Efficient Searchable Encryption Through Compression

Ioannis Demertzis
University of Maryland
yannis@umd.edu

Rajdeep Talapatra
University of Maryland
rajdeep@umd.edu

Charalampos Papamanthou
University of Maryland
cpap@umd.edu

ABSTRACT

In this work we design new searchable encryption schemes whose goal is to minimize the *number of cryptographic operations* required to retrieve the result—a dimension mostly overlooked by previous works, yet very important in practice. Our main idea is to utilize compression so as to reduce the size of the plaintext indexes before producing the encrypted searchable indices. Our solution can use any existing Searchable Encryption (SE) scheme as a black-box and any combination of lossless compression algorithms, without compromising security.

The efficiency of our schemes varies based on the leakage exposed by the underlying application. For instance, for private keyword search (more leakage), we demonstrate up to **188×** savings in search time, while for database search (less leakage) our savings are up to **62×**.

The power of our approach is better manifested when combined with more secure, yet less practical, cryptographic tools, such as Oblivious Random Access Memory (ORAM). In particular while ORAM is known to be prohibitively expensive for large-scale applications, we show that our compress-first-ORAM-next approach allows significant more efficient index search time, reducing the time for executing a query with result of size more than one million tuples from approximately 21 hours to 20 minutes.

1. INTRODUCTION

Searchable Encryption (SE) enables a data owner to outsource a document collection to a server in a private manner, so that the latter can still answer keyword search queries without learning too much information (formally modeled through a leakage function) about the underlying document collection and the queried keywords. SE schemes can be

used as an alternative to other approaches for searching encrypted data that are either very expensive (e.g., Oblivious RAM (ORAM) [27, 42, 49] and fully-homomorphic encryption [23, 24]) or offer very weak security guarantees (e.g., order-preserving encryption and deterministic encryption encryption [50, 4, 8, 43]). In a typical SE scheme, the data owner prepares an encrypted index which is then sent to the server. To perform a keyword search for a given keyword w, a token $t(w)$ is sent by the data owner to the server, thus allowing the server to retrieve pointers to the encrypted documents containing keyword w. SE is primarily used for private keyword search but can also be used for database searches, e.g., point queries. Recent works [11, 19, 22] used SE to perform range searches as well as more expressive queries often used in databases. In particular, Demertzis et al. [19] reduced the problem of range search to keyword search for multiple keywords using any SE scheme as a black box illustrating the importance of SE in private databases; any advances in SE directly impacts such works.

Since the first work on SE schemes proposed in 2000 [47], all follow-up works with linear size encrypted index (e.g., [31, 11, 10, 6]) require the server to perform cryptographic operations (PRF[1] evaluations) to retrieve the result, the number of which is at least equal to the size of the query result. The main reason is that for security purposes a query result r is stored in $|r|$ random positions indexed by $|r|$ values each of them produced by a PRF. The server, given a token $t(w)$, has to expand it on $|r|$ sub-tokens (using PRF evaluations) to locate, retrieve and return to the client all the $|r|$ pieces of the result. A PRF evaluation corresponds to the most expensive operation in the search algorithm. The main question we are therefore asking in this paper is:

Can we design SE schemes that retrieve the keyword search result r with less than $|r|$ cryptographic operations?

It is worth noting that previous SE schemes with constant locality (ones that require few random accesses to retrieve the result) (e.g.,[6, 21, 18]) have partially addressed this problem by reducing the number of cryptographic operations required by the server and not the total number of cryptographic operations. In this paper we take a more aggressive approach and aim at reducing the *total* number of cryptographic operations required by the protocol.

---

[1] A Pseudo Random Function (PRF) is a two-input function F: $\{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$, where the first input is called the key and the second is the input x. F can be distinguished from a truly random function only with negligible probability. For a formal definition see [32].

Our Contributions. In this paper we propose a novel SE scheme for private keyword and database search using compression that addresses the above question.

1. Our SE scheme is the first in which the document identifiers matching a queried keyword can be retrieved with potentially less cryptographic operations than the result size $|r|$.[2] Informally we achieve that by storing an encryption of a compressed index instead of a traditional encrypted index. While combining compression and encryption can create security problems [33], we leverage the already existing leakage of searchable encryption to overcome this. We formally prove that our scheme can use any secure SE scheme as black-box (including the recent locality-aware SE schemes [6, 21, 18]), and any set of lossless compression algorithms improving the search efficiency of the used black-box by orders of magnitude without affecting its security.

2. We experimentally evaluate our scheme and show that, for the case of keyword search, it achieves up to **188×** speedup in terms of search time, compared to the most practical in-memory SE scheme. For the case of database search (where there are no overlaps across results and thus less structural leakage—see Section 2.2), we show that our saving is still high, up to **62×** for the *location description* attribute; up to **203×** for binary attributes (see section 4). Our SE scheme can be used as black-box in [19, 20] for further improving the performance of private range/aggregation queries and in [29] for improving boolean queries.

3. We combine our scheme with Oblivious RAM (ORAM) approaches and propose Oblivious SE (OSE), a scheme that answers private keyword and point queries with ORAM-style security guarantees. Our OSE scheme reduces the index search time to access one million tuples using a state-of-the-art ORAM scheme approximately from 21 hours to 20 minutes. Our OSE scheme can be used in the recent works for oblivious querying processing [45, 57, 41] in order to further improve their performance.

Technical Highlights. Our scheme uses two main techniques: First it applies compression algorithms to reduce the size of the unencrypted document identifiers; then it partitions the compressed bit-string into λ bits and encrypts the λ-bit words—in this way it better utilizes the fact that the output of a PRF function is λ bits. Finally it uses any SE scheme as a black box to store the encrypted/packed index.

Like we mentioned above, in the version of our scheme with minimal leakage (database search), we need to compress identifiers that are uniformly distributed in the range $[0, n-1]$, where n is the total number of documents in our collection. We use EWAH [39] and FastPfor [38] since they have been shown [52] to work well in practice when applied to uniform data. Since the bottleneck in SE is the number of cryptographic operations we execute both compression algorithms for each keyword list and choose the best (by storing some extra bits per keyword).

---

[2]Our scheme offers better search time for result sizes greater than 1; otherwise our scheme requires one cryptographic operation, just like other SE schemes. Thus, our scheme does not have performance benefits when all documents contain different words as well as in the database search for unique-key attributes.

2. BACKGROUND

In this section we present the necessary related work, notation and security definitions on searchable encryption. We also provide the necessary background on the compression algorithms that we use throughout this paper.

2.1 Prior Work

Searchable Encryption (SE). In 2000, Song et al. [47] presented the first SE scheme, secure under Chosen Plaintext Attacks (CPA)[3]. Goh [25] realized that CPA security is not adequate for the case of SE schemes. Curtmola et al. [16] introduced the state-of-the-art security definitions for SE for both, non-adaptive settings, i.e. maintaining security only if all the queries are submitted at once in one batch, as well as adaptive settings, i.e. maintaining security even if the queries are progressively submitted, and provided constructions that satisfy their definitions. In this paper, we use the latter type of setting, namely the adaptive setting. The work of Curtmola et al. [16] led the way for the appearance of several new SE schemes [14, 51, 34, 31, 48, 30, 11, 40, 9], some of which allow updates [31, 48, 30, 40, 9], are parallelizable [30] and extend SE to support more expressive queries [29, 11, 22, 19]. For instance, [29] improved the efficiency and security of boolean queries, [22, 19] reduced the problem of range search to multi-keyword search using any SE scheme as a black-box, and [22] extended [11] to support substring, wildcard and phrase queries. In 2014, Cash et al. [10] experimentally showed that in-memory SE cannot scale to large datasets and provided the motivation for designing locality-aware SE as a new research direction, where locality is defined as the number of non-continuous reads that the server makes for each query. Cash and Tessaro [12] proved that a secure SE cannot simultaneously achieve linear index size, optimal locality and optimal read efficiency, where read efficiency is defined as the number of additional memory locations (false positives) that the server reads per result item. Asharov et al. [6] proposed the first schemes with optimal locality and space, and poly-logarithmic read efficiency and soon after, Demertzis and Papamanthou [21] published a tunable SE scheme that achieves various trade-offs between read efficiency, locality and space requirements outperforming in practice the locality optimal schemes of Asharov et al. for similar space requirements. Along this line, Demertzis et al. [18] proposed a new theoretical construction which strictly improves the schemes of Asharov et al. [6], thus providing the first SE with optimal locality, linear space and sub-logarithmic read efficiency.

Oblivious Random Access Memory (ORAM). ORAM compilers introduced by Goldreich and Ostrovsky [27] can compile any RAM program to a memory oblivious program, i.e., the memory access pattern is independent of the input. ORAM [27, 26, 28, 35, 42, 46, 49, 53, 54, 5] is a cryptographic tool that can be used to further improve the security guarantees of SE, such that ideal security is reached. However, despite recent advances in the research area of ORAM [49, 53, 54, 5] these tools remain prohibitively costly for large database applications, as they achieve obliviousness

---

[3]A scheme is secure against Chosen Plaintext Attacks (CPA) if the ciphertexts do not reveal any information about the plaintext even if the adversary can observe the encryption of the messages of his choice. For a formal definition see [32]

Table 1: Notation for keyword and database search.

Symbol	Definition
	Keyword Search
w	keyword
$\mathcal{D}$	input document collection
$\mathcal{D}(w)$	set of document identifiers that contain keyword w
Δ	dictionary containing all the keywords
N	number of keyword-id pairs, i.e., $N = \sum_{\forall w \in \Delta} |\mathcal{D}(w)|$
n	number of documents
D_i	document i
	Database Search
v	value
$\mathcal{T}$	input relational table
$\mathcal{T}^j(v)$	set of tuple idendifiers that contain value v in attribute j
Δ^j	dictionary containing the distinct values of attribute j
N	number of tuples
m	number of attributes
T_i	tuple i

by accessing a poly-logarithmic number of extra memory accesses. Note that, this work does not require deep knowledge on ORAM compilers because our proposed OSE scheme can use any of these as a black-box; for further details we refer the reader to the recent comprehensive analysis of Chang et al. [13] that covers a wide spectrum of existing ORAMs.

2.2 Keyword Search vs. Database Search

SE was originally meant for private file/keyword search, but recent works [19, 21] realized that SE can also be used for database search, and in particular for point and range queries. Keyword search and database search are very similar problems assuming for simplicity that in the latter case we want to support queries on a single attribute. Then, we can map the notion of keywords to the notion of attribute values, and the notion of documents to the notion of tuples, which allows the utilization of SE for database search. The only difference between these two problems is the following: While in the keyword search problem, two different keywords can map to the same document, in the database search problem, by definition of the problem, two different values of the same attribute will never map to the same tuple. For example, a patient cannot have more than one date of birth or SSN number. Therefore, the database search problem has less structural leakage. Table 1 summarizes the most important notation used in this paper, and illustrates the correlation between the keyword and database search. Throughout this paper, and for simplicity and compatibility with prior works in SE we follow the keyword search notation, unless stated otherwise.

Database search for multiple attributes. We can further extend SE to support private database search on multiple attributes. The first solution is to create m copies of the database, where m is the number of attributes, and use SE to encrypt each copy with a different key. This solution expands the space by a factor of m, but achieves optimal leakage, since it treats each attribute separately. The second solution considers only one copy of the database, but it increases the leakage. In particular, we create a single

encrypted index for the values of all attributes by setting as searchable value v_i of attribute $attr_j$, the value $attr_j||v_i$. In this case a tuple id will be found in exactly m searchable values, leaking more information than before (e.g., the set of tuples matching queries on two different attributes).

2.3 Searchable Encryption (SE) Definition

Let $\mathcal{D}$ be a collection of *documents*. Each document $D \in \mathcal{D}$ is assigned a unique document identifier and contains a set of keywords from a dictionary Δ. We recall $\mathcal{D}(w)$ denotes the document identifiers of documents containing keyword w. SE schemes focus on building an *encrypted index $\mathcal{I}$* on the document identifiers. For simplicity, we only consider the document identifiers instead of the actual documents since these are encrypted independently and stored in the server separately from the encrypted index $\mathcal{I}$; whenever the client retrieves a specific identifier during a search, he can send it to the server in an extra round and the server can send the corresponding documents back. Finally, N is the data collection size, i.e., $N = \sum_{\forall w \in \Delta} |\mathcal{D}(w)|$. A SE *protocol* considers two parties, a *client* and a *server* and consists of the following algorithms/protocols [16]:

- $k \leftarrow \mathsf{KeyGen}(1^\lambda)$: is a probabilistic algorithm performed by the client. It receives as input a security parameter λ and outputs a secret key k.

- $(st_\mathcal{C}, \mathcal{I}) \leftarrow \mathsf{Setup}(k, \mathcal{D})$: is a probabilistic algorithm performed by the client prior to sending any data to the server. It receives as input a secret key k and the data collection $\mathcal{D}$, and outputs an encrypted index $\mathcal{I}$. Index $\mathcal{I}$ is sent to the server. $st_\mathcal{C}$ is sent to the client and it contains only the secret key k.

- $(\mathcal{X}, st'_\mathcal{C}, \mathcal{I}') \leftrightarrow \mathsf{Search}(st_\mathcal{C}, w, \mathcal{I})$: is a protocol executed between the client and the server, where the client inserts the secret state $st_\mathcal{C}$ and a keyword w, while the server inserts an encrypted index $\mathcal{I}$. At the end of the protocol the client learns $\mathcal{X}$, the set of all document identifiers $\mathcal{D}(w)$ corresponding to the keyword w and the updated secret state $st'_\mathcal{C}$, while the server's output is the updated encrypted index $\mathcal{I}'$.

We provide the security definition for the above SE scheme that corresponds to the real-ideal world paradigm [16], with a slightly modified syntax in order to match the security definition of OSE.

DEFINITION 1. *Suppose* $(\mathsf{KeyGen}, \mathsf{Setup}, \mathsf{Search})$ *is a SE scheme based on the above definition, let* $\lambda \in \mathbb{N}$ *be the security parameter and consider experiments* $\mathbf{Real}(\lambda)$ *and* $\mathbf{Ideal}_{\mathcal{L}_{\mathsf{SETUP}}, \mathcal{L}_{\mathsf{QUERY}}}(\lambda)$ *presented in Figure 1, where* $\mathcal{L}_{\mathsf{SETUP}}$ *and* $\mathcal{L}_{\mathsf{QUERY}}$ *are leakage functions to be defined next. We say that the SE scheme is* $(\mathcal{L}_{\mathsf{SETUP}}, \mathcal{L}_{\mathsf{QUERY}})$-secure[4] *if for all polynomial-size adversaries* $\mathcal{A}$ *there exist polynomial-time*

---

[4]In prior works $(\mathcal{L}_{\mathsf{SETUP}}, \mathcal{L}_{\mathsf{QUERY}})$ is mentioned as $(\mathcal{L}_1, \mathcal{L}_2)$, where $\mathcal{L}_{\mathsf{SETUP}}$ or $\mathcal{L}_1$ refers to the total setup leakage, i.e. leakage prior to the query execution, and $\mathcal{L}_{\mathsf{QUERY}}$ or $\mathcal{L}_2$ refers to the total query leakage, i.e. leakage during the query execution.

$$\begin{array}{ll}
\textbf{Real}(\lambda) & \textbf{Ideal}_{\mathcal{L}_{\text{SETUP}}, \mathcal{L}_{\text{QUERY}}}(\lambda) \\
\hline
k \leftarrow \mathsf{KeyGen}(1^\lambda) & \\
(\mathcal{D}, st_{\mathcal{A}}) \leftarrow \mathcal{A}(1^\lambda) & (\mathcal{D}, st_{\mathcal{A}}) \leftarrow \mathcal{A}(1^\lambda) \\
(st_{\mathcal{C}}, \mathcal{I}_0) \leftarrow \mathsf{Setup}(k, \mathcal{D}) & (st_{\mathcal{S}}, \mathcal{I}_0) \leftarrow \mathsf{SimSetup}(\mathcal{L}_{\text{SETUP}}(\mathcal{D})) \\
\text{for } 1 \le i \le q & \text{for } 1 \le i \le q \\
\quad (w_i, st_{\mathcal{A}}) \leftarrow \mathcal{A}(st_{\mathcal{A}}, \mathcal{I}_{i-1}, M_1, \ldots, M_{i-1})^* & \quad (w_i, st_{\mathcal{A}}) \leftarrow \mathcal{A}(st_{\mathcal{A}}, \mathcal{I}_{i-1}, M_1, \ldots, M_{i-1})^* \\
\quad (\mathcal{X}_i, st_{\mathcal{C}}, \mathcal{I}_i) \leftrightarrow \mathsf{Search}(st_{\mathcal{C}}, w_i, \mathcal{I}_{i-1}) & \quad (\mathcal{X}_i, st_{\mathcal{S}}, \mathcal{I}_i) \leftrightarrow \mathsf{SimSearch}(st_{\mathcal{S}}, \mathcal{L}_{\text{QUERY}}(\mathcal{D}, w_i), \mathcal{I}_{i-1}) \\
\text{let } \mathbf{M} = M_1 \ldots M_q, \mathcal{I} = \mathcal{I}_0 \ldots \mathcal{I}_q \text{ and } \mathcal{X} = \mathcal{X}_0 \ldots \mathcal{X}_q & \text{let } \mathbf{M} = M_1 \ldots M_q, \mathcal{I} = \mathcal{I}_0 \ldots \mathcal{I}_q \text{ and } \mathcal{X} = \mathcal{X}_0 \ldots \mathcal{X}_q \\
\text{output } v = (\mathcal{I}, \mathbf{M}, \mathcal{X}), st_{\mathcal{A}} & \text{output } v = (\mathcal{I}, \mathbf{M}, \mathcal{X}), st_{\mathcal{A}}
\end{array}$$

* Let M_k be all the messages from client to server in the Search/SimSearch protocol above.

Figure 1: SE/OSE ideal-real security experiments.

simulators $\mathsf{SimSetup}$ *and* $\mathsf{SimSearch}$, *such that for all polynomial time algorithms* Dist:

$$| \Pr[\mathsf{Dist}(v, st_{\mathcal{A}}) = 1 : (v, st_{\mathcal{A}}) \leftarrow \textbf{Real}(\lambda)] -$$
$$\Pr[\mathsf{Dist}(v, st_{\mathcal{A}}) = 1 : (v, st_{\mathcal{A}}) \leftarrow \textbf{Ideal}_{\mathcal{L}_{\text{SETUP}}, \mathcal{L}_{\text{QUERY}}}(\lambda)]|$$
$$\le negl(\lambda),$$

where probabilities are taken over the coins of KeyGen *and* Setup *algorithms*[5].

Figure 1 presents the real and ideal games for (semi-honest) adaptive adversaries, as introduced in [15]. These games are used to formally prove the security of an SE scheme. They are partitioned into two worlds, the real and the ideal one. The real world represents a real SE scheme, where the adversary has access to the Setup and Search algorithms. More specifically, the real scheme creates a secret key to which the adversary does not have access. The adversary selects a document collection which is given as an input to the Setup algorithm. Furthermore, $st_{\mathcal{A}}$ denotes a state maintained by the adversary. The adversary observes the output of the Setup algorithm which is the encrypted index. Then, she selects a polynomial number of queries, and for each of these queries she observes the corresponding tokens. Having these tokens allows her to retrieve the encrypted result. In the ideal world, the adversary interacts with the simulator. The simulator $\mathcal{S}$, neither has access to the real document collection, nor to the real queries. Instead, the simulator only has access to predetermined leakage functions and by using these functions and her state she attempts to "fake" the algorithms Setup and Search. The adversary can only have access to one world, either to the real one, or to the ideal one. We consider only the strongest types of adversaries, i.e., adaptive adversaries that can select their own new queries based on previous ones. The adversary attempts to detect the world to which she has access. We prove that an adversary can distinguish the output of the real world from that of the ideal world only with negligible probability. This means that an adversary cannot learn anything more, than the predefined leakage. We refer the reader to [15, 19] for a more detailed explanation of the security game.

The above security definition and leakages apply only to static SE schemes. The extension of static SE schemes to a dynamic setting requires guaranteeing a property called forward privacy [48], where the server does not get to learn that a newly inserted keyword,id pair satisfies a query issued in the past.

---

[5] A function $\nu \colon \mathbb{N} \to \mathbb{N}$ is negligible in λ, $negl(\lambda)$, if for every positive polynomial $p(\cdot)$ and all sufficiently large λ, $\nu(\lambda) < 1/p(\lambda)$.

2.4 Oblivious RAM (ORAM)

Oblivious RAM (ORAM), introduced by Goldreich and Ostrovsky [27] is a compiler that encodes the memory, such that accesses on the compiled memory do not reveal access patterns on the original memory. We give the definition for a read-only ORAM as this will be a prerequisite in our scheme—the definition naturally extends for writes as well:

- $(\sigma, \mathsf{EM}) \leftarrow \textsc{OramInitialize}(1^\lambda, \mathsf{M})$: takes as input the security parameter λ and the memory array M of n values $(1, v_1), \ldots, (n, v_n)$ and outputs the secret state σ (for the client), and the encrypted memory EM (for the server).

- $(v_i, \sigma, \mathsf{EM}') \leftrightarrow \textsc{OramAccess}(\sigma, i, \mathsf{EM})$: is a protocol between the client and the server, where the client's input is a secret state σ and an index i. The server's input is the encrypted memory EM. The client's output is the value v_i assigned to i and the updated secret state σ'. The server's output is the updated encrypted memory EM'.

Intuitively, an ORAM scheme guarantees that there is no polynomial time adversary that can distinguish any of the two executions of the same program with different inputs of the same size. In particular, the adversary can pick the initial memory and any two polynomial size sequences of accesses y_1 and y_2 of the same length ($|y_1| = |y_2|$) and by observing the oblivious accesses of $o(y_1)$ and $o(y_2)$ she will not able to distinguish them, with non-negligible probability. We refer the reader to [18] for the formal correctness and security definitions of ORAM.

2.5 Leakage Functions

In total we present four SE constructions in this paper that satisfy Definition 1 presented before. Each such scheme has different leakage, as we detail in the following.

- A simple SE construction for keyword search (SE-K);

- A simple SE construction for database search (SE-D);

- An ORAM-based SE construction for keyword search (OSE-K);

- An ORAM-based SE construction for database search (OSE-D).

Every construction leaks different amount of information. In Table 2 we show all the leakages in detail. We now explain the intuition behind these leakages.

construction	$\mathcal{L}_{\text{SETUP}}$	$\mathcal{L}_{\text{QUERY}}$
SE-K	(N, n)	$(id(w), \mathcal{D}(w))$
SE-D	N	$(id(v), |\mathcal{T}^j(v)|)$
OSE-K	(N, n)	$|\mathcal{D}(w)|$
OSE-D	N	$|\mathcal{T}^j(v)|$

Leakages for SE-K: Our simple SE construction for keyword search leaks only the size of the index N and number of the indexed documents n during setup. During a query for w, it leaks $(id(w), \mathcal{D}(w))$ where $id(w)$ is a random-looking λ-bit number that we map to each keyword w, called alias of w (capturing the *search pattern*, i.e., whether a keyword query has been repeated or not). The set $\mathcal{D}(w)$ captures the *access pattern*, revealing which document overlaps between previously queried documents.

Leakages for SE-D: The main difference here is in the query leakage, where we *leak* the size of the access pattern, instead of the access pattern itself. Also, since $N = n$ in the database search scenario, only N is naturally leaked.

Leakages for OSE-K and OSE-D As opposed to construction SE-K and SE-D, our ORAM-based constructions *only leak the size of the result that is returned*. We can consider this leakage to be *ideal* for any scheme that supports sublinear-time search, since in order to hide the size of the result we will need to either download the entire encrypted index, or equivalently to pad the result to the maximum size. In both cases the size of each query answer will be proportional to the input size, i.e. $O(N)$.

2.6 Compression Schemes

Our implementation uses (variations of) two different compression schemes, as we detail in the following.

FastPfor. FastPfor [38] is a modification of PforDelta [58]. Given a list of n integers, the algorithm begins by computing the deltas between two consecutive integers, and then it proceeds to compress the deltas. For example, let $I = \{2, 5, 10, 17\}$, then the deltas would be $I' = \{2, 3, 5, 7\}$ where $I'[0] = I[0]$ and $I'[i] = I[i] - I[i-1] (i \geqslant 0)$. The deltas are then split into chunks of 128 deltas and each of the chunks is compressed separately. For each chunk, the scheme chooses the smallest b, such that a majority of elements (controlled by a threshold, say 90%) can be encoded using b bits. The chunks are then stored using 128 b-bit locations, in addition to some extra storage for the values that could not be represented by the b bits (called *exceptions*). FastPfor enhances PforDelta because it stores the exceptions more efficiently.

EWAH. EWAH (Enhanced Word-Aligned Hybrid) [39] is a bitmap index compression algorithm, which is an enhanced modification of WAH (Word-Aligned Hybrid) [56]. Both algorithms belong to the RLE (Run Length Encoding) compression family. Given a set of n integers, we first create a bitmap in which we set the i-th element of the bitmap to 1 if and only if i is present in the list of input numbers. In WAH, the input bitmap is then split into groups of 31 bits. The groups are classified into two categories;

if all the bits in a group are identical we consider it to be a filled group, otherwise a literal group. For example, 0000000000000000000000000000000 (0^{31} in short) is a filled group. Filled groups can be further classified into 0-fill groups (all bits are 0) and 1-fill groups (all bits are 1). WAH compresses a sequence of consecutive filled groups of the same type together using just one word. The scheme stores each literal group using one word (32 bits).

For instance, if the input bitmap is $10^{20}1^30^{111}1^{25}$ (160 bits), then WAH partitions it into 6 groups: G_1 ($10^{20}1^30^7$), G_2 (0^{31}), G_3 (0^{31}), G_4 (0^{31}), G_5 ($0^{11}1^{20}$) and G_6 ($0^{26}1^5$). Then, WAH encodes G_1 using ($010^{20}1^30^7$), i.e. the first bit is set to 0 denoting that it is a literal word and the remaining 31 bits contain G_1. Furthermore, it encodes G_2, G_3, G_4, G_5 together using ($100^{27}011$), i.e. the first bit is set to 1 indicating that it is a filled word, the second bit is set to 0 indicating that it is a 0-filled word and the remaining bits are used to store how many consecutive 0-groups are stored together. Finally, it encodes G_5 using ($00^{11}1^{20}$) and encodes G_6 using ($00^{26}1^5$).

EWAH is a modification of WAH because it addresses the latter's necessity to allocate too much space to store literal groups. Unlike WAH, EWAH divides an uncompressed input bitmap into 32-bit groups, whereas WAH uses 31-bit groups. Then it encodes a sequence of p ($p \leqslant 65535$) fill groups and q ($q \leqslant 32767$) literal groups into a marker word followed by q literal words (stored in their original form). The first word in EWAH is always a marker word.

3. OUR APPROACH

Our approach consists of two main steps: Given the dataset $\mathcal{D}$, in the first step we compress each list $\mathcal{D}(w)$; The second step uses the output compressed lists (for all keywords w) as input to the SE setup algorithm. When an encrypted search query is performed, the accessed list is much smaller (due to compression)—once we receive the compressed list, we can decompress it and retrieve the result. Note that while asymptotically the time required for the search is the same as other schemes, the number of cryptographic operations are only proportional to the size of the compressed list—this leads to significant savings in practice as we show in the experiments. We now describe various components of our approach in more detail.

Uniform document identifier reassignments. Prior SE schemes assume that for each document identifier we pick a random string of τ bits. Note that performing compression on random strings leads to almost negligible compression. In our approach, instead of assigning a random string of τ bits to each document identifier we assign an id chosen uniformly at random from the range $[0, n-1]$, where n is the total number of documents in our collection. This uniform reassignment of document identifiers is equivalent to having random strings of τ bits (this will be part of our security proofs). However, our approach leads to more efficient compression of the encrypted keyword lists.

Compression and partitioning. As mentioned earlier, instead of storing $\mathcal{D}(w) = \{id_1, \ldots, id_s\}$ in the encrypted index, we will store an encrypted version of that list, namely the string $\mathcal{Y} = \text{Compress}(\mathcal{D}(w))$, computed using some compression algorithm. Note that $\mathcal{Y}$'s size is at most $s \log n$
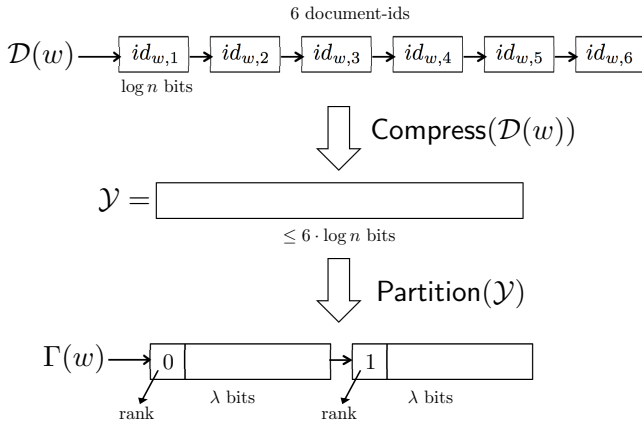
Figure 2: Our scheme first compresses the keyword lists and then performs the partitioning. Note that the packed words need to be stored with a rank, so that the decompression can work correctly.

bits meaning that the number of cryptographic operations required to retrieve the compressed result is reduced—the actual result can be easily retrieved from the compressed result with *no* cryptographic operations.

We further partition $\mathcal{Y}$ to chunks of $\lambda - \log n$ bits (we use $\log n$ bits to store the rank of the chunk as we explain below), where λ is the security parameter. This assures we "pack" the maximum amount of information into one PRF evaluation. For example, we can partition $\mathcal{Y}$ into words G_1, G_2 etc., ending up with approximately $\mu = \frac{|\mathcal{Y}|}{\lambda - \log n}$ words $G_1, G_2, \ldots, G_\mu$. Finally, we store in the encrypted index the compressed list $\Gamma(w) = \{G_1, \ldots, G_\mu\}$. Notice that in the actual construction we store for each word its rank i by attaching $\log n$ bits so that decompression works correctly. Thus, each compressed word will have size λ bits. Our approach is described in Figure 2.

3.1 Choosing Compression Algorithms

We note that there are more than 20 different algorithms for bitmap/inverted list compression — see [52], and therefore identifying the most suitable compression algorithm is a challenging task. In this work we selected two compression algorithms EWAH and FastPfor that take into consideration the specialized structure of SE/OSE, i.e. how we can efficiently compress $|\mathcal{D}(w)|$ uniformly distributed document identifiers for each keyword w. We choose our compression algorithm for each keyword list in a greedy fashion: Find the best compression algorithm for each keyword list $\mathcal{D}(w)$ individually, from a set of compression algorithms $\mathcal{C}$ and store some extra metadata to denote which compression algorithm was used. Below, we explain the reasons we chose EWAH and FastPfor and in which ranges we expect that each compression algorithm will be used in practice.

We chose EWAH for keyword lists of very large size. In particular, we modify the EWAH algorithm to yield compressed words of size $\phi = \lambda - \log n$, instead of the original 32 bits. Setting the compressed words to have size ϕ bits means that the maximum number of required compressed words will be $O(\frac{n}{\phi})$. Intuitively, we expect that we can benefit from this algorithm only when $|\mathcal{D}(w)| > \frac{n}{\phi}$. If

$|\mathcal{D}(w)| < \frac{n}{\phi}$ then with high probability the expected load of each compressed word will be ≤ 1, since the distribution of document identifiers is uniform. In the latter case, we represent each document identifier with $O(\lambda)$ bits, leading to no compression. EWAH achieves savings only if $\mathcal{D}(w) > n/\phi$. For example, in the extreme case that $|\mathcal{D}(w)| = n$, then EWAH will compress all the document identifiers in exactly one compressed keyword with λ bits; in this case EWAH achieves the best possible compression ratio.

We chose FastPfor for keyword lists with small, medium and large sizes. In the extreme case, that $|\mathcal{D}(w)| = n$, FastPfor will compress keyword w using approximately $O(n)$ bits, since it will require at least 1 bit per delta. Thus, we expect that EWAH will be superior for very large keyword-lists, i.e. $|\mathcal{D}(w)| > c_1 \cdot n/\lambda$ (for some constant c_1); FastPfor will handle the remaining keyword-lists. However, there is not a clear separation of the ranges where each of the above compression algorithms will be better and so we follow a greedy selection as we described above.

In the case that the compressed keyword-lists have size greater than $\log n * |\mathcal{D}(w)|$ bits, we use the original uncompressed representation. Notice that the greedy selection is a viable solution and does not significantly affect the Setup time, since the encryption cost is the dominant factor.

3.2 Our SE Construction

Our main SE construction is shown in Figure 3. Note the random document identifier reassignment (Line 2 of the Setup algorithm), compression (Line 4 of the Setup algorithm) and partitioning (Line 5 of the Setup algorithm). In particular, Setup works as follows. After parsing the input index $\mathcal{D}$ the algorithm compresses each keyword list $\mathcal{D}(w_i)$ individually by greedily selecting the best compression method from the set of lossless compression algorithms $\mathcal{C}$ (described in Section 3.1). Then, Setup performs partitioning as described above to obtain the index Γ, which is padded with up to N entries and encrypted using any SE scheme as a black-box.

As shown in Figure 3 the algorithms KeyGen and Token perform only calls to the blackbox algorithms SE.KeyGen and SE.Token, respectively.

The Search algorithm is applied to an input keyword w, in order to retrieve the list $\{1||G_1, 2||G_2, \ldots, \mu||G_\mu\}$ and decompresses the bit string $G_1||G_2||\ldots||G_\mu$ using the same compression algorithm $c \in \mathcal{C}$ that was used for compression. Finally, it outputs the real document identifiers $\{id_1, id_2, \ldots\}$.

Note that the utilized black-box SE scheme *must leak the actual access pattern* (e.g., [10, 21][6]), since otherwise our construction is not correct. This is because randomizing the identifiers in the black-box SE scheme would cause the decompression algorithm to produce garbage. In addition, in the keyword search problem it is not necessary for the used black-box SE to leak both N and n; there are SE schemes that leak only N. However, our construction additionally leaks n—this allows us to define the domain from which we draw the document identifiers.

---

[6] In the literature of SE schemes for the keyword search problem, some of the prior works [6, 18] focus for simplicity only on retrieving the document identifiers of a queried keyword w and not on getting the actual documents. These schemes do not leak the actual access pattern, but only its size. However, it is easy to extend these schemes, such that they return the actual documents and leak the actual access pattern.

$k \leftarrow \mathsf{KeyGen}(1^\lambda)$
1: $k \leftarrow \mathsf{SE.KeyGen}(1^\lambda)$.
2: **return** k.

$(st_\mathsf{C}, \mathcal{I}) \leftarrow \mathsf{Setup}(k, \mathcal{D})$
1: Set $N = |\{\mathcal{D}(w)\}_{w \in \mathbf{W}}|$. Let n be the number of documents ($n \leq N$).
2: Reassign document identifiers using a random permutation $p : [n] \to [n]$ (i.e., document i becomes document $p(i)$).
3: **for** each $w \in \mathbf{W}$ **do**
4: $\quad (\mathcal{Y}, c) \leftarrow \textbf{COMPRESS}(\mathcal{D}(w), \mathcal{C})$. $\quad \triangleright \ \mathcal{C}$ is a set of compression algorithms; c are the bits encoding this choice.
5: $\quad$ Write $\mathcal{Y}$ as $G_1||G_2\ldots||G_\mu$ where G_i is a bit string of $\lambda - \lceil \log n \rceil$ bits (pad the last bit string if needed).
6: $\quad$ Set $\Gamma(w) = \{c, 1||G_1, 2||G_2, \ldots, \mu||G_\mu\}$.
7: Pad Γ to have N entries. $\quad \triangleright$ We insert a dummy keyword which contains the necessary number of dummy values.[a]
8: $\mathcal{I} \leftarrow \mathsf{SE.Setup}(k, \Gamma)$.
9: Set st_C to include k.
10: **return** $(st_\mathsf{C}, \mathcal{I})$.

$(\mathcal{X}, st_\mathsf{C}, \mathcal{I}) \leftrightarrow \mathsf{Search}(st_\mathsf{C}, w, \mathcal{I})$
1: $(c, result) \leftarrow \mathsf{SE.Search}(st_\mathcal{C}, w, \mathcal{I})$.
2: Write $result$ as $\{1||G_1, 2||G_2, \ldots, \mu||G_\mu\}$.
3: $\mathcal{X} \leftarrow \textbf{DECOMPRESS}(G_1||G_2||\ldots||G_\mu, \mathcal{C}, c)$.
4: **return** $(\mathcal{X}, st_\mathsf{C}, \mathcal{I})$.

---
[a]This dummy keyword is never returned as part of any query.

Figure 3: Our more efficient SE construction using any SE and a set $\mathcal{C}$ of compression algorithms as black-box.

$(\mathcal{I}, st_\mathcal{S}) \leftarrow \mathsf{SimSetup}(\mathcal{L}_{\mathsf{SETUP}}(\mathcal{D}))$
1: $(N, n) \leftarrow \mathcal{L}_{\mathsf{SETUP}}(\mathcal{D})$.
2: $(\mathcal{I}, st) \leftarrow \mathsf{SE.SimSetup}(N)$.
3: Let $A = \{1, 2, \ldots, n\}$ be the set of document identifiers.
4: Let Previous and Access be empty hash tables.
5: **return** $(\mathcal{I}, (st, A, \mathsf{Previous}, \mathsf{Access}))$.

$(X, st_\mathcal{S}, \mathcal{I}) \leftarrow \mathsf{SimSearch}(st_\mathcal{S}, \mathcal{L}_{\mathsf{QUERY}}(\mathcal{D}, w), \mathcal{I})$
1: Parse $st_\mathcal{S}$ as $(st, A, \mathsf{Previous}, \mathsf{Access})$.
2: Let $(id(w), R_1, R_2, \ldots, R_s) \leftarrow \mathcal{L}_{\mathsf{QUERY}}(\mathcal{D}, w)$.
3: **if** $\mathsf{Previous.get}(id(w)) \neq$ null **then**
4: $\quad$ **return** $(\mathsf{Previous.get}(id(w)), st_\mathcal{S}, \mathcal{I})$.
5: **else**
6: $\quad$ **for** $i = 1, \ldots, s$ **do**
7: $\quad\quad$ **if** $\mathsf{Access.get}(R_i) =$ null **then**
8: $\quad\quad\quad$ Pick id_i uniformly at random from A and set $A = A - \{id_i\}$.
9: $\quad\quad\quad$ $\mathsf{Access.put}(R_i, id_i)$.
10: $\quad\quad$ **else**
11: $\quad\quad\quad$ $id_i \leftarrow \mathsf{Access.get}(R_i)$.
12: $\quad$ $(\mathcal{Y}, c) \leftarrow \textbf{COMPRESS}(id_1||id_2||\ldots||id_s, \mathcal{C})$.
13: $\quad$ Write $\mathcal{Y}$ as $G_1||G_2\ldots||G_\mu$ where G_i is a bit string of $\lambda - \lceil \log N \rceil$ bits (pad the last bit string if needed).
14: $\quad$ Set $\Gamma(w) = \{1||G_1, 2||G_2, \ldots, \mu||G_\mu\}$.
15: $\quad$ $(\mathcal{X}, st_\mathcal{S}, \mathcal{I}) \leftarrow \mathsf{SE.SimSearch}(st_\mathcal{S}, \Gamma(w), \mathcal{I})$.
16: $\quad$ Parse $\mathcal{X}$ as $(c, result)$ and compute $\mathcal{X}' \leftarrow \textbf{DECOMPRESS}(result, \mathcal{C}, c)$.
17: $\quad$ $\mathsf{Previous.put}(id(w), \mathcal{X}')$.
18: $\quad$ Update $st_\mathcal{S}$ with the new values of $(st, A, \mathsf{Previous}, \mathsf{Access})$.
19: **return** $(\mathcal{X}', st_\mathcal{S}, \mathcal{I})$.

Figure 4: Simulator algorithms $\mathsf{SimSetup}$ and $\mathsf{SimToken}$ for SE (keyword search problem)

Proof of security. We will now prove security of our SE-K construction, assuming the black-box SE scheme we use is $(\mathcal{L}_1, \mathcal{L}_2)$-secure where $\mathcal{L}_1$ leaks only the size of the index N (but not the number of documents n) and $\mathcal{L}_2$ leaks the search pattern and access pattern. We provide the proof for the keyword search problem; proofs for the database search problem are easily derived from the proof we present.

THEOREM 1. *Let $\mathcal{L}_{\mathsf{SETUP}}$, $\mathcal{L}_{\mathsf{QUERY}}$ be the leakages defined in Section 2.5 for the SE-K construction. If the SE scheme used as a black-box in our construction of Figure 3 is $(\mathcal{L}_1, \mathcal{L}_2)$-secure according to Definition 1, then our SE-K construction of Figure 3 is $(\mathcal{L}_{\mathsf{SETUP}}, \mathcal{L}_{\mathsf{QUERY}})$-secure.*

PROOF. Our black-box SE scheme is $(\mathcal{L}_1, \mathcal{L}_2)$-secure, we use its SE.SimSetup and SE.SimSearch algorithms. Our simulator $\mathsf{SimSetup}(\mathcal{L}_{\mathsf{SETUP}}(\mathcal{D}))$ and $\mathsf{SimSearch}(\mathcal{L}_{\mathsf{QUERY}}(\mathcal{D}, w))$ is described in Figure 4.

For the first part of the proof, we must show that no PPT algorithm Dist can distinguish, with more than negligible probability, between the index $\mathcal{I}_{real}$ output by $\mathsf{Setup}(k, \mathcal{D})$ and the index $\mathcal{I}_{ideal}$ output by $\mathsf{SimSetup}(\mathcal{L}_{\mathsf{SETUP}}(\mathcal{D}))$. This is because both $\mathcal{I}_{real}$ and $\mathcal{I}_{ideal}$ have the same number of entries and the black-box SE is $(\mathcal{L}_1, \mathcal{L}_2)$-secure.

For the second part of the proof we need to prove that Dist cannot distinguish between the outputs of $\mathsf{Search}(k, w)$ and the output of $\mathsf{SimSearch}(st_\mathcal{S}, \mathcal{L}_{\mathsf{QUERY}}(\mathcal{D}, w))$.

First, both Search and SimSearch produce the same messages, i.e. tokens and results, for the same repeated keywords. SimSearch uses the search pattern leakage Previous included in $st_\mathcal{S}$. Second, for a keyword w that has not been queried before, it is enough to show that the distribution of $\Gamma(w)$ in Line 6 of Setup and the distribution of $\Gamma(w)$ in Line 14 of SimSearch are identical. If so, the security will follow from the existence of a simulator of the black-box secure SE scheme. It is easy to see that the aforementioned distributions are identical. In the real game, the document identifiers that are compressed are chosen uniformly at random due to the random permutation at Line 2, and in the SimSearch algorithm the identifiers are again picked uniformly at random from A at Line 8, every time a new keyword comes in. The simulator also correctly simulates the overlapped document identifiers between different queries using its state $st_\mathcal{C}$ and the access pattern leakage—see Lines 2 and 11. □

Choosing the black-box SE: Our solution can achieve a high degree of parallelism, good locality trade-offs and dynamism, when selecting the SE black-box scheme to be the optimal read efficiency scheme proposed in [21]. According to [7], the latter scheme achieves optimal space and locality trade-offs since it matches a lower-bound for schemes with optimal read efficiency. The composition of our scheme with the above provides very efficient search time for both in-memory and external memory settings in the standard SE leakage profile. The schemes of [21] with non-optimal read efficiency have different leakage profile; however our scheme can use them as a black-box inheriting all the different trade-offs that they provide, but in that case our scheme will inherit also their leakage profile. Our solution can also be extended to the dynamic case achieving forward privacy (desired property for dynamic SE — see [48]) using the solution proposed in [19, 21].

3.3 Our OSE Construction

Our OSE-K/OSE-D construction is shown in Figure 5 and is based on modifying the SE construction presented in the previous section. The main difference is that instead of using a SE scheme as a black-box, we now use an Oblivious RAM scheme for that purpose. We summarize these modifications.

- The Setup algorithm uses Lines 4 to compute the optimal worst-case number of compressed words μ_0, i.e., it computes for each used compression algorithm the worst-case required number of compressed words (for a given n and $|\mathcal{D}(w)|$) and chooses the most efficient one (we will explain the intuition behind this point below). In Line 8, list $\Gamma(w)$ is padded to contain exactly μ_0 compressed words. In Line 9, Γ is padded to have $2 \cdot N$ entries. In Line 10, Setup computes the encrypted index using ORAMINITIALIZE, i.e., $(st_\mathcal{C}, \mathcal{I}) \leftarrow$ ORAMINITIALIZE$(k, \mathcal{D})$ and in Line 11 it outputs $(st_\mathsf{C}, \mathcal{I})$.

- The Search algorithm calls ORAMACCESS as many times as necessary to receive the entire compressed result, i.e., $\forall i \in [0, \mu + 1)$ we call

$$(\mathcal{X}, st'_\mathcal{C}, \mathcal{I}') \leftrightarrow \text{ORAMACCESS}(st_\mathcal{C}, st_\mathcal{C}.pos(G||i), \mathcal{I}).$$

The client's state $st_\mathcal{C}$, comprises all the information that the client needs to know in order to retrieve each G_i, i.e., a mapping indicating that G_i is stored in index j (this mapping is called *position map* and we denote it as $st_\mathcal{C}.pos(G_i)$—we will further explain the notion of position map below).

Important observation concerning security. We observe that in the SE construction (Figure 3) a keyword-list of $|\mathcal{D}(w)|$ size may be compressed into μ compressed words in one execution, while in another execution it may be compressed into $\mu + 1$ compressed words. However, as we proved this does not induce any security issues, since in both cases the distributions of the document identifiers are the same. Therefore, even if the adversary queries the same keyword multiple times, the simulator will simulate the query only the first time and for any subsequent execution of the same query she will use the search pattern leakage to return the previously chosen result (see Line 11 of Figure 4). A very important difference between the SE and OSE constructions is that the latter does not leak the search pattern, i.e., whether two encrypted search queries are the same. Let us consider the case of an adversary querying the same keyword w_1 multiple times in our OSE construction. In that case Setup will **always** produce the same number of compressed words μ, while SimSetup might yield a different number of compressed words in every execution.

In order to address the aforementioned problem, it is required that all keyword lists of the same size s to have the same number of compressed words. To achieve this, **WORST-COMPRESSION** (in Line 4) computes for each $c \in \mathcal{C}$ the worst-case compression (given n, s and $\mathcal{C}$) and returns the best algorithm c, and the worst-case number of compressed words μ_0 for c. Now, we first compress the list $\mathcal{D}(w)$ as before (see Lines 5-7) and then pad it to size μ_0 (Line 8).

We note here that computing μ_0 is easy for some algorithms, e.g., WAH, EWAH but for other algorithms, such

$k \leftarrow \mathsf{KeyGen}(1^\lambda)$

1: $k \leftarrow^\$ \{0,1\}^\lambda$.
2: **return** k.

$(st_\mathsf{C}, \mathcal{I}) \leftarrow \mathsf{Setup}(k, \mathcal{D})$

1: Set $N = |\{\mathcal{D}(w)\}_{w \in \mathbf{W}}|$. Let n be the number of documents $(n \leq N)$.
2: Reassign document identifiers based on a random permutation $p : [n] \rightarrow [n]$.
3: **for** each $w \in \mathbf{W}$ **do**
4: $\quad$ Compute $(\mu_0, c) \leftarrow$ **WORST-COMPRESSION**$(|\mathcal{D}(w)|, n, \mathcal{C})$.
5: $\quad$ $(\mathcal{Y}, c) \leftarrow$ **COMPRESS**$(\mathcal{D}(w), c)$.
6: $\quad$ Write $\mathcal{Y}$ as $G_1 || G_2 \ldots || G_\mu$ where G_i is a bit string of $\lambda - \lceil \log n \rceil$ bits (pad the last bit string if needed).
7: $\quad$ Set $\Gamma(w) = \{c, 1 || G_1, 2 || G_2, \ldots, \mu || G_\mu\}$.
8: $\quad$ Pad $\Gamma(w)$ to have exactly μ_0 bit-strings of the form $x || G_x$.
9: Pad Γ to have $2 \cdot N$ entries. ▷ We insert a dummy keyword which contains the necessary number of dummy values.[a]
10: $(st_\mathcal{C}, \mathcal{I}) \leftarrow \textsc{OramInitialize}(k, \Gamma)$.
11: **return** $(st_\mathcal{C}, \mathcal{I})$.

$(\mathcal{X}, st_\mathcal{C}', \mathcal{I}') \leftrightarrow \mathsf{Search}(st_\mathcal{C}, w, \mathcal{I})$

1: $i = 0$.
2: **while** $G_i \neq \perp$ **do**
3: $\quad$ $(G_i, st_\mathcal{C}', \mathcal{I}') \leftrightarrow \textsc{OramAccess}(st_\mathcal{C}, st_\mathcal{C}.pos(w||i), \mathcal{I})$.
4: $\quad$ $\mathcal{I} \leftarrow \mathcal{I}', st_\mathcal{C} \leftarrow st_\mathcal{C}', i \leftarrow i + 1$.
$\quad$ Write $result$ as $\{1||G_1, 2||G_2, \ldots, \mu||G_\mu\}$.
5: $\mathcal{X} \leftarrow$ **DECOMPRESS**$(G_1 || G_2 || \ldots || G_\mu, \mathcal{C}, c)$.
6: **return** $(\mathcal{X}, st_\mathcal{C}', \mathcal{I}')$.

$(\mathcal{I}, st_\mathcal{S}) \leftarrow \mathsf{SimSetup}(\mathcal{L}_\mathsf{SETUP}(\mathcal{D}))$

1: Let $(N, n) \leftarrow \mathcal{L}_\mathsf{SETUP}(\mathcal{D}, w)$ and compute $(\mathcal{I}, st) \leftarrow \textsc{SimOramInitialize}(2 \cdot N)$.
2: **return** $(\mathcal{I}, st)$.

$(\mathcal{X}, st_\mathcal{S}, \mathcal{I}) \leftrightarrow \mathsf{SimAccess}(st_\mathcal{S}, \mathcal{L}_\mathsf{QUERY}(\mathcal{D}, w), \mathcal{I})$

1: Let $s \leftarrow \mathcal{L}_\mathsf{QUERY}(\mathcal{D}, w), (N, n) \leftarrow \mathcal{L}_\mathsf{SETUP}(\mathcal{D}, w)$.
2: Compute $(\mu_0, c) \leftarrow$ **WORST-COMPRESSION**$(s, n, \mathcal{C})$.
3: **for** $i = 1, \ldots, \mu_0$ **do**
4: $\quad$ Pick a random index ind.
5: $\quad$ Compute $(\mathcal{X}, st, \mathcal{I}') \leftrightarrow \textsc{SimOramAccess}(st_\mathcal{S}, \mathsf{ind}, \mathcal{I})$.
6: $\quad$ Set $\mathcal{I} \leftarrow \mathcal{I}'$ and update $st_\mathcal{S}$ with the new values of st.
7: **return** $(\mathcal{X}, st_\mathcal{S}, \mathcal{I})$.

---
[a]This dummy keyword is never returned as part of any query.

Figure 5: Our OSE-K construction and the simulator algorithms **SimSetup** and **SimToken** using an Oblivious RAM and a set $\mathcal{C}$ of compression algorithms as black-box.

FastPfor, it is not. It is also possible that the worst-case compression for some algorithms to be very close to the uncompressed size, e.g. VB compression algorithm described in [17]. For compression algorithms in which computing the worst-case compression is either not viable in practice or the compression ratio is close to 1, we use an alternative methodology. We choose for each n and s a predefined μ_0, and we store the overflowed lists $(\mu - \mu_0)$ in a local stash in the client side. It is a good practice to choose μ_0 to be the expected number of compressed words (for a given n and s).

Proof of security. We will now prove the security of our OSE-K construction, assuming the black-box ORAM we use is secure and assuming that $\mu_0 > \mu$ always hold for Lines 4-8 :

THEOREM 2. *Let $\mathcal{L}_\mathsf{SETUP}$, $\mathcal{L}_\mathsf{QUERY}$ be the leakages defined in Section 2.5 for OSE-K. If the ORAM scheme used as a black-box in the construction of Figure 3 is secure, then our OSE-K construction of Figure 3 is ($\mathcal{L}_\mathsf{SETUP}$, $\mathcal{L}_\mathsf{QUERY}$) secure.*

PROOF. The deployed black-box ORAM scheme is considered to be secure, so our proof uses its SimOramInitialize and SimOramAccess. Our simulators **SimSetup**($\mathcal{L}_\mathsf{SETUP}(\mathcal{D})$) and **SimSearch**($\mathcal{L}_\mathsf{QUERY}(\mathcal{D}, w)$) are shown in Figure 5.

For the first part of the proof, we show that no PPT **Dist** algorithm exists that can distinguish, with more than negligible probability, between the index $\mathcal{I}_{real}$ and the index $\mathcal{I}_{ideal}$ since both have the same number of entries and the black-box ORAM scheme is secure. For the second part of the proof, we show that no PPT algorithm **Dist** exists

that can distinguish, with more than negligible probability, between the index Search and the index SimSearch, for the following reasons; (i) both Search and SimSearch produce indistinguishable messages, (ii) in both cases Dist observes the same number of ORAM accesses, (iii) ORAM being secure implies that Search and SimSearch are indistinguishable with non-negligible probability. $\square$

Choosing the ORAM black-box: Our OSE schemes can use any secure ORAM as a black-box. For instance, we propose using any Square Root ORAM, hierarchical-based ORAM or tree-based ORAM. The main efficiency metrics for an ORAM scheme are: (i) Amortized overhead, (ii) Worst-Case Overhead, (iii) Storage, (iv) Client Storage. In our solution we assume for simplicity reasons that the client locally stores a position map, i.e., a data structure that maintains mappings of specific keyword,id pairs to their currently stored indexes j in the ORAM. Different families of ORAM schemes stores a position map in different ways. For instance, PathORAM proposes a solution that increases the overhead and recursively outsources the position map in an oblivious manner. This work we suggest that any tree-based approach with the minimum worst-case overhead even if it stores the position map locally on the client, such as the non-recursive PathORAM, to be a good candidate for constituting the ORAM black-box. We can outsource the position maps using the notion of oblivious data structures described in [55] without increasing the worst case overhead (we create a single-linked list connecting all G_i together, each G_i will store the position of G_{i+1}, we store G_1 in an oblivious data structure and the remaining G_i in PathORAM). Creating a practical OSE scheme combining PathORAM with the idea of oblivious data structure requires in total, $O(N)$ space, $O(\log^2 N)$ worst-case overhead for result sizes smaller than $O(\log^2 N)$ ids, $O(\log N)$ worst-case overhead for result sizes greater than $O(\log^2 N)$ and client storage of $O(\log^2 N) \cdot \omega(1)$ ids. We omit providing further details on combining PathORAM with oblivious data structures, as our OSE construction is generic and can improve the search performance of an OSE scheme using any ORAM as black-box. We further refer the reader to the recent work of Chang et al. [13] for a comprehensive evaluation of various ORAM protocols.

4. EXPERIMENTS

In this section we experimentally evaluate the performance of our proposed schemes. We call the SE construction of section 3.2 as microSE and the OSE construction of section 3.3 as microOSE. We select the SE black-box scheme to be the basic construction of Cash et al. [10] as it is the state-of-the-art in-memory SE scheme with linear size encrypted index (it requires N encrypted entries) and we refer to it as $PiBas$. We did not choose the scheme with optimal read efficiency of Demertzis and Papamanthou [21] since it requires sN space; for $s = 1$ both schemes have the same performance; for $s > 1$ the optimal read efficient scheme of [21] outperforms $PiBas$ at the cost of more space. Furthermore, we denote by "microSE($PiBas$)" that microSE uses $PiBas$ as a black-box. We choose $PathORAM$ [49] to be the black-box ORAM scheme for microOSE (microOSE($PathORAM$)) and for simplicity we store the position map locally.

We evaluate the performance of microSE($PiBas$) and compare it to one of the original $PiBas$ scheme in order to illustrate the superiority of microSE; similarly we compare microOSE($PathORAM$) to $PathORAM$.

4.1 Setup

We carried out the implementation of our schemes, $PiBas$ and $PathORAM$ in Java. Our experiments were conducted on a 64-bit machine with an Intel Xeon E5-2676v3 and 64 GB RAM. We utilized the JavaX.crypto library and the bouncy castle library [1] for the cryptographic operations[7]. In particular, for the PRF and randomized symmetric encryption implementations we used HMAC-SHA-256 and AES128-CBC, respectively, for encryption. The compression algorithms that we use are $FastPfor$[37] and $EWAH$ [36] (with compressed keywords of size λ bits). We consider the following two datasets in our experimental setting. The first dataset is a real dataset [2] consisting of $6,123,276$ tuples with 22 attributes of reported incidents of crime in Chicago [2]. This is a typical database table, which does not have intersections between the keywords (database search). We consider the first query attribute to be the *location description* attribute which is an attribute following a skewed distribution containing 173 distinct keywords (this is the x-axis in Figures 7(a),and 7(b)). Among these keywords the one with minimum frequency contains 1 record, while the one with maximum frequency has $1,631,721$ records. We also consider the attribute *date* that does not follow a skewed distribution, in order to show the difference between a skewed and a "non-skewed" distribution in the database search case. The *date* attribute contains $58,404$ distinct keywords (this is the x-axis in Figures 8(a) and 8(b)). Among these keywords the one with minimum frequency contains 1 record, while the one with maximum frequency has $14,564$ records. In Figure 10(a). we provide the mean and best compression ratio for all the 22 attributes.

For our second dataset, we use the Enron email dataset [3], which consists of $30,109$ emails from the "sent mail" folder of 150 employees of the Enron corporation that were sent between $2000 - 2002$. We extracted keywords from this dataset. The words were first stemmed using the standard porter stemming algorithm [44], and then we removed 200 stop words.This dataset contains $76,577$ distinct keywords (this is the x-axis in Figures 9(a) and 9(b)). Among these keywords the one with minimum frequency contains 1 document-id, while the one with maximum has $24,642$ ids.

4.2 microSE/microOSE Evaluation

Index Costs. In the first set of experiments we evaluate the required index size and construction time of our scheme for different dataset sizes N. The results are shown in Figure 6. The construction time includes the I/O cost of transferring the dataset from the disk to the main memory, and the index size represents only the size of the encrypted index. Moreover, we partition the initial dataset into 12 sets of 500K tuples each, chosen uniformly at random from the entire

---

[7]We highlight that our Java implementation does not use *hardware supported* cryptographic operations. However, this does not affect our conclusions on the superiority of our proposed constructions. The use of hardware supported cryptographic operations will drastically improve both, the original constructions of $PiBas$ and $PathORAM$, as well as our own microSE($PiBas$) and microOSE($Path$) based constructions, thus maintaining the exact same "speed-up".
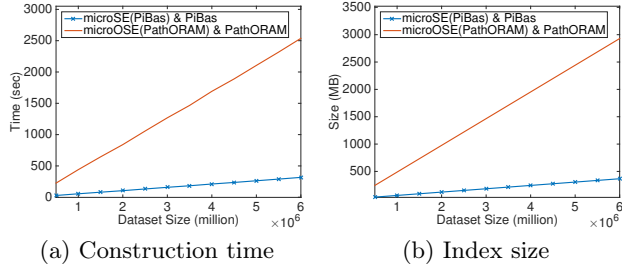
(a) Construction time   (b) Index size

Figure 6: Index costs

(a) Search Time   (b) # PRF evaluations

Figure 8: Search costs - Crime Dataset (Date attribute)

(a) Search Time   (b) # PRF evaluations

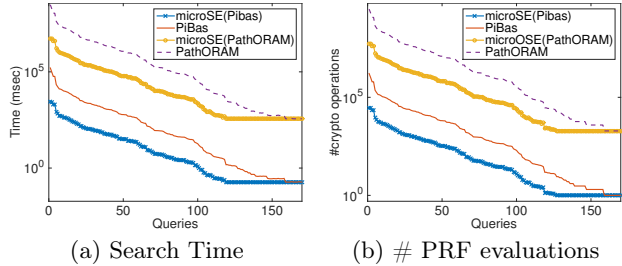Figure 7: Search costs - Crime Dataset (Location attribute)

(a) Search Time   (b) # PRF evaluations

Figure 9: Search costs - Enron Dataset

dataset. Then, we begin with the first partition and consider the other partitions in each step in order to represent the construction time (Figure 6(a)) and the index size (Figure 6(b)), as the size of the input gradually increases. Since we perform the same amount of work for every partition while building up the index, the storage and construction time required is linear in the number of partitions (or input size). Figure 6 reflects this observation. We observe that both microSE($PiBas$) and $PiBas$ have the same index costs, since microSE($PiBas$) performs padding to have exactly the same encrypted index size with $PiBas$, the same applies for microOSE($PathORAM$) and $PathORAM$. We highlight that the padding in the case of microSE affects only the setup costs since the inserted dummy records are never returned as part of any query, while in the case of microOSE it may slightly affect the query costs (depending on how we handle the overflowed lists), but the returned compressed response cannot exceed the uncompressed.

Search Cost. In this set of experiments, we illustrate the total time required by the server to retrieve and find the tuple-ids or document-ids for each query. For visualization purposes, we sort the queries based on their result size in descending order and we query each of them, i.e. x-axis for value x=0 depicts the query with the largest result size. In Figure 7(a) we observe the search time and in Figure 7(b) the number of cryptographic operations for the *location description* attribute both for microSE and microOSE. Similarly, in Figure 8(a) we observe the search time and in Figure 8(b) the number of cryptographic operations for the *date* attribute. In the case of microOSE, we calculate a specific μ_0 for a given size, by estimating heuristically its expected value for a given $n, |\mathcal{D}(w)|$ and we store the overflowed lists in a local stash γ on the client side. We experimentally observed that local stash γ was always smaller than the stash of $PathORAM$.
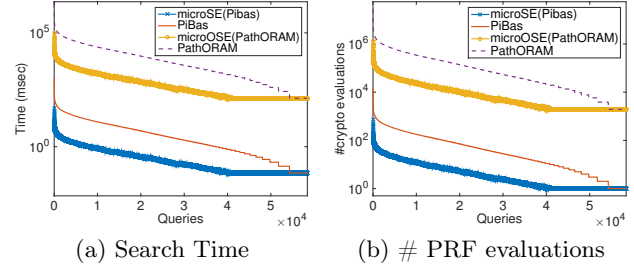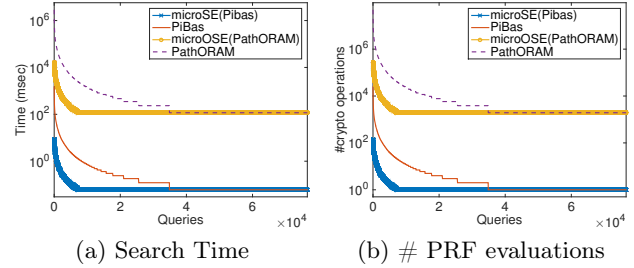
The maximum speed-up for the *location description* attribute is **62×** both for microSE and microOSE, while for the date attribute the corresponding number is **21×**. The *location description* attribute presents a more skewed distribution, as it contains high-frequency keywords. Note that more tuple-ids per keyword lead to a better compression ratio. microSE and microOSE have the same performance because in both cases they take advantage only of the size of each query.

In Figure 9(a), we observe the search time and in Figure 9(b) the number of PRF evaluations for the Enron dataset. In the case of SE we achieve up to **188×** speed-up, while in the case of OSE the speed-up was similar since in both case the compression takes advantage of the total number of documents and the size of each query.

In Figure 10(a), we use microSE for all the 22 attributes of the Crime dataset and we report the best and the mean speed-up that we achieve. We observe that for the attributes $1, 2$ the compression ratio is 1, which means that no compression is achieved. The reason is that the first 2 attributes contain unique values, so every value has result size 1, which is the minimum number of cryptographic operation that we have to perform. We also observe that attributes $8, 9$ achieve higher compression ratio than the other attributes (up to **166×**, **203×** respectively); the reason is that these are binary attributes (true or false) and the sizes of their values are proportional to the database size (attribute 8: whether an arrest was made or not, attribute 9: whether the incident was domestic-related or not).

Dynamic costs. In this set of experiments, we consider the case of dynamic microSE which is addressed as described in [19]. For these experiments, we fix the consolidation step s that is described in the original paper, to 2. This means that after every 2 new indexes, we initiate a consolidation phase that merges one or more indexes in order to construct a new one. The batch size is set to $100,000$ updates. Figure 10(b)
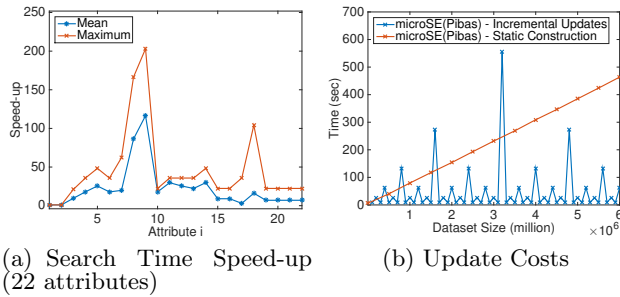
(a) Search Time Speed-up (22 attributes)

(b) Update Costs

Figure 10: Additional Experiments (Crime Dataset)

plots the time required for dynamic microSE($Pibas$) (labeled as "Incremental Updates") to maintain the index, when considering 10Mbps network bandwidth. This experiment includes the time required for downloading, decrypting, reconstructing (merging), re-encrypting and uploading the indexes. As a reference, the plot also includes the cost required by static microSE($Pibas$) to build the same index (including the time for uploading the index), assuming that the whole dataset is made available at once (labeled as "Static Construction"). We can also use the same approach in order to extend microOSE to the dynamic setting; the update costs will follow the same pattern but they will be scaled by a constant factor.

5. CONCLUSIONS

In this work we propose new searchable encryption schemes to address a new efficiency dimension, namely the *number of cryptographic operations* required to retrieve the result of a query. We present two new schemes, microSE and microOSE, that use compression techniques in order to reduce the size of the plaintext indexes before producing the encrypted searchable indexes using any SE/OSE as a black-box. Our schemes achieve significant savings in search time both for private database search and keyword search, while at the same time any future advances in the active research area of SE and ORAM can be readily incorporated into our proposed constructions.

6. ACKNOWLEDGEMENTS

7. REFERENCES

[1] Bouncy castle. http://www.bouncycastle.org.
[2] Crimes 2001 to present (city of chicago). https://data.cityofchicago.org/public-safety/crimes-2001-to-present/ijzp-q8t2.
[3] Enron email dataset. https://www.cs.cmu.edu/enron/.
[4] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *CIDR*, 2013.
[5] G. Asharov, T. H. Chan, K. Nayak, R. Pass, L. Ren, and E. Shi. Oblivious computation with data locality. *ePrint IACR*, 2017.
[6] G. Asharov, M. Naor, G. Segev, and I. Shahaf. Searchable Symmetric Encryption: Optimal Locality in Linear Space via Two-Dimensional Balanced Allocations. In *STOC, 2016.*
[7] G. Asharov, G. Segev, and I. Shahaf. Tight tradeoff s in searchable symmetric encryption. CRYPTO, 2018.
[8] S. Bajaj and R. Sion. Trusteddb: a trusted hardware based database with privacy and data confidentiality. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 205–216. ACM, 2011.
[9] R. Bost. Sofos: Forward Secure Searchable Encryption. In *CCS*, 2016.
[10] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *NDSS*, 2014.
[11] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *CRYPTO*, 2013.
[12] D. Cash and S. Tessaro. The Locality of Searchable Symmetric Encryption. In *EUROCRYPT, 2014.*
[13] Z. Chang, D. Xie, and F. Li. Oblivious ram: a dissection and experimental evaluation. *PVLDB*, 9(12):1113–1124, 2016.
[14] M. Chase and S. Kamara. Structured Encryption and Controlled Disclosure. In *ASIACRYPT*, 2010.
[15] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. *Journal of Computer Security, 2011.*
[16] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In *CCS*, 2006.
[17] D. Cutting and J. Pedersen. Optimization for dynamic inverted index maintenance. In *SIGIR*, 1989.
[18] I. Demertzis, D. Papadopoulos, and C. Papamanthou. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. *CRYPTO*, 2018.
[19] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis. Practical Private Range Search Revisited. In *SIGMOD*, 2016.
[20] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, M. Garofalakis, and C. Papamanthou. Practical private range search in depth. *TODS*, 2018.
[21] I. Demertzis and C. Papamanthou. Fast searchable encryption with tunable locality. In *SIGMOD*, 2017.
[22] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich Queries on Encrypted Data: Beyond Exact Matches. In *ESORICS*, 2015.
[23] C. Gentry. *A Fully Homomorphic Encryption Scheme.* PhD thesis, Stanford University, 2009.
[24] C. Gentry. Computing Arbitrary Functions of Encrypted Data. *Commun. of the ACM*, 2010.
[25] E.-J. Goh et al. Secure Indexes. *IACR Cryptology ePrint Archive*, 2003.
[26] O. Goldreich, S. Goldwasser, and S. Micali. How to Construct Random Functions. *J. ACM*, 33(4):792–807, 1986.

[27] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM, 1996.*

[28] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *ICALP*, 2011.

[29] S. Kamara and T. Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *EUROCRYPT*, 2017.

[30] S. Kamara and C. Papamanthou. Parallel and Dynamic Searchable Symmetric Encryption. In *FC*, 2013.

[31] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic Searchable Symmetric Encryption. In *CCS*, 2012.

[32] J. Katz and Y. Lindell. *Introduction to modern cryptography.* CRC press, 2014.

[33] J. Kelley and R. Tamassia. Secure compression: Theory and practice. *IACR Cryptology ePrint Archive*, 2014, 2014.

[34] K. Kurosawa and Y. Ohtaki. UC-Secure Searchable Symmetric Encryption. In *FC, 2012.*

[35] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *SIAM*, 2012.

[36] Lemire. javaewah, Dec 2016.

[37] Lemire. Javafastpfor, Apr 2017.

[38] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1âĂŞ29, 2013.

[39] D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *Data & Knowledge Engineering*, 69(1):3–28, 2010.

[40] I. Miers and P. Mohassel. IO-DSSE: Scaling Dynamic Searchable Encryption to Millions of Indexes By Improving Locality. In *NDSS*, 2017.

[41] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In *Oblix: An Efficient Oblivious Search Index.* SP, 2018.

[42] B. Pinkas and T. Reinman. Oblivious RAM Revisited. In *CRYPTO.* 2010.

[43] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *SOSP*, 2011.

[44] M. Porter. An algorithm for suffix stripping. *Program*, 14(3):130âĂŞ137, 1980.

[45] C. Priebe, K. Vaswani, and M. Costa. Enclavedb: A secure database using sgx. SP, 2018.

[46] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with o ((logn) 3) worst-case cost. In *AsiaCrypt*, 2011.

[47] D. X. Song, D. Wagner, and A. Perrig. Practical Techniques for Searches on Encrypted Data. In *SP*, 2000.

[48] E. Stefanov, C. Papamanthou, and E. Shi. Practical Dynamic Searchable Encryption with Small Leakage. In *NDSS*, 2014.

[49] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path Oram: An Extremely Simple Oblivious Ram Protocol. In *CCS*, 2013.

[50] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *PVLDB*, 6(5):289–300, 2013.

[51] P. Van Liesdonk, S. Sedghi, J. Doumen, P. Hartel, and W. Jonker. Computationally Efficient Searchable Symmetric Encryption. In *SDM*. 2010.

[52] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson. An experimental study of bitmap compression vs. inverted list compression. *Proceedings of the 2017 ACM International Conference on Management of Data - SIGMOD '17*, 2017.

[53] X. Wang, H. Chan, and E. Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *CCS*, 2015.

[54] X. S. Wang, Y. Huang, T. H. Chan, A. Shelat, and E. Shi. Scoram: oblivious ram for secure computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 191–202. ACM, 2014.

[55] X. S. Wang, K. Nayak, C. Liu, T. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *CCS*, 2014.

[56] K. Wu, E. J. Otoo, A. Shoshani, and H. Nordberg. Notes on design and implementation of compressed bit vectors. Technical report, Technical Report LBNL/PUB-3161, Lawrence Berkeley National Laboratory, Berkeley, CA, 2001.

[57] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, 2017.

[58] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. *22nd International Conference on Data Engineering (ICDE'06)*, 2006.