$ bits.*

When a block is read and rewritten on the server, its encryption must change to prevent the server from recognising it. This is typically done using randomised encryption, for which several schemes exist in the literature. For this work, we consider the Advanced Encryption Standard (AES) with 128 bits in Cipher Block Chaining (CBC) mode, which is slower than other modes, so it can be used for estimating the worst-case performance of AES-128.

2.2 Patricia Tries

We focus on *complex queries* on strings, such as lexicographic queries that return the rank of a query string within a sorted dictionary of variable-length strings. Notice that membership queries, which check whether a given string occurs in the indexed dictionary, or prefix queries, which check whether a given string occurs as a prefix of dictionary strings, can be easily derived from the previous one (Navarro, 2016; Ferragina et al., 2025) without any slowdown. These operations are common in search engines, databases, and key-value stores.

Hashing solutions cannot be applied to these queries because strings are not searched exactly. A classical alternative is comparison-based binary search over an array of string pointers, which, on a dataset of size n, looks at $O(\log n)$ strings. A more efficient family of data structures for handling string searches is based on tries (Navarro, 2016), in which each string is stored as a path from the root to a leaf, and string search has an asymptotic cost proportional to its length. Over the years, many variants and succinct representations have been proposed to improve space and query efficiency (Jacobson, 1989; Grossi and Ottaviano, 2015; Navarro, 2016; Zhang et al., 2020; Ferragina et al., 2025). In this work, we rely on the Patricia trie (PT), which keeps in the trie structure as little information as possible and stores the indexed strings in a slower memory level.

A Patricia trie (PT) (Morrison, 1968) is obtained from the compacted trie by storing, for each node, the length (in characters) of the root-to-node path, and for each edge only its first character. Its space depends on the number of strings n rather than on their total length, and is $O(n \log l)$ bits, where l is the maximum root-to-node path length stored in the nodes (upper bounded by the maximum LCP between consecutive sorted strings) (Ferragina et al., 2025).

Its search algorithm, called *blind search* (Ferragina and Grossi, 1999; Ferragina et al., 2025), works in three phases. First, a downward traversal locates a leaf l whose pointed string s is the one in the indexed dictionary sharing the longest common prefix (LCP) with the query q. Then, s is retrieved from memory and compared with q to determine their LCP. Finally, an upward traversal locates the edge/node used to identify the lexicographic position of q among all the indexed strings.

Lemma 4. *If the Patricia trie is stored in the client's internal memory and the strings are stored in another memory level (e.g. disk), assuming each indexed string fits one disk page, this search procedure requires the comparison of at most $|q|$ characters on the Patricia Trie and at most 2 random I/Os.*

2.3 Strings Compression

To reduce the space required to store strings, one can use several compression schemes. A simple approach is *rear coding* (Martínez-Prieto et al., 2016; Ferragina et al., 2025): strings are sorted, the longest common prefix (LCP) between each string and its predecessor is computed, and each string is encoded as the number of characters to remove from the previous string to obtain the LCP plus the remaining suffix. A drawback is that decompression depends on previous strings, and in the worst case, all strings must be scanned to reconstruct a target string, making this representation expensive for random access. A common mitigation is to apply the technique to blocks of strings, so decompression requires scanning only within a block.

If random access to compressed strings is required, a straightforward alternative is a statistical code such as *Huffman coding* (Navarro, 2016). Huffman coding builds a model from symbol frequencies and assigns shorter codewords to more frequent symbols and longer ones to rarer symbols. The resulting codewords are prefix-free, i.e. each codeword is not a prefix of any other generated codeword, enabling each string to be compressed and decompressed independently by replacing each character with its codeword and vice versa. It is well-known that the Huffman code is optimal among all prefix-free codes that assign distinct codewords to individual characters.

3 OUR SCHEME

As anticipated in the Introduction, we consider a client with limited memory that wishes to support complex queries over a large string dictionary outsourced to an untrusted server with unbounded storage. The goal is to prevent the server (and any eavesdropper on the communication channel) from learning information about the dictionary contents, the issued queries, and the results.

To this end, we propose a scheme that combines Ring ORAM for privacy-preserving storage with the two-level indexing approach for large string dictionaries based on the Patricia trie introduced in (Ferragina et al., 2025).

Table 1 summarizes the notation used throughout this presentation.

Ring ORAM Blocks Construction. First of all, we focus on how to store the strings in fixed-size blocks with minimum padding (Section 1.1).

Table 1: Main notation used in the proposed scheme.

Symbol	Definition
d	number of strings in the dictionary
D	total length of all strings in the dictionary
M	length of the longest string in the dictionary
B	size of a block of compressed strings
m	number of blocks storing the compressed strings
n	number of real blocks stored in Ring ORAM
L	height of the Ring ORAM tree
Z	number of real blocks per Ring ORAM bucket
S	number of dummy blocks per Ring ORAM bucket
A	Ring ORAM path-reshuffle frequency
λ	security parameter, i.e., encryption-seed size

We build the blocks following the storage construct introduced in (Ferragina et al., 2025). Given d strings of total length D and maximum length M, we lexicographically sort them and compress them via Rear coding. We then store the compressed strings into m fixed-size blocks of size $B \geq M$[1], filling each block with as many strings as possible. Since different blocks may contain different numbers of strings, the client additionally stores an auxiliary structure of size $O(m \log d)$ bits (Ferragina et al., 2025) that allows computing, for each block, how many strings are stored in all preceding blocks.

We will link this parameter space with the proposed oblivious searching scheme in Section 4.

Then, assuming n real blocks with *block size* B contain our input dictionary, we define the client and the server data structures.

In Figure 1, we show a graphical representation of our scheme on the string dictionary is $\{$*hello, hells, her, herbology, name, no, noble, nolo*$\}$.

Server's Data Structures. These blocks are stored on the server through a Ring ORAM tree of height L, eviction frequency A, *buckets* consisting of $Z+S$ blocks (of size B each), where Z and S are, respectively, the number of blocks reserved for real and dummy content, and encryption seeds of size λ.

Because of the XOR trick, we need to know the content of the dummy blocks, so we assume all of them are filled with the same content (but their encryptions differ because of the random encryption).

[1]So that each string is stored in exactly one block

Recall from the Section 2 that $Z+S$ blocks, plus some (auxiliary) metadata, form a bucket of the Ring ORAM. To reduce metadata size, we slightly change the stored metadata defined in (Ren et al., 2015). We replace the pointers that for each real block tell us its position inside the bucket with *checking bits*, one per block (real and dummy) in the bucket, assuming that the ids of the real blocks are stored following the order in which they appear in the bucket. These checking bits indicate, for each block, if the block is real or not. With this scheme, we can locate the block with id x by scanning ids and checking bits: if x is the i^{th} id of the bucket, its position is the one of the i^{th} set checking bit. Furthermore, the block mapping information (called *leaves* in (Ren et al., 2015)) is not stored in the metadata, as it is already maintained in the client-side position map. Because of this change, we consider metadata blocks (whose size depends on Z, S, and n) with the following size, which differs from the one seen in Lemma 3:

Lemma 5. *Each metadata block has size* $\log S + 2(Z+S) + Z\log n + \lambda$ *bits.*

From the discussion above, it immediately follows the space needed to store n real blocks with their metadata in the server's Ring ORAM. In the following Lemma 6, we indicate the server space.

Lemma 6. *The space used on the server is* $(2^{L+1}-1)(B(Z+S)+\log S+2(Z+S)+Z\log n+\lambda)$ *bits.*

Client's Data Structures. On the client, we keep the stash whose size is $O(\log n)$ blocks, the position map of nL bits, where L is the height of the Ring ORAM tree, and a succinct Patricia trie (Ferragina et al., 2025) of size $\Theta(n)$, built on the first string of each real block. Assuming the Ring ORAM tree has height L, and eviction frequency A, the client's data structures need the following space.

Lemma 7. *The space used on the client is given by three data structures: the stash of size* $O(\log n)$ *blocks, the position map of size nL bits, and the Patricia trie, of size* $O(n \log l)$ *bits, where l is the maximal length of its edges*

Different from (Ferragina et al., 2025), we remove the *leaves* array and implicitly encode leaf identifiers: block ids $0,\ldots,n-1$ are assigned to leaves in the same order in which leaves appear in the LOUDS representation. Therefore, given a LOUDS position x, the corresponding leaf id (and position-map index) is obtained in $O(1)$ time via rank queries, i.e., $rank_0(x) - rank_{10}(x) - 1$.

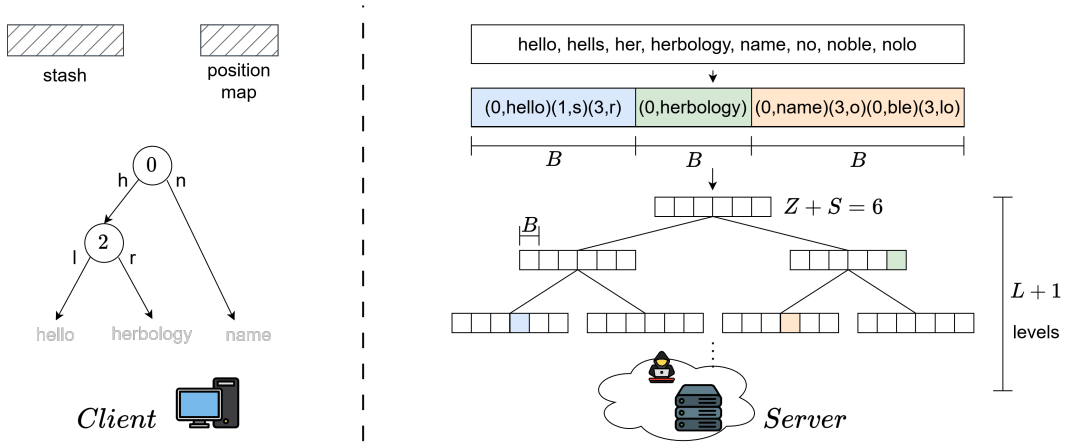

Figure 1: Graphical illustration of our privacy-preserving indexing scheme. Input strings are compressed via Rear coding and divided into (three coloured) blocks of fixed size B stored in a Ring ORAM on the server. Client keeps, besides the stash and position map, a Patricia trie that indexes the first string of each block (three strings in the example). The first strings (reported in grey just for illustrative purposes) are not fully stored in the client, which keeps only a constant-sized part of them.

3.1 Performing Complex Queries

We recall that we are interested in lexicographic queries, that return the rank of a query string q within a sorted dictionary of variable-length strings. Also, recall that membership queries, which check whether a given string occurs in the indexed dictionary, or prefix queries, which check whether a given string occurs as a prefix of dictionary strings, can be easily derived from the previous one (Navarro, 2016; Ferragina et al., 2025) without any slowdown. In fact, with the scheme just described, these queries are answered with exactly two Ring ORAM accesses. Let us see how.

First, the client downward traverses the Patricia trie to identify a candidate block b_1 and retrieves b_1 via one Ring ORAM access. After decrypting b_1, the client extracts its first string s and compares s with q to compute their LCP and determine the lexicographic position of q among the blocks indexed by the Patricia trie (Ferragina et al., 2025). Then, a second Ring ORAM access is always performed. If the analysis indicates that q lies in b_1, the client issues a dummy access to a uniformly random block; otherwise, it retrieves the block that should contain q. Finally, the client scans the correct block to locate q and determine its position.

The overall costs of these steps are summarised in the lemma below. Here, Z and S are the real and dummy blocks in each bucket, A is the frequency of path reshuffle, and L is the height of the ORAM tree.

Lemma 8. *Performing a lexicographic query for an input string q requires $O(|q|)$ character comparisons and 2 ORAM accesses. The amortised number of blocks downloaded and uploaded by the client is $A+Z(L+1)$ and $(Z+S)(L+1)$, respectively, while the amortised number of decrypted and encrypted blocks per ORAM access is $A+Z(L+1)$ and $AL+(Z+S)(L+1)$, respectively.*

Data and Operation Leakage. In the scheme above, strings are partitioned into fixed-size blocks, which are then stored on an untrusted server using Ring ORAM. Block construction is independent of the ORAM protocol; therefore, given a collection of n blocks, we can store them by exploiting Ring ORAM unchanged.

Each lexicographic query triggers a Patricia-trie traversal on the client and exactly two Ring ORAM accesses. Since the Patricia trie is maintained locally on the client, its traversal leaks no information to the server and is only used to identify the blocks to be retrieved via Ring ORAM. Because every query induces the same number of ORAM accesses, the scheme does not incur volume leakage: queries are indistinguishable in terms of their number of ORAM accesses. In particular, the server may only learn that each operation consists of two ORAM accesses but cannot distinguish any two sequences of l operations.

Because Ring ORAM is executed unchanged, performing two ORAM accesses per query does not affect its security and failure probability: the analysis in (Ren et al., 2015, §3.5, §4) continues to apply. Therefore we have proved the following:

Lemma 9. *The scheme supports lexicographic, prefix, and membership queries with no data and operation leakage: every sequence of l queries induces exactly $2l$ ORAM accesses, making any two such se-*

quences indistinguishable with respect to access volume.

3.2 Support Oblivious String Updates

Updating strings may either modify individual characters or change their length by inserting/removing symbols. Character substitutions are relatively simple, since they preserve the overall dataset size and therefore do not affect the failure probability of the underlying ORAM. Length-changing updates are substantially harder, typically requiring additional constraints. Following wORAM (Assouline and Minaud, 2023), one can assume a maximum total capacity N and allow strings to grow as long as the dataset remains within this bound. However, setting N close to the current data size D leaves little room for growth, whereas choosing $N \gg D$ wastes space and increases communication and encryption costs. As a result, dynamically resizing strings in ORAM remains challenging to support efficiently and flexibly. Supporting complex queries (e.g., lexicographic searches) over large dynamic dictionaries poses similar difficulties: efficient methods rely on maintaining sorted order, but updates may change a string's lexicographic position, triggering relocations and potentially requiring partial reconstruction of indexing data structures (Navarro, 2016).

Our architecture could be extended to dynamic datasets by combining extra allocated space within blocks with selective updates of both the blocks and the Patricia trie. For the Patricia trie, a dynamic succinct representation, such as DFUDS (Navarro, 2016), can be adopted to enable local updates without full rebuilding, while blocks could keep a small fraction of free space to absorb moderate string growth. When an update changes a string's lexicographic position, the affected block can be rebuilt before being written back, and the updated string is temporarily buffered on the client until it is reinserted into the correct block. This requires an additional client-side structure to track pending reinsertion. These ideas outline a possible path toward a dynamic version of our system, although turning them into a practical design would require careful engineering and evaluation. Finally, although we focus on static string dictionaries, these considerations remain relevant, as several oblivious solutions in the literature are also designed for static scenarios. For instance, oblivious key-value stores such as 3H-GCT (Garimella et al., 2021) and RB-OKVS (Bienstock et al., 2023) have constructions that depend heavily on the input set, so updates require a full reconstruction of the oblivious key–value store.

Table 2: Values of Z, A, and S (depending on Z) used for the performance evaluation.

Z	**4**	**8**	**10**	**16**	**32**	**33**
A	3	8	11	20	46	48
S	6	14	18	29	61	63

4 PERFORMANCE EVALUATION

In this section, we analytically evaluate the practical efficiency of our scheme along four main performance factors: (i) storage overhead on the server and, most importantly, on the resource-constrained client; (ii) client-server communication cost per query; (iii) encryption/decryption work induced by each query; and (iv) the time required to traverse the Patricia trie. We then compare these costs against wORAM, which is the current state-of-the-art general-purpose ORAM for variable-length data (see Section 1.1). We omit TWORAM-based SSE constructions as they address a different problem setting, DivORAM due to its leakage, and vORAM because its efficiency relies on specific assumptions on the string length distribution.

We instantiate our analysis on the two large real-world datasets introduced in (Ferragina et al., 2025). The *URLs* dataset contains 3.7 billion URLs collected from multiple crawls (Boldi et al., 2018), for a total of 272.7 GB and a maximum length of 2083 characters. The *Filenames* dataset contains 2.3 billion file names, for 68.9 GB, with a maximum length of 16051 characters. Due to their nature, URLs are typically longer and exhibit more repetition than file names.

4.1 Performance of Our Scheme

We evaluate our searching scheme on the two datasets introduced above and study how performance varies with the Ring ORAM parameter Z (with A and S set as a function of Z as in (Ren et al., 2015, §5); see Table 2). In all experiments, we consider block sizes B ranging from 512 bytes to 256 KiB, $\lambda = 128$ bits, and we set the Ring ORAM height to $L = \log\left(\frac{2n}{A}\right)$, as in (Ren et al., 2015), where n is the number of real blocks.

Let d be the number of strings and D their total length, the number of real blocks $n = O(D/B)$ can be much smaller than d, depending on the string lengths and on the effectiveness of Rear coding, whose practical efficiency is evaluated in (Ferragina et al., 2025).

4.1.1 Server-Side Space Occupancy

To estimate server space, we first compute how many real blocks of size B are needed to store the compressed dictionaries. Because the string's length is bounded by B, we also evaluate our ability to correctly store them by counting how many strings are longer than B, and thus truncated when stored in blocks. Table 3 reports the resulting number of real blocks n (in millions) and the number of truncated strings. We observe that 194 million and 77 million blocks suffice to store, respectively, 3.7 billion URLs and 2.3 billion filenames. As expected, doubling B approximately halves n, and truncation quickly drops to (almost) zero: in our datasets, $B = 4$ KiB avoids truncating any URL and truncates only 5 filenames, while $B = 16$ KiB prevents truncation in both datasets.

Using Lemma 6 together with Table 2 and the values of n from Table 3, we derive that the resulting Ring ORAM storage is about $2.5\times$ (URLs) and $5\times$ (Filenames) larger than the raw dictionaries for $Z = 16$, and grows to about $5\times$ (URLs) and $10\times$ (Filenames) for $Z = 8$. This overhead is typical in ORAM settings, where additional server-side storage enables access-pattern obfuscation.

4.1.2 Client-Side Space Occupancy

As indicated in Lemma 7, the client stores a stash, a position map, and a Patricia trie. The stash is negligible in our setting, and we therefore focus on the other two components.

Using the parameters in Table 2, the position-map size is $nL = n\log\left(\frac{2n}{A}\right)$ and decreases when B doubles and Z increases, since n approximately halves while A increases. This behaviour is reflected in Figure 2, which reports the exact position-map size (in MiB) for varying B and Z. Overall, the position map requires at most 624 MiB (0.24% of the dataset size) for URLs and 238 MiB (0.36%) for Filenames. For a practical choice of $B = 4$ KiB (disk-page size), it drops to 58 MiB (0.02%) and 25 MiB (0.04%), respectively.

We now consider the Patricia trie. Its size depends on the number of indexed strings, i.e., on the number n of real blocks, and therefore scales roughly linearly with n (see Table 3). In our experiments, the Patricia trie requires at most 1.3 GB (URLs) and 474 MB (Filenames) when indexing the first strings of about 195 and 80 million blocks. This decreases to 134 MB and 55 MB for $B = 4$ KiB, and to 2 MB and 1 MB for $B = 256$ KiB. Compared to (Ferragina et al., 2025), our optimized Patricia trie is $27\% - 35\%$ smaller thanks to removing the explicit *leaves* array.

In summary, for $B \geq 4$ KiB the overall client footprint of our indexing scheme (position map plus Patricia trie, with negligible stash) is at most 194 MB for URLs and 81 MB for Filenames, i.e., only 0.05% and 0.08% of the respective dataset sizes.

Table 3: The table reports for the two datasets (URLs and Filename) the number n of compressed (and encrypted) blocks stored in the Ring ORAM (in millions); the number of strings with length larger than B, and thus truncated to their B-long prefix; and the size in MBs of the Patricia trie (PT) to be stored in the client.

URLs			
B (KiB)	***n* (M)**	**truncated strings**	**PT size (MB)**
0.5	193.68	7.1 M	1277.87
1	88.29	1.3 M	586.61
2	41.61	7.6 K	277.92
4	20.16	0	133.08
8	9.93	0	65.43
16	4.93	0	32.35
32	2.45	0	16.06
64	1.22	0	7.94
128	0.61	0	3.96
256	0.30	0	1.96
Filenames			
B (KiB)	***n* (M)**	**truncated strings**	**PT size (MB)**
0.5	76.52	1.7 K	473.27
1	36.57	77	225.47
2	17.89	16	109.55
4	8.85	5	54.69
8	4.40	2	26.49
16	2.19	0	13.13
32	1.09	0	6.63
64	0.54	0	3.34
128	0.27	0	1.65
256	0.13	0	0.82

4.1.3 Client-Server Communication Cost

Because we want to be able to correctly store and manage most of the strings in the input dictionaries, we will restrict our attention only to the case of $B \geq 4$ KiB. Since early reshuffling in Ring ORAM strictly depends on the searched queries, we do not include its cost in the following evaluation, and highlight to the reader that it possibly increases communication cost by up to 37% (derived from the discussion in (Ren et al., 2015, §5)).

Knowing the number of real/dummy blocks (of size B) and metadata blocks (whose size is indicated in Lemma 3) exchanged every A accesses (Lemma 1), we can estimate the amortised communication cost

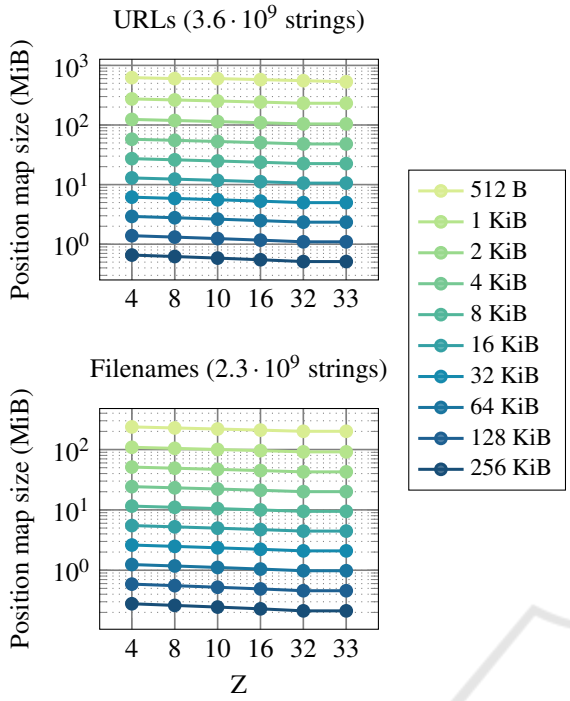

Figure 2: Position map size at varying B and Z.

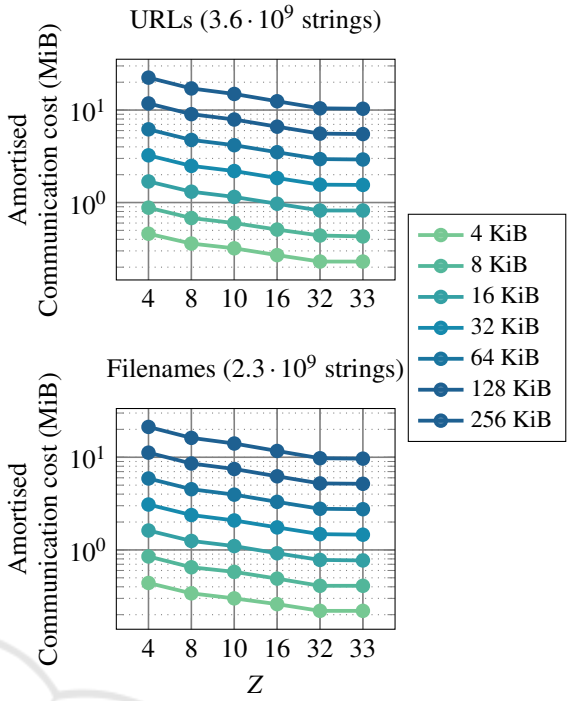

Figure 3: Amortised communication cost per access to Ring ORAM, varying B and Z.

(size of exchanged data) and time per Ring ORAM access. In Figure 3 (next page), the amortised communication cost is shown in MiB at varying B and Z. We can see that it doubles with B, while it decreases when Z increases. Specifically, communication costs increase from 0.5 MiB up to 22 MiB, for both URLs and filenames, when B increases from 4 KiB to 256 KiB and $Z = 4$. When Z increases from 4 to 33, the communication cost reduces by 46-55%, reaching 0.2 MiB and 10 MiB, respectively.

To estimate in a fair way the efficiency of our approach, we consider the cost of exchanging the above amount of data during a query operation over a moderate-speed network with the following average performance: 14.4 MB/s in download and 8.0 MB/s in upload.[2] Assuming these network performance, when $Z = 4$ and $B = 4$ KiB we spend about 50 ms in client-server communication per access, and this time doubles up to reach 2.4 seconds when $B = 256$ KiB. These times decrease to, respectively, 25 ms and 1.1 seconds when $Z = 33$.

[2]We examine the download speeds of the countries ranked just below position 50 - Belgium, Guyana, and Cyprus - on 152 countries and assume that their upload speeds are approximately 1.8× slower than their download speeds, in line with the trend observed in the Global Performance data. The reference data are those reported on speedtest.net

Table 4: Average encryption and decryption times for different block sizes.

B (KiB)	Avg encr. time (μs)	Avg decr. time (μs)
4	12.13	3.60
8	20.81	4.89
16	38.22	7.46
32	73.05	12.45
64	142.61	22.46
128	281.59	42.42
256	561.66	81.99

4.1.4 Encryption and Decryption Cost

From Lemma 2 we know how many (data and metadata) blocks are encrypted and decrypted every A accesses to Ring ORAM. However, we know from Section 2 that encryption times are higher than decryption times. To better understand the impact of using encryption on query times, we estimate the amortised encryption time per access to Ring ORAM by experimentally evaluating the time needed to encrypt and decrypt a block with AES-128 in CBC mode. For the experiments, we used the implementation provided by the OpenSSL library[3].

[3]https://openssl-library.org/

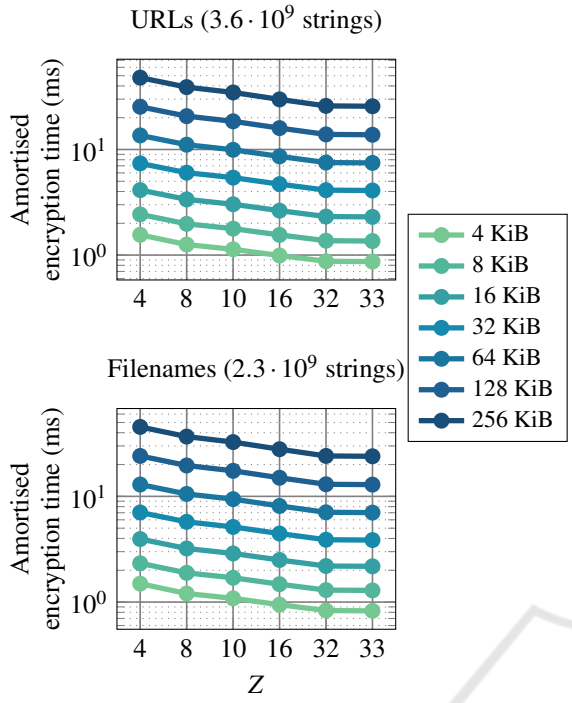

Figure 4: Amortised time per query spent in encryption and decryption operations, varying B and Z.

Table 4 shows the time needed to encrypt and decrypt one block of size B, when B ranges from 4 KiB to 256 KiB. Here, the encryption times are always longer than the decryption times, and their difference grows with B. Specifically, encryption times are 3.4× bigger than decryption times when $B = 4$ KiB and grow up to be 6.8× bigger when $B = 256$ KiB. Encryption times about double with B, while decryption times show the same trend for blocks $B \geq 16$ KiB but grow slowly for smaller blocks. The maximum size of metadata blocks is 0.14 KiB, so their encryption and decryption times are bounded by the cost on real blocks for $B = 4$ KiB.

With both encryption and decryption times, as well as the number of encrypted and decrypted blocks available, we can now estimate the amortised time to make one Ring ORAM access, which is shown in Figure 4. Here we can see that the amortised time becomes $1.5 - 1.9\times$ bigger when B doubles, while it decreases when Z increases. Specifically, the amortised time increases from 1.6 ms and 1.5 ms up to 48 ms and 46 ms for, respectively, URLs and filenames, when B increases from 4 KiB to 256 KiB and $Z = 4$. However, when Z increases from 4 to 33, the amortised time per access to the Ring ORAM reduces by 46-55%, reaching 0.8 ms and 26 ms for URLs, and 0.8 ms and 24 ms for filenames, when $B = 4$ KiB and $B = 256$ KiB, respectively. Compared to the client-server communication cost, stated before, the amortised time a query spends in encryption/decryption operations is negligible ($28 - 49\times$ smaller).

4.1.5 Patricia Trie Traversal and Block Scanning

We finally estimate the time spent by the client to traverse the Patricia trie and scan the retrieved block from the Ring ORAM by looking at the results discussed in (Ferragina et al., 2025, §4.3). We know that trie traversal and block scan on URLs and filenames require, respectively, about 101 μs and 31 μs when $B = 4$ KiB and 93 μs and 34 μs when $B = 8$ KiB. For $B \geq 8$ KiB, these times increase with B, but they stay negligible with respect to all the other costs.

Overall, from the above calculations and the search algorithm stated in Section 3, it follows that the lexicographic search for a string q in a bounded-length string dictionary involves two accesses to the Ring ORAM stored in the server, and thus its time is about 100 ms for $B = 4$ KiB and $Z = 4$. Moreover, this time grows when B gets larger and/or Z gets smaller, as commented above. We can manage longer strings, up to $B = 32$ KiB, exchanging at most about 3 MiB (in 350 ms) per query if $Z = 33$.

4.2 Ring ORAM vs wORAM

In the proposed scheme, we prefer storing strings in groups with Ring ORAM rather than storing the strings using ORAMs designed for variable-length data. In this section, we compare the performance discussed above with the one we would have by storing the strings with wORAM, the only general-purpose ORAM designed for variable-length data with no leakage. We do not consider the privacy-preserving SSE based on TWORAM, which addresses a different problem. We assume wORAM is instantiated on Path ORAM, and not on Ring ORAM, because, by design, Ring ORAM is not well-suited for the wORAM construction. For the comparison, we focus on three of the four metrics discussed above: the server and client space usage, the communication cost, and the encryption cost. For a fair comparison, before storing the strings of URLs and Filenames dataset in wORAM, we assume that they are stored compressed with Huffman coding, which, unlike Rear coding (used when strings are stored with Ring ORAM), allows us to encode and decode each string independently.

4.2.1 Space Occupancy in the Server

We begin our wORAM analysis by evaluating the space required to store the URLs and Filenames datasets when compressed using Huffman coding.

Assuming that the codewords are computed over the entire dataset (i.e., a single symbol–codeword mapping is used for all strings, as discussed in Section 2.3), the total space needed to store 272.2 GB URLs and 68.9 GB filenames is reduced to 175.8 GB and 46.3 GiB, respectively. This corresponds to an overall space reduction of approximately 40%. As also observed for our scheme (see Table 3), when wORAM is used with block sizes between 4 KiB and 256 KiB, not all strings are stored in their entirety. For instance, when blocks have size $B = 4$ KiB, only 6 (compressed) filenames result longer than B, and so they are truncated[4] to guarantee the wORAM's condition on string length is satisfied.

As said above, for our experiments, we assume wORAM is built on Path ORAM. As in (Stefanov et al., 2018), we assume Path ORAM has height $L = \lceil \log_2(N/B) \rceil - 1$, where N is the size of our compressed datasets and B is the size of the blocks. Furthermore, we assume buckets are composed of $Z = 4$ blocks, which is the minimum value of Z for which Path ORAM works well in practice (Stefanov et al., 2018), and the configuration that yields the smallest communication and encryption costs. Then, the wORAM tree has buckets of size $(Z+1)B = 5B$, because the weighted version of Path ORAM has bucket capacity $Z+1$ blocks of size B (Assouline and Minaud, 2023).

From the tree height and bucket size, we compute the resulting wORAM tree size, which, for all the considered block sizes B, is approximately $5\times$ larger than the corresponding non-compressed datasets. This space usage is comparable to that of our searching scheme on Filenames, but roughly twice as large for URLs. when compared to our best configuration.

4.2.2 Performance Comparison

wORAM, as Ring ORAM, stores on the client a position map, whose size depends on the number n of variable-length data we store, and a stash[5]. As before, we won't analyse the stash due to its negligible size.

We compare wORAM performance with the one shown by the best configuration of our approach.

Figure 5 reports the position-map size and communication cost of our approach (in its best configuration) compared with wORAM built over the compressed URLs and Filenames datasets, considering

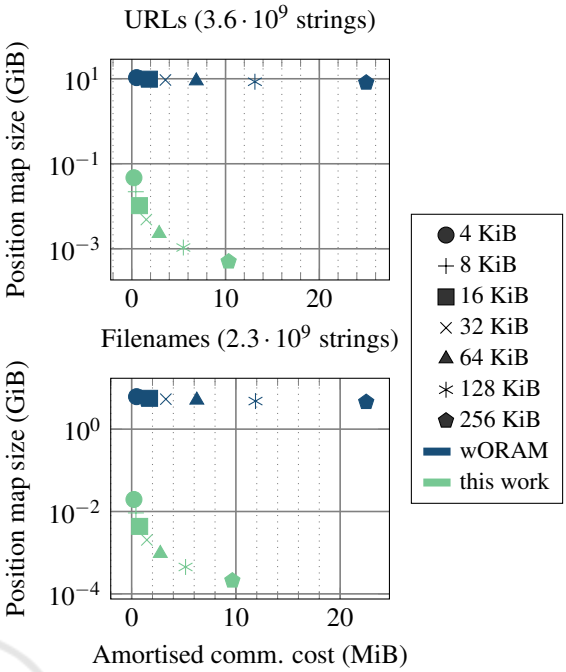

Figure 5: Position map size and communication cost of wORAM and our searching scheme for block sizes between 4 and 256 KiB.

block sizes B ranging from 4 KiB to 256 KiB.

We observe that wORAM produces significantly larger position maps than our proposal, as it stores one entry per string regardless of its length. For URLs, the position map size ranges from 11.9 to 8.1 GB as B increases from 4 KiB to 256 KiB, while for Filenames, it ranges from 6.9 to 4.5 GB. These values are between $23\times$ and $16188\times$ larger than those of our approach for URLs, and between $35\times$ and $21908\times$ larger for Filenames.

Communication cost of wORAM doubles with B, increasing from approximately 0.5 MiB to 25.0 MiB for URLs and from 0.5 MiB to 22.5 MiB for Filenames. Across all considered block sizes B, this cost is consistently about twice (between $1.9\times$ and $2.4\times$) that of our approach.

The comparison with wORAM's encryption times is reported in Figure 6. Across all block sizes B, wORAM exhibits encryption times per access operation ranging from 1 to 64 milliseconds for both URLs and Filenames, remaining $2.3 - 2.5\times$ higher than those achieved by our approach.

4.2.3 Overall wORAM Results

From the discussion above, we observe that our proposal outperforms wORAM in client space usage, amortized communication cost, and amortized encryption time, while achieving comparable or better

[4]For truncated strings, the removed characters are not considered in the total length N of the compressed dataset.

[5]To be able to compress and decompress the strings, the client must keep in its memory some additional information about Huffman coding. However, the space due to these additional data is negligible, so we do not include it in the discussion about client memory footprint.

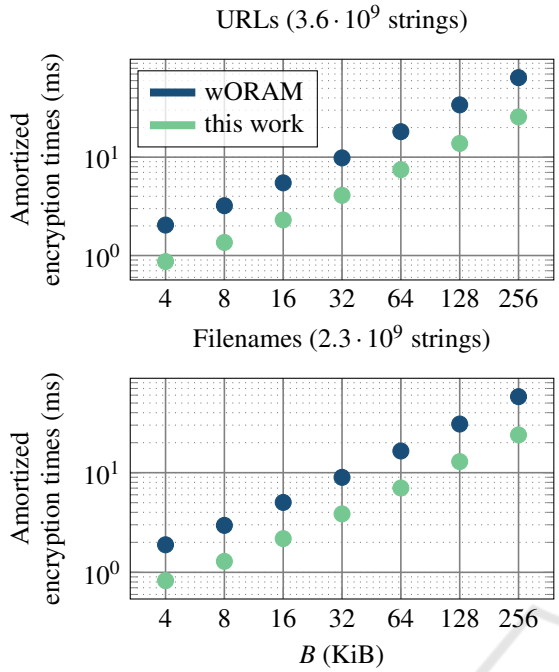

Figure 6: Amortized encryption times of wORAM and our searching scheme for block sizes between 4 and 256 KiB.

server space usage.

While wORAM exhibits communication and encryption costs roughly twice as high as those of our proposal, its position maps are substantially larger: at least 23× bigger than the ones we construct in this paper. Due to their size, being able to keep wORAM's position maps on the client is not a given, but storing them on the server would further increase communication and encryption overhead.

5 CONCLUSIONS

In this paper, we show that an oblivious search scheme for variable-length strings can be obtained by combining two well-known components: Ring ORAM, to hide access patterns while storing data on an honest-but-curious server, and a succinct Patricia trie for efficient indexing of strings. This design allows us to store bounded-length strings of variable length efficiently while supporting oblivious lexicographic, prefix, and membership queries.

We support these claims through evaluations on real-world datasets of up to 273 GB and 3.7 billion strings. Here, estimate server and client space, communication, and cryptographic costs as functions of the protocol parameters, and we show that our construction preserves the guarantees of the underlying primitives. Specifically, we show that we can support the complex string queries above with a Patricia trie of at most 140 MB in the Client, and exchanging $0.5-3$ MiB (in 50-350 ms) per query when the block size $B \in [4,32]$ KiB and $Z = 33$, where Z is the maximum number of real blocks per bucket.

Overall, our analyses indicate that this simple combination of established techniques not only works well in practice, but also improves over state-of-the-art ORAM protocols for variable-length data, delivering a $2.0-2.5\times$ reduction in communication and encryption/decryption costs and a position map that is from 25 up to 20,000× smaller. This provides a practical foundation for oblivious search over large outsourced string dictionaries.

ACKNOWLEDGEMENTS

The work of Paolo Ferragina was partially supported by the Alfred P. Sloan Foundation with the grant #G-2025-25193 (sloan.org). The work of Giuseppe Persiano was partially supported by a Google Sponsored Research Contract.

AI was used in this work only to improve writing. Everything else (tables, analyses, etc.) was produced without AI.

REFERENCES

Assouline, L. and Minaud, B. (2023). Weighted oblivious RAM, with applications to searchable symmetric encryption. In Hazay, C. and Stam, M., editors, *EUROCRYPT 2023, Part I*, volume 14004 of *LNCS*, pages 426–455. Springer, Cham.

Bienstock, A., Patel, S., Seo, J. Y., and Yeo, K. (2023). Near-optimal oblivious key-value stores for efficient PSI, PSU and volume-hiding multi-maps. In Calandrino, J. A. and Troncoso, C., editors, *USENIX Security 2023*, pages 301–318. USENIX Association.

Boldi, P., Marino, A., Santini, M., and Vigna, S. (2018). BUbiNG: massive crawling for the masses. *ACM Trans. Web*, 12(2):12:1–12:26. Datasets of URLs available at https://law.di.unimi.it/datasets.php.

Dautrich, Jr., J. L., Stefanov, E., and Shi, E. (2014). Burst ORAM: Minimizing ORAM response times for bursty access patterns. In Fu, K. and Jung, J., editors, *USENIX Security 2014*, pages 749–764. USENIX Association.

Fan, J., Khan, J., Singh, N. P., Pibiri, G. E., and Patro, R. (2024). Fulgor: a fast and compact k-mer index for large-scale matching and color queries. *Algorithms for Molecular Biology*, 19(1):3.

Ferragina, P. and Grossi, R. (1999). The string b-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM (JACM)*, 46(2):236–280.

Ferragina, P., Rotundo, M., and Vinciguerra, G. (2025). Two-level massive string dictionaries. *Information Systems*, 128:102490.

Garg, S., Mohassel, P., and Papamanthou, C. (2016). TWORAM: Efficient oblivious RAM in two rounds with applications to searchable encryption. In Robshaw, M. and Katz, J., editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 563–592. Springer, Berlin, Heidelberg.

Garimella, G., Pinkas, B., Rosulek, M., Trieu, N., and Yanai, A. (2021). Oblivious key-value stores and amplification for private set intersection. In Malkin, T. and Peikert, C., editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 395–425, Virtual Event. Springer, Cham.

Grossi, R. and Ottaviano, G. (2015). Fast compressed tries through path decompositions. *ACM J. Exp. Algorithmics*, 19.

Jacobson, G. (1989). Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science*, pages 549–554.

Khan, J., Kokot, M., Deorowicz, S., and Patro, R. (2022). Scalable, ultra-fast, and low-memory construction of compacted de bruijn graphs with cuttlefish 2. *Genome biology*, 23(1):190.

Liu, Z., Huang, Y., Li, J., Cheng, X., and Shen, C. (2018). Divoram: Towards a practical oblivious ram with variable block size. *Information Sciences*, 447:1–11.

Martínez-Prieto, M. A., Brisaboa, N., Cánovas, R., Claude, F., and Navarro, G. (2016). Practical compressed string dictionaries. *Information Systems*, 56:73–108.

Morrison, D. R. (1968). Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534.

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

Pibiri, G. E. (2023). On weighted k-mer dictionaries. *Algorithms for Molecular Biology*, 18(1):3.

Pibiri, G. E., Shibuya, Y., and Limasset, A. (2023). Locality-preserving minimal perfect hashing of k-mers. *Bioinformatics*, 39(Supplement_1):i534–i543.

Ren, L., Fletcher, C. W., Kwon, A., Stefanov, E., Shi, E., van Dijk, M., and Devadas, S. (2015). Constants count: Practical improvements to oblivious RAM. In Jung, J. and Holz, T., editors, *USENIX Security 2015*, pages 415–430. USENIX Association.

Roche, D. S., Aviv, A. J., and Choi, S. G. (2016). A practical oblivious map data structure with secure deletion and history independence. In *2016 IEEE Symposium on Security and Privacy*, pages 178–197. IEEE Computer Society Press.

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*, 65(4):1–26.

Zhang, H., Lim, H., Leis, V., Andersen, D. G., Kaminsky, M., Keeton, K., and Pavlo, A. (2020). Succinct range filters. *ACM Trans. Database Syst.*, 45(2).