Tight Tradeoffs in Searchable Symmetric Encryption

Gilad Asharov[1], Gil Segev[2], and Ido Shahaf[2(✉)]

[1] Cornell Tech, New York, NY, USA
asharov@cornell.edu
[2] School of Computer Science and Engineering,
Hebrew University of Jerusalem, 91904 Jerusalem, Israel
{ido.shahaf,segev}@cs.huji.ac.il

Abstract. A searchable symmetric encryption (SSE) scheme enables a client to store data on an untrusted server while supporting keyword searches in a secure manner. Recent experiments have indicated that the practical relevance of such schemes heavily relies on the tradeoff between their *space overhead*, *locality* (the number of non-contiguous memory locations that the server accesses with each query), and *read efficiency* (the ratio between the number of bits the server reads with each query and the actual size of the answer). These experiments motivated Cash and Tessaro (EUROCRYPT '14) and Asharov et al. (STOC '16) to construct SSE schemes offering various such tradeoffs, and to prove lower bounds for natural SSE frameworks. Unfortunately, the best-possible tradeoff has not been identified, and there are substantial gaps between the existing schemes and lower bounds, indicating that a better understanding of SSE is needed.

We establish tight bounds on the tradeoff between the space overhead, locality and read efficiency of SSE schemes within two general frameworks that capture the memory access pattern underlying all existing schemes. First, we introduce the "pad-and-split" framework, refining that of Cash and Tessaro while still capturing the same existing schemes. Within our framework we significantly strengthen their lower bound, proving that any scheme with locality L must use space $\Omega(N \log N / \log L)$ for databases of size N. This is a tight lower bound, matching the tradeoff provided by the scheme of Demertzis and Papamanthou (SIGMOD '17) which is captured by our pad-and-split framework.

Then, within the "statistical-independence" framework of Asharov et al. we show that their lower bound is essentially tight: We construct a scheme whose tradeoff matches their lower bound within an additive $O(\log \log \log N)$ factor in its read efficiency, once again improving

G. Asharov—Supported by a Junior Fellow award from the Simons Foundation.
G. Segev and I. Shahaf—Supported by the European Union's Horizon 2020 Framework Program (H2020) via an ERC Grant (Grant No. 714253), by the Israel Science Foundation (Grant No. 483/13), by the Israeli Centers of Research Excellence (I-CORE) Program (Center No. 4/11), and by the US-Israel Binational Science Foundation (Grant No. 2014632).

H. Shacham and A. Boldyreva (Eds.): CRYPTO 2018, LNCS 10991, pp. 407–436, 2018.
https://doi.org/10.1007/978-3-319-96884-1_14

upon the existing schemes. Our scheme offers optimal space and locality, and nearly-optimal read efficiency that depends on the frequency of the queried keywords: For a keyword that is associated with $n = N^{1-\epsilon(n)}$ document identifiers, the read efficiency is $\omega(1) \cdot \epsilon(n)^{-1} + O(\log\log\log N)$ when retrieving its identifiers (where the $\omega(1)$ term may be arbitrarily small, and $\omega(1) \cdot \epsilon(n)^{-1}$ is the lower bound proved by Asharov et al.). In particular, for any keyword that is associated with at most $N^{1-1/o(\log\log\log N)}$ document identifiers (i.e., for any keyword that is not exceptionally common), we provide read efficiency $O(\log\log\log N)$ when retrieving its identifiers.

1 Introduction

A searchable symmetric encryption (SSE) scheme [11,27] enables a client to store data on an untrusted server and later perform keyword searches: Given a keyword w, the client should be able to retrieve all data items that are associated with w (e.g., all document identifiers that contain w). This typically consists of a two-stage process: First, the client encrypts her database and uploads it to the server, and then the client repeatedly queries the server with various keywords by providing the server with keyword-specific search tokens. Informally, the security requirement of SSE schemes asks that the server does not learn any information about keywords for which the client did not issue any queries.

The practical relevance of SSE schemes. Motivated by the increasingly-growing technological interest in outsourcing data to remote (and thus potentially untrusted) servers, a very fruitful line of research in the cryptography community focused on the design of SSE schemes (e.g., [2,5–12,14,19,20,22,23,27,29]). Most of the proposed schemes offer strong and meaningful notions of security, and some even extend the basic keyword search functionality to more expressive ones.

Despite these promising developments, Cash et al. [7] showed via experiments with real-world databases that the practical performance of the known schemes is quite disappointing, and scales badly to large databases. Somewhat surprisingly, they observed that performance issues resulting from impractical memory layouts may be significantly more crucial compared to performance issues resulting from the cryptographic processing of the data. More specifically, Cash et al. observed that schemes with poor *locality* (i.e., schemes in which the server has to access a rather large number of *non-contiguous* memory locations with each query) have poor practical performance when dealing with large databases that require the usage of disk-storage mechanisms.

Practical locality, however, is obviously insufficient: Any practically-relevant SSE scheme should (at least) not suffer from either a significant *space overhead* (i.e., encrypted databases should not be much larger than the original databases),

or from a poor *read efficiency* (i.e., servers should not read much more data than needed for answering each query)[1].

Efficiency tradeoffs and existing lower bounds. This state of affairs naturally poses the challenge of constructing an SSE scheme that simultaneously enjoys asymptotically-optimal space overhead, locality, and read efficiency – but unfortunately no such scheme is currently known. This has motivated Cash and Tessaro [8] to initiate the study of understanding the tradeoff between these central measures of efficiency. They proved a lower bound showing that, for a large and natural class of SSE schemes, it is in fact impossible to simultaneously enjoy asymptotically-optimal space overhead, locality, and read efficiency. Specifically, they considered the class of SSE schemes with "non-overlapping reads": Schemes in which distinct keywords induce non-overlapping memory regions which the server may access upon their respective queries (we refer the reader to the work of Cash and Tessaro [8] for a formal definition of their notion of non-overlapping reads).

The class of SSE schemes with non-overlapping reads captures the basic techniques underlying all existing SSE schemes other than two schemes proposed by Asharov et al. [2]. These two schemes may have arbitrary overlapping reads, and offer an improved tradeoff between their space overhead, locality, and read efficiency compared to the previously suggested schemes. This tradeoff, however, is still non-optimal, and Asharov et al. showed that this is in fact inherent to their approach. Similarly to Cash and Tessaro, they proved that also for a different class of SSE schemes, it is impossible to simultaneously enjoy asymptotically-optimal space overhead, locality, and read efficiency. Specifically, they considered the class of SSE scheme with "statistically-independent reads": Schemes in which distinct keywords induce statistically-independent memory regions which the server accesses upon their respective queries.

The lower bounds proved by Cash and Tessaro and by Asharov et al. capture all of the existing SSE schemes (except for various schemes with non-standard leakage or functionality that we do not consider in this work). That is, the basic techniques underlying each of the known SSE schemes belong either to the class of "non-overlapping reads" or to the class of "statistically-independent reads". In both cases, however, the existing lower bounds are not tight, as there are still noticeable gaps between the lower bounds and the performance guarantees of the existing schemes (as we detail in the next section). This unsatisfying situation calls for obtaining a better understanding of SSE techniques: Either by strengthening the known lower bounds, or by designing new schemes with better performance guarantees.

[1] We consider the notions of locality and read efficiency as formalized by Cash and Tessaro [8]: The locality of a scheme is the number of non-contiguous memory accesses that the server performs with each query, and the read efficiency of a scheme is the ratio between the number of bits the server reads with each query and the actual size of the answer. We refer the reader to Sect. 2.1 for the formal definitions.

1.1 Our Contributions

We prove tight bounds on the tradeoff between the space overhead, locality, and read efficiency of SSE schemes within the following two general frameworks:

The pad-and-split framework: We formalize a framework that refines the non-overlapping reads framework of Cash and Tessaro [8] while still capturing the same existing SSE schemes (i.e., all existing schemes other than those of Asharov et al. [2])[2]. We refer to this framework as the "pad-and-split" framework given the structure of the SSE schemes that it captures.

Within this framework we significantly strengthen the lower bound of Cash and Tessaro: We show that any pad-and-split scheme with locality L must use space $\Omega\left(N \cdot \log N / \log L\right)$ for databases of size N. For example, for any constant locality (i.e., $L = O(1)$) and for any logarithmic locality (i.e., $L = O(\log N)$) our lower bound shows that any such scheme must use space $\Omega(N \log N)$ and $\Omega(N \log N / \log \log N)$, respectively, and is thus not likely to be of substantial practical relevance (whereas the lower bound of Cash and Tessaro would only yield space $\omega(N)$ when the locality is constant).

Then, we observe that our lower bound is in fact tight, as it is matched by a recent scheme proposed by Demertzis and Papamanthou [14] that is captured by our framework (i.e., their scheme is an optimal instantiation of our framework). We refer the reader to Sects. 1.2 and 3 for a high-level overview and for a detailed description of this framework, its instantiations, and of our lower bound, respectively.

The statistical-independence framework: We consider the statistical-independence framework of Asharov et al. [2], and show that their lower bound for SSE schemes in this framework is essentially tight: Based on the existence of any one-way function, we construct a scheme whose efficiency guarantees match their lower bound for constant locality within an additive $O(\log \log \log N)$ factor in the read efficiency, and improve upon those of their two schemes.

Specifically, for databases of size N, our scheme offers both optimal space and optimal locality (i.e., space $O(N)$ and locality $O(1)$), and comes very close to offering optimal read efficiency as well. The read efficiency of our scheme when querying for a keyword w depends on the length of the list $\mathsf{DB}(w)$ that is associated with w (that is, the read efficiency depends on the number of identifiers that are associated with w).[3] When querying for a keyword that is associated with $n = N^{1-\epsilon(n)}$ identifiers, the read efficiency of our scheme is $f(N) \cdot \epsilon(n)^{-1} + O(\log \log \log N)$, where $f(N) = \omega(1)$ may be any pre-determined function, and $\omega(1) \cdot \epsilon(n)^{-1}$ is a lower bound as proved by

[2] Each of the schemes that are captured by our framework offers other important implementation details, improvements and optimizations that we do not intend to capture, since these are not directly related to the tradeoff between space, locality, and read efficiency.

[3] We emphasize that this does not hurt the security of SSE schemes, and still results in minimal leakage as required.

Asharov et al. [2]. In particular, for any keyword that is associated with at most $N^{1-1/o(\log\log\log N)}$ identifiers (i.e., for any keyword that is not exceptionally common), the read efficiency of our scheme when retrieving its identifiers is $O(\log\log\log N)$. We refer the reader to Sects. 1.2 and 4 for a high-level overview and for a detailed description of this framework and of our new scheme, respectively.

Our results in the pad-and-split and statistical-independence frameworks, which are summarized in Table 1 and presented in more detail in Sect. 1.2, show a significant gap between the performance guarantees that can be offered within these two frameworks. In both frameworks we establish tight bounds that capture the basic techniques underlying all of the existing SSE schemes. Thus, any attempt to further improve upon the tradeoff between the space overhead, locality and read efficiency of our schemes must be based on new techniques that deviate from all known SSE schemes.

Table 1. A summary of our contributions. We denote by N the size of the database. The read efficiency in the lower bound of Asharov et al. [2] and in our statistical-independence scheme (Theorem 1.2) when querying for a keyword w depends on the number $n = N^{1-\epsilon(n)}$ of identifiers that are associated with w. In addition, our statistical-independence scheme is based on the modest assumption that no keyword is associated with more than $N/\log^3 N$ identifiers, whereas the scheme of Asharov et al. [2] is based on the stronger assumption that no keyword is associated with more than $N^{1-1/\log\log N}$ identifiers (thus, the read efficiency of their scheme does not contradict their lower bound, and our scheme has better read efficiency compared to their scheme). Finally, we note that the $\omega(1)$ term in the read efficiency of our scheme can be set to any super-constant function (e.g., $\log\log\log\log N$).

	Space	Locality	Read Efficiency
This work (Theorem 1.1)**:** Pad-and-split lower bound	$\Omega(N\log N/\log L)$	L	$O(1)$
[14]: Pad-and-split scheme	$O(N\log N/\log L)$	L	$O(1)$
[2]: Statistical-independence lower bound	$O(N)$	$O(1)$	$\omega(1)\cdot\epsilon(n)^{-1}$
[2]: Statistical-independence scheme	$O(N)$	$O(1)$	$\tilde{O}(\log\log N)$
This work (Theorem 1.2)**:** Statistical-independence scheme	$O(N)$	$O(1)$	$\omega(1)\cdot\epsilon(n)^{-1}+$ $O(\log\log\log N)$

1.2 Overview of Our Contributions

In this section we provide an overview of the two frameworks that we consider in this work, and present our results within each framework. As standard in the line of research on searchable symmetric encryption, we represent a database as a collection $\mathsf{DB} = \{\mathsf{DB}(w_1),\ldots,\mathsf{DB}(w_{n_\mathsf{W}})\}$, where $w_1,\ldots,w_{n_\mathsf{W}}$ are distinct keywords, and $\mathsf{DB}(w)$ is the list of all identifiers that are associated with each keyword w. We denote by $N = \sum_{i=1}^{n_\mathsf{W}} |\mathsf{DB}(w_i)|$ the size of the database.

Our pad-and-split framework. Our pad-and-split framework considers schemes that are characterized by an algorithm denoted $\mathsf{SplitList}$ and consist of two phases. In the first phase, given a database $\mathsf{DB} = \{\mathsf{DB}(w_1), \ldots, \mathsf{DB}(w_{n_\mathsf{W}})\}$ of size N, for each keyword w_i the scheme invokes the $\mathsf{SplitList}$ algorithm on the length n_i of its corresponding list $\mathsf{DB}(w_i)$, to obtain a vector $(x_i^{(1)}, \ldots, x_i^{(m)})$ of integers. The scheme then potentially pads the list $\mathsf{DB}(w_i)$ by adding "dummy" elements, and splits the padded list into sublists of lengths $\mathsf{len}^{(1)}, \ldots, \mathsf{len}^{(m)}$, where $x_i^{(j)}$ denotes the number of sublists of each length $\mathsf{len}^{(j)}$. Then, in the second phase, for each possible length $\mathsf{len}^{(j)}$, the scheme groups together all sublists of length $\mathsf{len}^{(j)}$, and independently processes each such group to produce an encrypted database EDB.

We consider any possible instantiation of the $\mathsf{SplitList}$ algorithm (satisfying the necessary requirement that no list is longer than the sum of lengths of its sublists), and this enables us to describe a general template for constructing an SSE scheme based on any such algorithm given any one-way function. Our template yields schemes whose space usage and locality are essentially inherited from similar properties of their underlying $\mathsf{SplitList}$ algorithm, and whose read efficiency is always constant. We then demonstrate that this template captures the memory access patterns underlying essentially all existing schemes other than those of Asharov et al. [2]. Specifically, we show that each of these schemes can be obtained as an instantiation of our template using a suitable $\mathsf{SplitList}$ algorithm.

A tight lower bound for pad-and-split schemes. Equipped with our general notion of pad-and-split schemes, we prove a lower bound on the asymptotic efficiency guarantees of such schemes. Whereas the lower bound of Cash and Tessaro [8] states that SSE schemes with non-overlapping reads cannot simultaneously offer asymptotically-optimal space overhead and locality, we prove the following lower bound (capturing the same existing schemes) stating that the efficiency guarantees of pad-and-split schemes must in fact be very far from optimal:

Theorem 1.1. *Any pad-and-split SSE scheme for databases of size N with locality $L = L(N)$ uses space $\Omega(N \log N / \log L)$.*

We show that this lower bound is tight, as it matches the tradeoff offered by the scheme of Demertzis and Papamanthou [14] (i.e., their scheme is an optimal instantiation of our framework). We refer the reader to Sect. 3 for a detailed and more formal presentation of our results, including an in-depth discussion of the existing pad-and-split instantiations.

The statistical-independence framework. The statistical-independence framework of Asharov et al. [2] considers symmetric searchable encryption schemes that are characterized by a pair of algorithms, denoted $\mathsf{RangesGen}$ and $\mathsf{Allocation}$, and consist of two phases. In the first phase, given a database $\mathsf{DB} = \{\mathsf{DB}(w_1), \ldots, \mathsf{DB}(w_{n_\mathsf{W}})\}$ of size N, for each keyword w_i the scheme invokes the $\mathsf{RangesGen}$ algorithm on the length n_i of its corresponding list $\mathsf{DB}(w_i)$,

to obtain a set of *possible locations* in which the scheme may place the elements of the list $\mathsf{DB}(w_i)$.[4] Then, in the second phase, given the sets of possible locations for all keywords, the scheme invokes the $\mathsf{Allocation}$ algorithm on these sets to obtain the *actual locations* for the corresponding lists. A key property of this framework is that the $\mathsf{RangesGen}$ algorithm, which determines the set of possible locations for each list $\mathsf{DB}(w_i)$, is applied separately and independently to the length of each list. Thus, the possible locations of each list are independent of the possible locations of all other lists (in contrast, the actual locations of the lists are naturally correlated).

Asharov et al. referred to a pair $(\mathsf{RangesGen}, \mathsf{Allocation})$ of such algorithms as an allocation scheme, and showed that any such allocation scheme can be used to construct an SSE scheme. Then, by constructing two allocation schemes they obtained two SSE schemes with space $O(N)$ and locality $O(1)$. Without making any assumptions on the structure of the database, their first scheme has read efficiency $\tilde{O}(\log N)$, and under the assumption that no keyword is associated with more than $N^{1-1/\log\log N}$ identifiers, their second scheme has read efficiency $\tilde{O}(\log\log N)$.

Our leveled two-choice scheme. Within the statistical-independence framework, as discussed above, we construct a scheme whose tradeoff between space, locality, and read efficiency matches the lower bound proved by Asharov et al. for scheme in this framework to within an additive $O(\log\log\log N)$ factor in its read efficiency (see Sect. 4 for a formal statement of their lower bound).

Specifically, we construct a scheme whose read efficiency when querying for a keyword w depends on the length of the list $\mathsf{DB}(w)$ that is associated with w (that is, the read efficiency depends on the number of identifiers that are associated with w). For any $n \leq N$ we denote by $\mathsf{r}(N, n)$ the read efficiency when retrieving a list of length n, and prove the following theorem:

Theorem 1.2. *Assuming the existence of any one-way function, for any function $f(N) = \omega(1)$ there exists an adaptively-secure symmetric searchable encryption scheme for databases of size N in which no keyword is associated with more than $N/\log^3 N$ identifiers, with the following guarantees:*

- *Space* $O(N)$.
- *Locality* $O(1)$.
- *Read efficiency* $\mathsf{r}(N, n) = f(N) \cdot \epsilon(n)^{-1} + O(\log\log\log N)$, *where* $n = N^{1-\epsilon(n)}$.
- *Token size* $O(1)$.

Our construction applies to databases of size N under the modest assumption that no keyword is associated with more than $N/\log^3 N$ identifiers (note that the construction of Asharov et al. [2] is based on the stronger assumption that no keyword is associated with more than $N^{1-1/\log\log N}$ identifiers). One can always

[4] Looking ahead, when supplied with a token corresponding to a keyword w_i, the server will return to the client all data stored in the possible locations of the list $\mathsf{DB}(w_i)$ (the server will not actually know in which of the possible locations the elements of the list are actually placed).

generically deal (in a secure manner) with such extremely-common keywords by first excluding them from the database and applying our proposed scheme, and then applying in addition any other scheme for these extremely-common keywords (e.g., the "one-choice scheme" of Asharov et al. [2] or the recent scheme of Demertzis et al. [13] – see Sect. 1.3 for more details).

When comparing our scheme to the scheme of Asharov et al. (see Table 1), both schemes offer space $O(N)$ and locality $O(1)$, where the read efficiency of our scheme is strictly better than the read efficiency of their scheme – see Fig. 1. In particular, for any keyword that is not exceptionally frequent (specifically, associated with at most $N^{1-1/o(\log\log\log N)}$ identifiers), our scheme provides read efficiency $O(\log\log\log N)$ whereas their scheme provides read efficiency $\tilde{O}(\log\log N)$.

The structure of our scheme. Our scheme is a *leveled* generalization of the "two-choice" scheme of Asharov et al. and consists of three levels for storing the elements of a given database. The first level consists of the two-choice SSE scheme of Asharov et al. but with *an exponentially improved read efficiency*. Our key observation is that when viewing the first level as a collection of "bins", then by allowing a few elements to "overflow" we can reduce the maximal load of each bin from $\tilde{O}(\log\log N)$ (as in [2]) to $O(\log\log\log N)$ and also handle much longer lists (i.e., much more frequent keywords). This then translates into improving the read efficiency in this level from $\tilde{O}(\log\log N)$ to $O(\log\log\log N)$, while still using space $O(N)$ and locality $O(1)$.

At this point, however, we have to store the overflowing elements. We store the vast majority of these elements in our second level, which consists of roughly $\log N$ cuckoo hashing tables [26], where the j hash table is designed to store at most $\widehat{N}/2^j$ values each of which of size 2^j. Our specific choice of cuckoo hashing as a static dictionary (i.e., a hash table) is due to its specific properties that guarantee the security of our scheme (see Sect. 2.3 for a discussion of these specific properties). In particular, our third level consists of a cuckoo hashing *stash* for each of the second-level cuckoo hashing tables. The goal of introducing this level is to reduce the failure probably of cuckoo hashing from noticeable to negligible, which is essential for the security of our resulting SSE scheme. We refer the reader to Sect. 4 for a detailed description of our scheme.

1.3 Related Work

The notion of searchable symmetric encryption was put forward by Song et al. [27] who suggested several practical constructions. Formal notions of security and functionality for SSE, as well as the first constructions satisfying them, were later provided by Curtmola et al. [11,12]. Additional work in this line of research developed searchable symmetric encryption schemes with various efficiency properties, support for data updates, authenticity, support for more advanced searches, and more (see [2,5–12,14,16,19,20,22,23,27,29] and the references therein). The two frameworks that we consider in this work capture schemes that satisfy that standard notions of SSE introduced by Curtmola et al. [11,12]. These schemes are

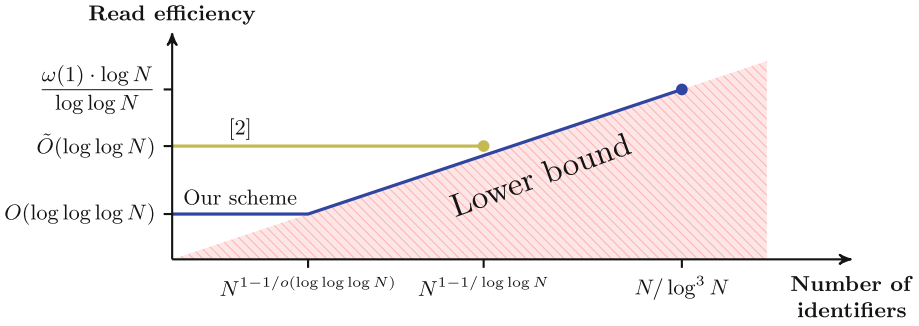

Fig. 1. The read efficiency of our statistical-independence scheme compared to that of Asharov et al. [2] and to the lower bound. The read efficiency of our scheme is depicted by the blue line, and the read efficiency of the scheme of Asharov et al. is depicted by the yellow line (recall that our scheme supports keywords that are associated with up to $N/\log^3 N$ identifiers, whereas the scheme of Asharov et al. only supports keywords that are associated with at most $N^{1-1/\log\log N}$ identifiers). The read efficiency lower bound of Asharov et al. is depicted by the red triangle (note that it coincides with our blue line for keywords that are associated with at least $N^{1-1/o(\log\log\log N)}$ and at most $N/\log^3 N$ identifiers). In all three cases the read efficiency is presented as a function of the number of identifiers that are associated with the queried keyword. (Color figure online)

discussed in Sect. 3.2 as instantiations of our pad-and-split framework, and in Sect. 4.2 as instantiations of the statistical-independence framework of Asharov et al. [2].

Our statistical-independence scheme can be applied to any database in which no keyword is associated with more than $N/\log^3 N$ identifiers. As discussed above, one can always generically deal (in a secure manner) with such extremely-frequent keywords by first excluding them from the database and applying our proposed scheme, and then applying in addition any other scheme for these extremely-common keywords. For example, for these keywords one can apply the "one-choice scheme" of Asharov et al. or the recent scheme of Demertzis, Papadopoulos and Papamanthou [13] that provides a sub-logarithmic read efficiency when searching for extremely frequent keywords[5]. Specifically, Demertzis et al. proposed a scheme that handles such extremely frequent keywords and improves their read efficiency from $\tilde{O}(\log N)$ as guaranteed by the "one-choice scheme" of Asharov et al. to $O(\log^{2/3+\delta} N)$ for any fixed constant $\delta > 0$ (for all other keywords they use the two schemes of Asharov et al., which can now be replaced by our new scheme in its appropriate range of parameters).

[5] The scheme of Demertzis et al. [13] is not captured by the two frameworks we consider in this work, as it requires the server to modify its stored data (i.e., the encrypted database) and the user to update her local state whenever a search query is issued.

1.4 Paper Organization

The remainder of this paper is organized as follows. In Sect. 2 we review the standard notion of symmetric searchable encryption schemes, as well as various tools that are used in our constructions. Then, in Sect. 3 we put forward our pad-and-split framework and then present our lower bound and new scheme in this framework. In Sect. 4 we review the statistical-independence framework and then present our new scheme in this framework.

2 Preliminaries

In this section we present the notation, definitions, and basic tools that are used in this work. We denote by $\lambda \in \mathbb{N}$ the security parameter. For a distribution X we denote by $x \leftarrow X$ the process of sampling a value x from the distribution X. Similarly, for a set $\mathcal{X}$ we denote by $x \leftarrow \mathcal{X}$ the process of sampling a value x from the uniform distribution over $\mathcal{X}$. For an integer $n \in \mathbb{N}$ we denote by $[n]$ the set $\{1, \ldots, n\}$. A function $\mathsf{negl} : \mathbb{N} \to \mathbb{R}^+$ is $\mathsf{negligible}$ if for every constant $c > 0$ there exists an integer N_c such that $\mathsf{negl}(n) < n^{-c}$ for all $n > N_c$. All logarithms in this paper are to the base of 2.

2.1 Searchable Symmetric Encryption

Let $\mathsf{W} = \{w_1, \ldots, w_{n_\mathsf{W}}\}$ denote a set of keywords, where each keyword w_i is associated with a list $\mathsf{DB}(w_i) = \{\mathsf{id}_1, \ldots, \mathsf{id}_{n_i}\}$ of document identifiers (these may correspond, for example, to documents in which the keyword w_i appears). A database $\mathsf{DB} = \{\mathsf{DB}(w_1), \ldots, \mathsf{DB}(w_{n_\mathsf{W}})\}$ consists of several such lists. We assume that each keyword and document identifier can be represented using a constant number of machine words, each of length $O(\lambda)$ bits, in the unit-cost RAM model[6]. There are various different syntaxes for SSE schemes in the literature, where the main differences are in the flavor of interaction between the server and the client with each query. In this work we consider both a setting where the server decrypts the set of identifiers by itself, and a setting where the server does not decrypt this but rather sends encrypted data back to the client (who can then decrypt and learn the set of identifiers).

Functionality

A searchable symmetric encryption scheme is a 5-tuple $(\mathsf{KeyGen}, \mathsf{EDBSetup}, \mathsf{TokGen}, \mathsf{Search}, \mathsf{Resolve})$ of probabilistic polynomial-time algorithms satisfying the following requirements:

- The key-generation algorithm KeyGen takes as input the security parameter $\lambda \in \mathbb{N}$ in unary representation and outputs a secret key K.

[6] The unit cost word-RAM model is considered the standard model for analyzing the efficiency of data structures (see, for example, [15,17,18,24,25] and the references therein).

- The database setup $\mathsf{EDBSetup}$ algorithm takes as input a secret key K and a database DB, and outputs an encrypted database EDB.
- The token-generation algorithm TokGen takes as input a secret key K and a keyword w, and outputs a token τ and some internal state ρ.
- The search algorithm Search takes as input a token τ and an encrypted database EDB, and outputs a list R of results.
- The resolve algorithm $\mathsf{Resolve}$ takes as input a list R of results and an internal state ρ, and outputs a list M of document identifiers.

An SSE scheme for databases of size $N = N(\lambda)$ is correct if for any database DB of size N and for any keyword w, with an overwhelming probability in the security parameter $\lambda \in \mathbb{N}$, it holds that $M = \mathsf{DB}(w)$ at the end of the following experiment:

1. $K \leftarrow \mathsf{KeyGen}(1^\lambda)$.
2. $\mathsf{EDB} \leftarrow \mathsf{EDBSetup}(K, \mathsf{DB})$.
3. $(\tau, \rho) \leftarrow \mathsf{TokGen}(K, w)$.
4. $R \leftarrow \mathsf{Search}(\tau, \mathsf{EDB})$.
5. $M = \mathsf{Resolve}(\rho, R)$.

We note that one can also consider a more adversarially-flavored notion of correctness, where an adversary adaptively interacts with a server with the goal of producing a query that results in an incorrect output. We refer the reader to [2] for more details, and here we only point out that our schemes in this paper satisfy such a notion as well.

Efficiency Measures

Our notions of space usage, locality and read efficiency follow those introduced by Cash and Tessaro [8].

Space. A symmetric searchable encryption scheme $(\mathsf{KeyGen}, \mathsf{EDBSetup}, \mathsf{TokGen}, \mathsf{Search}, \mathsf{Resolve})$ uses space $s = s(\lambda, N)$ if for any $\lambda, N \in \mathbb{N}$, for any database DB of size N, and for any key K produced by $\mathsf{KeyGen}(1^\lambda)$, the algorithm $\mathsf{EDBSetup}(K, \mathsf{DB})$ produces encrypted databases that can be represented using s machine words.

Locality. The search procedure of any SSE scheme can be decomposed into a sequence of contiguous reads from the encrypted database EDB, and the locality is defined as the number of such reads. Specifically, locality is defined by viewing the Search algorithm of an SSE scheme as an algorithm that does not obtain as input the actual encrypted database, but rather only obtains oracle access to it. Each query to this oracle consists of an interval $[a_i, b_i]$, and the oracle replies with the machine words that are stored in this interval of EDB. At first, the Search algorithm is invoked on a token τ and queries its oracle with some interval $[a_1, b_1]$. Then, it iteratively continues to compute the next interval to read based on τ and all previously read intervals. We denote these intervals by $\mathsf{ReadPat}(\mathsf{EDB}, \tau)$.

Definition 2.1 (Locality). *An SSE scheme Π is* d-local *(or has locality* d*) if for every λ, DB and $w \in \mathsf{W}$, $K \leftarrow \mathsf{KeyGen}(1^\lambda)$, $\mathsf{EDB} \leftarrow \mathsf{EDBSetup}(K, \mathsf{DB})$ and $\tau \leftarrow \mathsf{TokGen}(K, w)$ we have that $\mathsf{ReadPat}(\mathsf{EDB}, \tau)$ consists of at most d intervals.*

Read efficiency. The notion of read efficiency compares the overall size of the portion of EDB that is read on each query to the size of the actual answer to the query. For a given DB and w, we let $||\mathsf{DB}(w)||$ denote the number of words in the encoding of $\mathsf{DB}(w)$.

Definition 2.2 (Read efficiency). *An SSE scheme Π is* r-read efficient *(or has read efficiency* r*) if for any λ, DB, and $w \in \mathsf{W}$, we have that $\mathsf{ReadPat}(\tau, \mathsf{EDB})$ consists of intervals of total length at most $\mathsf{r} \cdot ||\mathsf{DB}(w)||$ words.*

Security Notions

The standard security definition for SSE schemes follows the ideal/real simulation paradigm. We consider both static and adaptive security, where the difference is whether the adversary chooses its queries statically (i.e., before seeing any token), or in an adaptive manner (i.e., the next query may be a function of the previous tokens). In both cases, some information is leaked to the server, which is formalized by letting the simulator receive the evaluation of some "leakage function" on the database itself and the real tokens. We start with the static case.

The real execution. The real execution is parameterized by the scheme Π, the adversary $\mathcal{A}$, and the security parameter λ. In the real execution the adversary is invoked on 1^λ, and outputs a database DB and a list of queries $\mathbf{w} = \{w_i\}_i$. Then, the experiment invokes the key-generation algorithm and the database setup algorithms, $K \leftarrow \mathsf{KeyGen}(1^\lambda)$ and $\mathsf{EDB} \leftarrow \mathsf{EDBSetup}(K, \mathsf{DB})$. Then, for each query $\mathbf{w} = \{w_i\}_i$ that the adversary has outputted, the token generator algorithm is run to obtain $\tau_i = \mathsf{TokGen}(w_i)$. The adversary is given the encrypted database EDB and the resulting tokens $\boldsymbol{\tau} = \{\tau_i\}_{w_i \in \mathbf{w}}$, and outputs a bit b.

The ideal execution. The ideal execution is parameterized by the scheme Π, a leakage function $\mathcal{L}$, the adversary $\mathcal{A}$, a simulator $\mathcal{S}$ and the security parameter λ. In this execution, the adversary $\mathcal{A}$ is invoked on 1^λ, and outputs $(\mathsf{DB}, \mathbf{w})$ similarly to the real execution. However, this time the simulator $\mathcal{S}$ is given the evaluation of the leakage function on $(\mathsf{DB}, \mathbf{w})$ and should output $\mathsf{EDB}, \boldsymbol{\tau}$ (i.e., $(\mathsf{EDB}, \boldsymbol{\tau}) \leftarrow \mathcal{S}(\mathcal{L}(\mathsf{DB}, \mathbf{w}))$). The execution follows by giving $(\mathsf{EDB}, \boldsymbol{\tau})$ to the adversary $\mathcal{A}$, which outputs a bit b.

Let SSE-$\textsc{Real}_{\Pi,\mathcal{A}}(\lambda)$ denote the output of the real execution, and let SSE-$\textsc{Ideal}_{\Pi,\mathcal{L},\mathcal{A},\mathcal{S}}(\lambda)$ denote the output of the ideal execution, with the adversary $\mathcal{A}$, simulator $\mathcal{S}$ and leakage function $\mathcal{L}$. We now ready to define security of SSE:

Definition 2.3 (Static $\mathcal{L}$-secure SSE). *Let $\Pi = (\mathsf{KeyGen}, \mathsf{EDBSetup}, \mathsf{TokGen}, \mathsf{Search})$ be an SSE scheme and let $\mathcal{L}$ be a leakage function. We say that*

the scheme Π is static $\mathcal{L}$-secure searchable encryption *if for every* PPT *adversary $\mathcal{A}$, there exists a* PPT *simulator $\mathcal{S}$ and a negligible function* $\mathsf{negl}(\cdot)$ *such that*

$$\left|\Pr\left[\text{SSE-REAL}_{\Pi,\mathcal{A}}(\lambda)=1\right]-\Pr\left[\text{SSE-IDEAL}_{\Pi,\mathcal{L},\mathcal{A},\mathcal{S}}(\lambda)=1\right]\right|<\mathsf{negl}(\lambda)$$

Adaptive setting. In the adaptive setting, the adversary is not restricted to specifying all of its queries $\mathbf{w}$ in advance, but can instead choose its queries during the execution in an adaptive manner, depending on the encrypted database EDB and on the tokens that it sees. Let $\text{SSE-REAL}^{\mathsf{adapt}}_{\Pi,\mathcal{A}}(\lambda)$ denote the output of the real execution in this adaptive setting. In the ideal execution, the simulator $\mathcal{S}$ is now an interactive Turing machine, which interacts with the experiment by responding to queries. First, the simulator $\mathcal{S}$ is invoked on $\mathcal{L}(\mathsf{DB})$ and outputs EDB. Then, for every query w_i that $\mathcal{A}$ may output, the function $\mathcal{L}$ is invoked on DB and all previously queries $\{w_j\}_{j<i}$ and the new query w_i, outputs some new leakage information which is given to the simulator $\mathcal{S}$. The latter outputs some t_i, which is given back to $\mathcal{A}$, who may then issue a new query. At the end of the execution, $\mathcal{A}$ outputs a bit b. Let $\text{SSE-IDEAL}^{\mathsf{adapt}}_{\Pi,\mathcal{L},\mathcal{A},\mathcal{S}}(\lambda)$ be the output of the ideal execution. The adaptive security of SSE is defined as follows:

Definition 2.4 (Adaptive $\mathcal{L}$-secure SSE). *Let* $\Pi=(\mathsf{KeyGen},\mathsf{EDBSetup},\mathsf{TokGen},\mathsf{Search})$ *be an SSE scheme and let $\mathcal{L}$ be a leakage function. We say that the scheme Π is* adaptive $\mathcal{L}$-secure searchable encryption *if for every* PPT *adversary $\mathcal{A}$, there exists a* PPT *simulator $\mathcal{S}$ and a negligible function* $\mathsf{negl}(\cdot)$ *such that*

$$\left|\Pr\left[\text{SSE-REAL}^{\mathsf{adapt}}_{\Pi,\mathcal{A}}(\lambda)=1\right]-\Pr\left[\text{SSE-IDEAL}^{\mathsf{adapt}}_{\Pi,\mathcal{L},\mathcal{A},\mathcal{S}}(\lambda)=1\right]\right|<\mathsf{negl}(\lambda)$$

The leakage function. Following the standard notions of security for SSE we consider the leakage function $\mathcal{L}_{\min}$ for one-round protocols and the leakage function $\mathcal{L}_{\mathsf{sizes}}$ for two-round protocols, where

$$\mathcal{L}_{\min}(\mathsf{DB},\mathbf{w})=\left(N,\{\mathsf{DB}(w)\}_{w\in\mathbf{w}}\right),$$
$$\mathcal{L}_{\mathsf{sizes}}(\mathsf{DB},\mathbf{w})=\left(N,\{|\mathsf{DB}(w)|\}_{w\in\mathbf{w}}\right),$$

and $N=\sum_{w\in W}|\mathsf{DB}(w)|$ is the size of the database. That is, both functions return the size of the database, and the difference between them is that the function $\mathcal{L}_{\min}$ returns the actual documents that contain each keyword $w\in\mathbf{w}$ that the adversary has queried, whereas the function $\mathcal{L}_{\mathsf{sizes}}$ returns only the number of such documents.

The leakage functions in the adaptive setting are defined analogously. That is, for a database DB, a set of "previous" queries $\{w_j\}_{j<i}$, and a new query w_i, we define

$$\mathcal{L}^{\mathsf{adap}}_{\min}(\mathsf{DB},\{w_j\}_{j<i},w_i)=\begin{cases}N & \text{if } (\{w_j\}_{j<i},w_i)=(\perp,\perp)\\ \mathsf{DB}(w_i) & \text{otherwise}\end{cases}$$
$$\mathcal{L}^{\mathsf{adap}}_{\mathsf{size}}(\mathsf{DB},\{w_j\}_{j<i},w_i)=\begin{cases}N & \text{if } (\{w_j\}_{j<i},w_i)=(\perp,\perp)\\ |\mathsf{DB}(w_i)| & \text{otherwise}\end{cases}.$$

2.2 Static Hash Tables

In our schemes we rely on static hash tables (also known as static dictionaries). These are data structures that given a set S can support lookup operations in constant time in the standard unit-cost word-RAM model. Specifically, a static hash table consists of a pair of algorithms denoted $(\mathsf{HTSetup}, \mathsf{HTLookup})$. The algorithm $\mathsf{HTSetup}$ gets as input a set $S = \{(\ell_i, d_i)\}_{i=1}^{k}$ of pairs (ℓ_i, d_i) of strings, where $\ell_i \in \{0,1\}^s$ is the label and $d_i \in \{0,1\}^r$ is the data. The output of this algorithm is a hash table $\mathsf{HT}(S)$. The lookup algorithm $\mathsf{HTLookup}$ on input $(\mathsf{HT}(S), \ell)$ returns d if $(\ell, d) \in S$, and $\perp$ otherwise.

There exist many constructions of static hash tables that use linear space (i.e., $O(k(r+s))$ bits) and answer lookup queries by reading a constant number of contiguous s-bit blocks and r-bit blocks (see, for example, [1,26], and the many references therein).

2.3 Cuckoo Hashing with a Stash

Cuckoo hashing is an efficient and practical hash table designed by Pagh and Rodler [26], providing worst-case constant lookup time and uses linear space. An important property of cuckoo hashing is that by storing a few elements in a secondary (small) data structure, referred to as a "stash", it is possible to decrease its failure probability from noticeable to negligible [21]. For our purposes in this work, it suffices to consider the following abstraction of cuckoo hashing with a stash:

- The memory is an abstract array $[m]$, where each cell may contain a single element or NULL.
- The potential locations of any element are randomly sampled (instead of being determined by hash functions).

We now summarize the abstract properties of cuckoo hashing with a stash in which we are interested for our construction in Sect. 4:

1. For storing n lists, where each list consists of ℓ elements, an array of size $O(n \cdot \ell)$ is used. The array is partitioned into two segments – a cuckoo hashing segment of size $O(n \cdot \ell)$ and a stash segment of size $s \cdot \ell$.
2. Fetching a list requires accessing two random locations (of size ℓ each) in the cuckoo hashing segment and accessing the entire stash segment.
3. When using a stash of size $s = n^{o(1)}$, the probability that n lists can be successfully stored is $1 - O(n^{s/2})$ [4, Theorem 2].[7]

[7] Note that in the original work of Kirsch et al. [21] they considered a constant-sized stash, whereas in this work we are interested in a stash whose size is not necessarily constant, and thus we rely on [4].

3 The Pad-and-Split Framework: A Stronger Lower Bound

In this section we first formalize our pad-and-split framework for the design of symmetric searchable encryption schemes (Sect. 3.1). Then, we show that it captures the memory access patterns underlying essentially all of the existing symmetric searchable encryption schemes other than the schemes of Asharov et al. [2] (Sect. 3.2), and discuss the instantiation of Demertzis and Papamanthou (Sect. 3.3) whose tradeoff matches our lower bound (Sect. 3.4).

3.1 The Pad-and-Split Framework

Our framework considers symmetric searchable encryption schemes that are characterized by a deterministic algorithm denoted $\mathsf{SplitList}$, and consist of the following two phases:

- Given a database $\mathsf{DB} = \{\mathsf{DB}(w_1), \ldots, \mathsf{DB}(w_{n_\mathsf{W}})\}$ of size N, for each keyword w_i the scheme invokes the $\mathsf{SplitList}$ algorithm on the length n_i of its corresponding list $\mathsf{DB}(w_i)$, to obtain a vector $(x_i^{(1)}, \ldots, x_i^{(m)})$ of integers. The scheme then potentially pads the list $\mathsf{DB}(w_i)$ by adding "dummy" elements, and splits the padded list into sublists of lengths $\mathsf{len}^{(1)}, \ldots, \mathsf{len}^{(m)}$, where $x_i^{(j)}$ denotes the number of sublists of each length $\mathsf{len}^{(j)}$.
- For each possible length $\mathsf{len}^{(j)}$, the scheme groups together all sublists of length $\mathsf{len}^{(j)}$, and independently processes each such group to produce an encrypted database EDB.

A key property of our framework is that the $\mathsf{SplitList}$ algorithm, which determines the number of sublists of each length, does not take as input an actual list $\mathsf{DB}(w_i)$ but only its length $n_i = |\mathsf{DB}(w_i)|$. This algorithm is parameterized by the possible lengths $\mathsf{len}^{(1)}, \ldots, \mathsf{len}^{(m)}$ of sublists, and also by upper bounds $s^{(1)}, \ldots, s^{(m)}$ on the total number of sublists of lengths $\mathsf{len}^{(1)}, \ldots, \mathsf{len}^{(m)}$, respectively. We allow the parameters m, $\mathsf{len}^{(1)}, \ldots, \mathsf{len}^{(m)}$ and $s^{(1)}, \ldots, s^{(m)}$ to depend on the total length $N = \sum_{i=1}^{n_\mathsf{W}} |\mathsf{DB}(w_i)|$ of the database, but do not explicitly denote this for ease of notation.

We consider any possible instantiation of the $\mathsf{SplitList}$ algorithm subject to satisfying two natural requirements. First, we require that each list $\mathsf{DB}(w_i)$ is split into sublists whose total length is at least the length of $\mathsf{DB}(w_i)$. Second, we require that for every possible sublist length $\mathsf{len}^{(j)}$ there are at most $s^{(j)}$ sublists of length $\mathsf{len}^{(j)}$ in the worst-case over all possible databases of size N. Formally:

Definition 3.1. *We say that a* $\mathsf{SplitList}$ *algorithm, parameterized by* $(\mathsf{len}^{(1)}, \ldots, \mathsf{len}^{(m)})$ *and* $(s_1, \ldots, s_m)$ *is* valid *if for every integer* N *and for every vector of lengths* $(n_1, \ldots, n_k)$ *with* $\sum_{i=1}^{k} n_i = N$*, it holds that:*

- **Each list is not longer than the sum of lengths of its sublists:** *For every* n_i *it holds that* $x_i^{(1)} \cdot \mathsf{len}^{(1)} + \cdots + x_i^{(m)} \cdot \mathsf{len}^{(m)} \geq n_i$*, where* $(x_i^{(1)}, \ldots, x_i^{(m)}) = \mathsf{SplitList}(N, n_i)$*.*
- **Each** $s^{(j)}$ **upper bounds the number of sublists of length** $\mathsf{len}^{(j)}$**:** *For every* $j \in [m]$ *it holds that* $\sum_{i=1}^{k} x_i^{(j)} \leq s^{(j)}$*, where* $(x_i^{(1)}, \ldots, x_i^{(m)}) = \mathsf{SplitList}(N, n_i)$ *for every* $i \in [k]$*.*

In addition, we say that SplitList has locality L if each list $\mathsf{DB}(w_i)$ is split into at most L sublists. Formally:

Definition 3.2. *We say that a* SplitList *algorithm* has locality $L = L(N)$ *if for every* n_i *it holds that* $x_i^{(1)} + \cdots + x_i^{(m)} \leq L$*, where* $(x_i^{(1)}, \ldots, x_i^{(m)}) = \mathsf{SplitList}(N, n_i)$*.*

Equipped with our notion of a valid SplitList algorithm, we describe a general template (see Construction 3.5) for constructing symmetric searchable encryption schemes given any such algorithm. We rely, in addition, on a pseudorandom function PRF and a private-key encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ with pseudorandom ciphertexts – both of which can be constructed based on any one-way function. This yields the following theorem:

Theorem 3.3. *Given a valid* SplitList *algorithm with parameters* $(\mathsf{len}^{(1)}, \ldots, \mathsf{len}^{(m)})$ *and* $(s^{(1)}, \ldots, s^{(m)})$*, a pseudorandom function* PRF*, and a private-key encryption scheme* $(\mathsf{Enc}, \mathsf{Dec})$ *with pseudorandom ciphertexts, Construction 3.5 is a static* $\mathcal{L}_{\text{min}}$*-secure symmetric searchable encryption scheme for databases of size* N *with the following parameters:*

- *Space* $O\left(\sum_{j=1}^{m} s^{(j)} \cdot \mathsf{len}^{(j)}\right)$*.*
- *Locality* $O(L(N))$*, where* SplitList *has locality* $L(N)$*.*
- *Read efficiency* $O(1)$*.*
- *Token size* $O(1)$*.*

Moreover, Construction 3.5 is an adaptive $\mathcal{L}_{\text{min}}^{\mathsf{adap}}$*-secure symmetric searchable encryption scheme in the random-oracle model, when instantiating* PRF *and* $(\mathsf{Enc}, \mathsf{Dec})$ *appropriately.*

Note that Theorem 3.3 guarantees that Construction 3.5 is statically-secure in the standard model (although, we do prove it can be made adaptively secure in the random-oracle model). The next theorem shows that a simple modification of the scheme (described as Construction 3.6), based on an idea sketched by Stefanov et al. [28], is in fact adaptively secure in the standard model. This comes at the cost of increasing the token size from tokens of size $O(1)$ to tokens of size $O(L)$, where L is the locality of the SplitList algorithm. In this scheme, the client decrypts the results sent by the server (using the Resolve algorithm), and thus the scheme leaks only the size of the results. This is in contrast to the

scheme described in Construction 3.5, where the server decrypts the results, and thus the scheme leaks the results themselves[8].

Theorem 3.4. *Given a valid* SplitList *algorithm with parameters* $(\mathsf{len}^{(1)}, \ldots, \mathsf{len}^{(m)})$ *and* $(s^{(1)}, \ldots, s^{(m)})$*, a pseudorandom function* PRF*, and a private-key encryption scheme* $(\mathsf{Enc}, \mathsf{Dec})$ *with pseudorandom ciphertexts, Construction 3.6 is an adaptive* $\mathcal{L}^{\mathsf{adap}}_{\mathsf{size}}$*-secure symmetric searchable encryption scheme for databases of size* N *with the following parameters:*

- *Space* $O\left(\sum_{j=1}^{m} s^{(j)} \cdot \mathsf{len}^{(j)}\right)$.
- *Locality* $O(L(N))$*, where* SplitList *has locality* $L(N)$.
- *Read efficiency* $O(1)$.
- *Token size* $O(L(N))$.

In the remainder of this section we provide a high-level overview of these schemes. The proofs of Theorems 3.3 and 3.4 can be found in the full version of this paper [3].

Overview of the schemes. In both schemes each list of document identifiers $\mathsf{DB}(w_i)$ is padded and split as dictated by the output $(x_i^{(1)}, \ldots, x_i^{(m)}) = \mathsf{SplitList}(N, |\mathsf{DB}(w_i)|)$. That is, $\mathsf{DB}(w_i)$ is padded to length $x_i^{(1)} \cdot \mathsf{len}^{(1)} + \cdots + x_i^{(m)} \cdot \mathsf{len}^{(m)}$, and split into sublists, where for each $j \in [m]$ there are $x_i^{(j)}$ sublists of length $\mathsf{len}^{(j)}$. Then, we construct an encrypted database which consists of the following hash tables:

- A hash table that stores (in encrypted manner) the lengths of all lists, and is padded to contain exactly N elements.
- For every $j \in [m]$ a hash table that stores (in an encrypted manner) all sublists of length $\mathsf{len}^{(j)}$, and is padded to contain exactly $s^{(j)}$ sublists of this length.

In all hash tables we store the various elements according to pseudorandom labels that are derived from each corresponding keyword w via a pseudorandom function whose key is known only to the client. Intuitively speaking, the scheme is secure for any valid SplitList algorithm due to the following three reasons: (1) The number of padded elements and the number of sublists each list is split into depend only on the length of each list, (2) each hash table consists of encrypted elements with pseudorandom labels, and (3) the size of each hash table depends only on the size of the database.

The main differences between Construction 3.5 (providing static security) and Construction 3.6 (providing adaptive security) are as follows:

1. The lengths of the lists in Construction 3.6 are encrypted using one-time pads. This is required in order to allow "explaining" a random value as the encryption of any particular length on the fly (given that adversaries may be adaptive).

[8] Note that any scheme in which the server decrypts the results can be easily transformed into a scheme where only the client decrypts the results by adding an additional encryption layer – but this does not necessarily hold in the other direction.

2. In Construction 3.5, for each searched keyword the server is given keys derived from that keyword, allowing it to compute the labels and decrypt the document identifiers associated with that keyword. In Construction 3.6 the server is given the labels themselves (thus, the token size is $O(L)$), and can only locate the encrypted document identifiers, but not to decrypt them.

3.2 The Generality of the Pad-and-Split Framework

We now demonstrate that our pad-and-split framework captures the memory access patterns underlying the vast majority of existing symmetric searchable encryption schemes for supporting keywords search (i.e., we show that these schemes can be obtained as instantiations of our framework). We note that each of these schemes offers other important implementation details, improvements and optimizations that we do not intend to capture using our framework (since these are not directly related to the tradeoff between space, locality, and read efficiency), and we refer to the relevant papers for further details.

The scheme of Curtmola et al. [11]. This is the most common technique underlying the vast majority of existing schemes (in particular, [7,10,12,20,23,29]). In this scheme each list is split into single elements (i.e., sublists of length 1), and those are stored in the same hash table. This is captured by our framework when setting $m = 1$, $\mathsf{len}^{(1)} = 1$, $s^{(1)} = N$, and $\mathsf{SplitList}(N, n_i) = (n_i)$. This results in a scheme with space $O(N)$, locality $O(N)$, and read efficiency $O(1)$.

The 2lev scheme of Cash et al. [6]. This scheme can be viewed as a pad-and-split scheme with two possible lengths, b and B, where $b < B$. A list of length at most b is padded to length b, and a list of length greater than b is padded to a length that is a multiple of B and then split into sublists of length B (in order to reduce space overhead, this scheme does not add dummy lists, thus resulting in a non-standard leakage function). This results in a scheme with space $O(N \cdot (b + \frac{B}{b+1}))$, locality $O(N/B)$, and read efficiency $O(1)$.

A simple scheme with $O(N^2)$ space. In this scheme each list is padded to the maximal possible length (i.e., to length N, where $N = \sum_{i=1}^{n_\mathsf{W}} |\mathsf{DB}(w_i)|$), and all lists are stored in the same hash table. This is captured by our framework when setting $m = 1$, $\mathsf{len}^{(1)} = N$, $s^{(1)} = N$ and $\mathsf{SplitList}(N, n_i) = (1)$. This results in a scheme with space $O(N^2)$, locality $O(1)$, and read efficiency $O(1)$.

The scheme of Cash and Tessaro [8]. This scheme splits a list of length n_i into at most $\log n_i$ sublists of lengths that are powers of 2 according to the binary representation of n_i. Then, for each possible power of 2, the scheme stores sublists of that length in a separate hash table. This is captured by our framework when setting $m = \lfloor \log N \rfloor + 1$, $\mathsf{len}^{(j)} = 2^{j-1}$, $s^{(j)} = N/2^{j-1}$, and the $\mathsf{SplitList}$ algorithm on input n_i outputs a binary vector of length m which corresponds to the binary representation of n_i. This results in a scheme with space $O(\sum_{j=1}^{m} \mathsf{len}^{(j)} s^{(j)}) = O(N \log N)$, locality $O(\log N)$, and read efficiency $O(1)$.

CONSTRUCTION 3.5 (One-Round Pad-and-Split SSE Scheme)

A pad-and-split SSE scheme is parameterized by a $\mathsf{SplitList}$ algorithm, and by the following values (all values are functions of the size N of the database):

1. Locality parameter L.
2. Possible lengths $\mathsf{len}^{(1)}, \ldots, \mathsf{len}^{(m)}$ of sublists.
3. Upper bounds $s^{(1)}, \ldots, s^{(m)}$ on the total number of sublists of lengths $\mathsf{len}^{(1)}, \ldots, \mathsf{len}^{(m)}$, respectively.

Key generator. The algorithm KeyGen on input 1^λ samples and outputs a key $K \leftarrow \{0,1\}^\lambda$ for PRF.

Setup. The algorithm $\mathsf{EDBSetup}$ on input (K, DB) is defined as follows:

1. Initialize $t+1$ empty sets $T, T_1, \ldots, T_m$, where T will consist of the lengths of the lists, and each set T_j will consist of all sublists of length $\mathsf{len}^{(j)}$.
2. For every keyword $w_i \in \mathsf{W}$ with an associated list $\mathsf{DB}(w_i) = \{\mathsf{id}_1, \ldots, \mathsf{id}_{n_i}\}$:
   (a) Compute $(\mathsf{label}_i, K_i, \widehat{K_i}) = \mathsf{PRF}_K(w_i)$.
   (b) Compute $\widehat{n_i} = \mathsf{Enc}_{K_i}(n_i)$ and add the pair $(\mathsf{label}_i, \widehat{n_i})$ to the set T.
   (c) Compute $(x_i^{(1)}, \ldots, x_i^{(m)}) = \mathsf{SplitList}(N, n_i)$.
   (d) For every $j = 1, \ldots, m$:
      i. For every $x = 1, \ldots, x_i^{(j)}$:
         A. Take the next $\mathsf{len}^{(j)}$ elements from the list $\mathsf{DB}(w_i)$ and create a block $\{\mathsf{id}'_1, \ldots, \mathsf{id}'_{\mathsf{len}^{(j)}}\}$. If there are less than $\mathsf{len}^{(j)}$ elements left in $\mathsf{DB}(w_i)$, then pad with dummy elements.
         B. Compute a label: $\mathsf{label}_{j,x} = \mathsf{PRF}_{\widehat{K_i}}(j, x)$.
         C. Encrypt $d_{j,x} = \left(\mathsf{Enc}_{K_i}(\mathsf{id}'_1), \ldots, \mathsf{Enc}_{K_i}(\mathsf{id}'_{\mathsf{len}^{(j)}})\right)$.
         D. Insert the pair $(\mathsf{label}_{j,x}, d_{j,x})$ into the set T_j.
3. Pad the set T to contain exactly N elements by adding dummy elements, and pad each set T_j to contain exactly $s^{(j)}$ elements by adding dummy elements.
4. For each set $T, T_1, \ldots, T_m$, uniformly shuffle the set, and generate a hash table by invoking the $\mathsf{HTSetup}$ algorithm for obtaining hash tables $\mathsf{HT}(T), \mathsf{HT}(T_1), \ldots, \mathsf{HT}(T_m)$.
5. Output $\mathsf{EDB} = (\mathsf{HT}(T), (\mathsf{HT}(T_1), \ldots, \mathsf{HT}(T_m)))$.

Token generator. The algorithm TokGen on input (K, w_i) computes and outputs the token $\tau_i = (\mathsf{label}_i, K_i, \widehat{K_i}) = \mathsf{PRF}_K(w_i)$.

Search. The algorithm Search on input (τ_i, EDB), where $\tau_i = (\mathsf{label}_i, K_i, \widehat{K_i})$ and $\mathsf{EDB} = (\mathsf{HT}(T), \mathsf{HT}(T_1), \ldots, \mathsf{HT}(T_m))$, is defined as follows:

1. Initialize a list of document identifiers $R = \emptyset$.
2. Invoke $\mathsf{HTLookup}$ on the hash table $\mathsf{HT}(T)$ and label label_i to retrieve $\widehat{n_i} = \mathsf{Enc}_{K_i}(n_i)$. Decrypt $\widehat{n_i}$ using the key K_i, and compute $(x_i^{(1)}, \ldots, x_i^{(m)}) = \mathsf{SplitList}(N, n_i)$.
3. For every $j \in [m]$ and for every $x \in \left[x_i^{(j)}\right]$ compute $\mathsf{label}_{j,x} = \mathsf{PRF}_{\widehat{K_i}}(j, x)$. Invoke $\mathsf{HTLookup}$ on the hash table $\mathsf{HT}(T_j)$ for the label $\mathsf{label}_{j,x}$, and obtain the block $d_{j,x}$. Decrypt the block using the key K_i and add the elements to the list R. For a block that contains dummy elements, obtain and decrypt only the part that does not contain dummy elements.

CONSTRUCTION 3.6 (Two-Round Pad-and-Split SSE Scheme)

A pad-and-split SSE scheme is parameterized by a $\mathsf{SplitList}$ algorithm, and the following values (all values are functions of the size N of the database):

1. Locality parameter L.
2. Possible lengths $\mathsf{len}^{(1)}, \ldots, \mathsf{len}^{(m)}$ of sublists.
3. Upper bounds $s^{(1)}, \ldots, s^{(m)}$ on the total number of sublists of lengths $\mathsf{len}^{(1)}, \ldots, \mathsf{len}^{(m)}$, respectively.

Key generator. The algorithm KeyGen on input 1^λ samples a key $K \leftarrow \{0,1\}^\lambda$ for PRF, samples a key $\widehat{K} \leftarrow \{0,1\}^\lambda$ for $(\mathsf{Enc}, \mathsf{Dec})$, and outputs $(K, \widehat{K})$.
Setup. The algorithm $\mathsf{EDBSetup}$ on input $((K, \widehat{K}), \mathsf{DB})$ is defined as follows:

1. Initialize $t+1$ empty sets $T, T_1, \ldots, T_m$, where T will consist of the lengths of the lists, and each set T_j will consist of all sublists of length $\mathsf{len}^{(j)}$.
2. For every keyword $w_i \in \mathsf{W}$ with an associated list $\mathsf{DB}(w_i) = \{\mathsf{id}_1, \ldots, \mathsf{id}_{n_i}\}$:
   (a) Compute $((\mathsf{label}_i, K_i), (\mathsf{label}_{i,1}, \ldots, \mathsf{label}_{i,L})) = \mathsf{PRF}_K(w_i)$.
   (b) Compute $\widehat{n_i} = K_i \oplus n_i$ and add the pair $(\mathsf{label}_i, \widehat{n_i})$ to the set T.
   (c) Compute $(x_i^{(1)}, \ldots, x_i^{(m)}) = \mathsf{SplitList}(N, n_i)$.
   (d) For every $j = 1, \ldots, m$:
      i. For every $x = 1, \ldots, x_i^{(j)}$:
         A. Take the next $\mathsf{len}^{(j)}$ elements from the list $\mathsf{DB}(w_i)$ and create a block $\{\mathsf{id}'_1, \ldots, \mathsf{id}'_{\mathsf{len}^{(j)}}\}$. If there are less than $\mathsf{len}^{(j)}$ elements left in $\mathsf{DB}(w_i)$, then pad with dummy elements.
         B. Let label be the first unused label from $(\mathsf{label}_{i,1}, \ldots, \mathsf{label}_{i,L})$.
         C. Encrypt $d_{j,x} = \left(\mathsf{Enc}_{\widehat{K}}(\mathsf{id}'_1), \ldots, \mathsf{Enc}_{\widehat{K}}(\mathsf{id}'_{\mathsf{len}^{(j)}})\right)$.
         D. Insert the pair $(\mathsf{label}, d_{j,x})$ into the set T_j.
3. Pad the set T to contain exactly N elements by adding dummy elements, and pad each set T_j to contain exactly $s^{(j)}$ elements by adding dummy elements.
4. For each set $T, T_1, \ldots, T_m$, uniformly shuffle the set, and generate a hash table by invoking the $\mathsf{HTSetup}$ algorithm for obtaining hash tables $\mathsf{HT}(T), \mathsf{HT}(T_1), \ldots, \mathsf{HT}(T_m)$.
5. Output $\mathsf{EDB} = (\mathsf{HT}(T), (\mathsf{HT}(T_1), \ldots, \mathsf{HT}(T_m)))$.

Token generator. The algorithm TokGen on input $((K, \widehat{K}), w_i)$ computes and outputs the token $\tau_i = ((\mathsf{label}_i, K_i), (\mathsf{label}_{i,1}, \ldots, \mathsf{label}_{i,L})) = \mathsf{PRF}_K(w_i)$.
Search. The algorithm Search on input (τ_i, EDB), where $\tau_i = ((\mathsf{label}_i, K_i), (\mathsf{label}_{i,1}, \ldots, \mathsf{label}_{i,L}))$ and $\mathsf{EDB} = (\mathsf{HT}(T), \mathsf{HT}(T_1), \ldots, \mathsf{HT}(T_m))$, is defined as follows:

1. Initialize a list of results $R = \emptyset$.
2. Invoke $\mathsf{HTLookup}$ on the hash table $\mathsf{HT}(T)$ and label label_i to retrieve $\widehat{n_i} = K_i \oplus n_i$. Decrypt $n_i = K_i \oplus \widehat{n_i}$ and compute $(x_i^{(1)}, \ldots, x_i^{(m)}) = \mathsf{SplitList}(N, n_i)$.
3. For every $j \in [m]$ and for every $x \in \left[x_i^{(j)}\right]$, let label be the first unused label from $(\mathsf{label}_{i,1}, \ldots, \mathsf{label}_{i,L})$. Invoke $\mathsf{HTLookup}$ on the hash table $\mathsf{HT}(T_j)$ for the label $\mathsf{label}_{j,x}$, obtain the block $d_{j,x}$, and add its elements to the list R. For a block that contains dummy elements, obtain only the part that does not contain dummy elements.

Resolve. The algorithm $\mathsf{Resolve}$ on input $((K, \widehat{K}), R)$ computes and outputs the identifiers $M = \{\mathsf{Dec}_{\widehat{K}}(c) : c \in R\}$.

The scheme of Asharov et al. [2, Sect. 5]**.** This scheme improves the one of Cash and Tessaro [8]. In this scheme, a list of length $2^{p_i-1} < n_i \leq 2^{p_i}$ is padded to length 2^{p_i} and stored as a whole. This is captured by our framework when setting $m = \lceil \log N \rceil + 1$, $\mathsf{len}^{(j)} = 2^{j-1}$, $s^{(j)} = 2N/2^{j-1}$, and the $\mathsf{SplitList}$ algorithm, on input n_i, outputs a vector of length m where all the entries are zeros except for a one that appears in the location $\lceil \log n_i \rceil + 1$. This results in a scheme with space $O(\sum_{j=1}^{m} \mathsf{len}^{(j)} s^{(j)}) = O(N \log N)$, locality $O(1)$, and read efficiency $O(1)$.

3.3 An Optimal Instantiation for Any Locality

As discussed in Sect. 1.1, the lower bound that we prove for schemes in the pad-and-split framework matches the tradeoff provided by the scheme of Demertzis and Papamanthou [14] (which is captured by our framework). Specifically, when setting the read efficiency of their scheme to $O(1)$, one obtains a statically-secure scheme with space $O(N \log N / \log L)$, locality L, and read efficiency $O(1)$. It should be noted that their scheme supports also non-constant read efficiency, but in that case it is not captured by our framework as it leaks additional information (in particular, the random choices made by the setup algorithm).

In what follows we describe their instantiation within our above-described template. Their scheme is obtained by splitting each list to sublists of lengths that are a power of the locality L. In our notation, we set $m = \lfloor \log N / \log L \rfloor + 1 = \lfloor \log_L N \rfloor + 1$, $\mathsf{len}^{(j)} = L^{j-1}$, and $s^{(j)} = 2N/\mathsf{len}^{(j)}$ for every $j \in [m]$. As for the splitting algorithm, a list of length $L^{j-1} \leq n_i < L^j$ is padded to a length that is a multiple of L^{j-1}, and split into at most L sublists of length L^{j-1}. More formally, $\mathsf{SplitList}(N, n_i)$ outputs a vector of length m, where all the entries are zeros except for the entry in the position $j = \lfloor \log_L(n_i) \rfloor + 1$, which is set to the value $\lceil n_i / L^{j-1} \rceil \in \{1, \ldots, L\}$.

This is indeed a valid $\mathsf{SplitList}$ algorithm, and its locality is L. Specifically, for each n_i and j it holds that $\lceil n_i / L^{j-1} \rceil \cdot L^{j-1} \geq n_i$, that is, each list is not longer than the sum of the lengths of its sublists. Moreover, for $j = \lfloor \log_L(n_i) \rfloor + 1$ it also holds that $\lceil n_i / L^{j-1} \rceil \leq L$ and $\lceil n_i / L^{j-1} \rceil \cdot L^{j-1} < 2 \cdot n_i$. This means that the locality is L, and that the padding at most doubles the length of the list. Therefore, it is suffices to set $s^{(j)} = 2N/\mathsf{len}^{(j)}$, and thus it holds that $\sum_{j=1}^{m} \mathsf{len}^{(j)} \cdot s^{(j)} = m \cdot 2N = O(N \cdot \log N / \log L)$.

According to Theorems 3.3 and 3.4, the above splitting algorithm results in a searchable symmetric encryption schemes with space $O(N \cdot \log N / \log L)$, locality $O(L)$, and read efficiency $O(1)$. This yields the following corollaries:

Corollary 3.7 ([14]). *Assuming the existence of any one-way function, for any $L = L(N) > c$ (where c is an absolute constant) there exists a static $\mathcal{L}_{\min}$-secure symmetric searchable encryption scheme for databases of size N with the following parameters:*

- *Space $O(N \cdot \log N / \log L)$.*
- *Locality $L(N)$.*

- *Read efficiency* $O(1)$.
- *Token size* $O(1)$.

Moreover, the scheme is adaptively $\mathcal{L}_{\min}^{\mathsf{adap}}$*-secure in the random-oracle model, when instantiating its building blocks appropriately.*

Corollary 3.8. *Assuming the existence of any one-way function, for any* $L = L(N) > c$ *(where* c *is an absolute constant) there exists an adaptive* $\mathcal{L}_{\mathsf{size}}^{\mathsf{adap}}$*-secure symmetric searchable encryption scheme for databases of size* N *with the following parameters:*

- *Space* $O(N \cdot \log N / \log L)$.
- *Locality* $L(N)$.
- *Read efficiency* $O(1)$.
- *Token size* $O(L(N))$.

Better efficiency for super-constant sub-polynomial locality. For locality $L(N)$ satisfying $\omega(1) \leq L(N) \leq N^{o(1)}$ we can in fact instantiate our framework in a manner that reduces the expression $\sum_{j=1}^{m} \mathsf{len}^{(j)} s^{(j)}$ to $(1 + o(1))(N \cdot \log N / \log L)$. This matches our lower bound, which is shown to be $(1 - o(1))(N \cdot \log N / \log L)$, to within an additive lower-order term.

This is done as follows. Let $\widehat{L} = \lfloor L / \log L \rfloor$, and for a list of length n_i let j such that $\widehat{L}^j \leq n_i < \widehat{L}^{j+1}$. Represent $n_i = a \cdot \widehat{L}^j + b \cdot \widehat{L}^{j-1} + c$, where $a \in \{1, \ldots, \widehat{L} - 1\}$, $b \in \{0, \ldots, \widehat{L} - 1\}$, and $c \in \{0, \ldots, \widehat{L}^{j-1} - 1\}$. If $a \geq \log L$, then pad and split the list into at most $\widehat{L}$ sublists of length $\widehat{L}^j$. Otherwise, pad and split the list into at most $\widehat{L} \cdot \log L \leq L$ sublists of length $\widehat{L}^{j-1}$. This way, we never pad a list more than $(1 + 1/\log L)$ times its length, so for any j, we can set $s^{(j)} = (1 + 1/\log L)N/\mathsf{len}^{(j)}$, and obtain

$$\sum_{j=1}^{m} \mathsf{len}^{(j)} s^{(j)} = \left(1 + \frac{1}{\log L}\right) N \cdot \left(\left\lfloor \frac{\log N}{\log \widehat{L}} \right\rfloor + 1\right) = \big(1 + o(1)\big) N \cdot \frac{\log N}{\log L},$$

where the last equality holds since $\omega(1) \leq L \leq N^{o(1)}$.

3.4 Our Lower Bound for Pad-and-Split Schemes

In this section we present our lower bound on the trade-off between the space and the locality of any pad-and-split scheme. Recall that each such a scheme is characterized by a SplitList algorithm that satisfies a modest validity requirement (recall Definition 3.1), and is associated with the following parameters (all of which may be functions of the size N of the database):

- The possible lengths $\mathsf{len}^{(1)}, \ldots, \mathsf{len}^{(m)}$ of sublists to which the SplitList algorithm splits the list associated with each keyword, as described in Sect. 3.1.
- Upper bounds $s^{(1)}, \ldots, s^{(m)}$ on the total number of sublists of lengths $\mathsf{len}^{(1)}, \ldots, \mathsf{len}^{(m)}$, respectively, that are produced by the SplitList algorithm when processing an entire database.

Equipped with the above parameters, recall from Theorems 3.3 and 3.4 that the space usage of a pad-and-split scheme is $O\left(\sum_{j=1}^{m} s^{(j)} \cdot \mathsf{len}^{(j)}\right)$, and the locality of such a scheme is $O(L)$ where $L = L(N)$ is the locality of its $\mathsf{SplitList}$ algorithm (i.e., each list is split into at most L sublists). Thus, proving a lower bound on the trade-off between the space and the locality of pad-and-split schemes translates to proving such a lower bound on the corresponding parameters of their underlying $\mathsf{SplitList}$ algorithm. Theorem 1.1 follows as an immediate corollary of the following theorem, which we prove in the full version of this paper [3]:

Theorem 3.9. *Let* $\mathsf{SplitList}$ *be a valid splitting algorithm with parameters* $\mathsf{len}^{(1)}, \ldots, \mathsf{len}^{(m)}$ *and* $s^{(1)}, \ldots, s^{(m)}$*, and with locality* $L = L(N)$*. Then, for any* $0 < c < 1$ *it holds that*

$$\sum_{j=1}^{m} \mathsf{len}^{(j)} \cdot s^{(j)} \geq (1-c) \cdot N \cdot \left(\frac{\log N}{\log L - \log c + C_1} - C_2 \right),$$

where C_1 *and* C_2 *are small absolute constants.*

In particular, by setting $c = 1/2$ we obtain the lower bound $\sum_{j=1}^{m} \mathsf{len}^{(j)} \cdot s^{(j)} = \Omega(N \cdot \log N / \log L)$, which implies Theorem 1.1. In addition, if $\omega(1) \leq L(N) \leq N^{o(1)}$ then by setting $c = 1/\log L$ we obtain the tighter lower bound $\sum_{j=1}^{m} \mathsf{len}^{(j)} \cdot s^{(j)} \geq (1 - o(1))N \cdot \log N / \log L$.

4 The Statistical-Independence Framework: A Leveled Two-Choice Scheme

In this section we consider the statistical-independence framework introduced by Asharov et al. [2] for the design of symmetric searchable encryption schemes. As discussed in Sect. 1.2, within this framework we construct a scheme whose read efficiency when querying for a keyword w may depend on the length of the list $\mathsf{DB}(w)$ that is associated with w, and for any $n \leq N$ we denote by $\mathsf{r}(N, n)$ the read efficiency when retrieving a list of length n.[9] We prove the following theorem:

Theorem 4.1. *Assuming the existence of any one-way function, for any function* $f(N) = \omega(1)$ *there exists an adaptive* $\mathcal{L}_{\mathsf{size}}^{\mathsf{adap}}$*-secure symmetric searchable encryption scheme for databases of size* N *in which no keyword is associated with more than* $N/\log^3 N$ *identifiers, with the following parameters:*

- *Space* $O(N)$.
- *Locality* $O(1)$.
- *Read efficiency* $\mathsf{r}(N, n) = f(N) \cdot \epsilon(n)^{-1} + O(\log \log \log N)$*, where* $n = N^{1-\epsilon(n)}$.
- *Token size* $O(1)$.

[9] We emphasize that having the read efficiency depend on the length of the retrieved list does not hurt the security of SSE schemes, and our scheme still results in minimal leakage as required.

Comparing the performance of our new scheme with the lower bound of Asharov et al. in the statistical-independence framework, Theorem 4.1 matches their lower bound to within an additive $O(\log\log\log N)$ factor in the read efficiency. Specifically, Asharov et al. proved the following lower bound for schemes in the statistical-independence framework (restated to consider read efficiency $\mathsf{r}(N,n)$ that may depend on the length n of each list, and to consider constant locality):

Theorem 4.2 ([2]). *For any searchable symmetric encryption scheme in the statistical-independence framework with space $O(N\log N)$, locality $O(1)$, and read efficiency $\mathsf{r}(N,n)$, there exists a function $f(N)=\omega(1)$ such that $\mathsf{r}(N,n)=f(N)\cdot\epsilon(n)^{-1}$ for every $1\leq n\leq N/\log N$, where $n=N^{1-\epsilon(n)}$.*

In the remainder of this section we first overview the statistical independence framework for the design of symmetric searchable encryption schemes (Sect. 4.1), and then present our new scheme within this framework (Sect. 4.2).

4.1 The Statistical-Independence Framework

The statistical-independence framework of Asharov et al. [2] considers symmetric searchable encryption schemes that are characterized by a pair of algorithms, denoted $\mathsf{RangesGen}$ and $\mathsf{Allocation}$, and consist of the following two phases:

- Given a database $\mathsf{DB}=\{\mathsf{DB}(w_1),\ldots,\mathsf{DB}(w_{n_\mathsf{W}})\}$ of size N, for each keyword w_i the scheme invokes the $\mathsf{RangesGen}$ algorithm on the length n_i of its corresponding list $\mathsf{DB}(w_i)$, to obtain a set of *possible locations* in which the scheme may place the elements of the list $\mathsf{DB}(w_i)$. This set consists of several intervals and we denote it by $R_i=\{[a_1,b_1],\ldots,[a_d,b_d]\}\leftarrow\mathsf{RangesGen}(N,n_i)$.
  Looking ahead, when supplied with a token corresponding to a keyword w_i, the server will return to the client all data stored in the possible locations of the list $\mathsf{DB}(w_i)$ (the server will not actually know in which of the possible locations the elements of the list are actually placed).
- Given the sets of possible locations $R_1,\ldots,R_{n_\mathsf{W}}$ of the lists corresponding to all keywords $w_1,\ldots,w_{n_\mathsf{W}}$, respectively, the scheme invokes the $\mathsf{Allocation}$ algorithm on these sets (and on the respective lengths of the lists) to obtain the *actual locations* for the elements of all lists. We denote the actual locations as an array $\mathsf{map}\leftarrow\mathsf{Allocation}\left((n_1,R_1),\ldots,(n_{n_\mathsf{W}},R_{n_\mathsf{W}})\right)$, where each of its entries is either a pair (i,j) (representing that this entry is the actual location of the jth element from the list $\mathsf{DB}(w_i)$) or NULL (representing an empty entry).

A key property of this framework is that the $\mathsf{RangesGen}$ algorithm, which determines the set of possible locations for each list $\mathsf{DB}(w_i)$, is applied separately and independently to the length of each list. Thus, the possible locations of each list are independent of the possible locations of all other lists (in contrast, the actual locations of the lists are naturally allowed to be correlated).

Asharov et al. referred to a pair $(\mathsf{RangesGen},\mathsf{Allocation})$ of such algorithms as an allocation scheme, and showed that any such allocation scheme satisfying a

natural correctness requirement can be used to construct a searchable symmetric encryption scheme. The correctness requirement asks that for any database, with all but a negligible probability, these algorithms produce an actual allocation map in which each element has exactly one actual placement (where the probability is taken over the internal coin tosses of the algorithms $\mathsf{RangesGen}$ and $\mathsf{Allocation}$).

The resulting scheme of Asharov et al. inherits its space, locality and read efficiency from those of its underlying allocation scheme, defined as follows:

Definition 4.3. *A pair* $(\mathsf{RangesGen}, \mathsf{Allocation})$ *of algorithms satisfying the above correctness requirement is an* $(\mathsf{s}, \mathsf{d}, \mathsf{r})$*-allocation scheme, for some functions* $\mathsf{s}(\cdot)$, $\mathsf{d}(\cdot)$ *and* $\mathsf{r}(\cdot,\cdot)$*, if the following properties hold:*

- **Space:** *For any input* $(n_1, \ldots, n_k)$*, the array* $\mathsf{map} \leftarrow \mathsf{Allocation}((n_1, R_1), \ldots, (n_k, R_k))$*, where* $R_i = \{[a_1, b_1], \ldots, [a_{\mathsf{d}}, b_{\mathsf{d}}]\} \leftarrow \mathsf{RangesGen}(N, n_i)$ *for every* $i \in [k]$*, is of size at most* $\mathsf{s}(N)$*, where* $N = \sum_{i=1}^{k} n_i$*.*
- **Locality:** *For any input* (N, n_i)*, the algorithm* $\mathsf{RangesGen}$ *outputs at most* $\mathsf{d}(N)$ *ranges.*
- **Read efficiency:** *For any input* (N, n_i) *for the algorithm* $\mathsf{RangesGen}$ *it holds that:*

$$\frac{\sum_{j=1}^{\mathsf{d}} (b_j - a_j + 1)}{n_i} \leq \mathsf{r}(N, n_i) \ ,$$

where $\{[a_1, b_1], \ldots, [a_{\mathsf{d}}, b_{\mathsf{d}}]\} \leftarrow \mathsf{RangesGen}(N, n_i)$*.*

Equipped with the above notation, Asharov et al. proved the following:

Theorem 4.4 ([2]). *Given any* $(\mathsf{s}, \mathsf{d}, \mathsf{r})$*-allocation scheme and any one-way function, there exists an* $\mathcal{L}_{\mathsf{size}}^{\mathsf{adap}}$*-secure searchable symmetric encryption scheme for databases of size* N *with space* $O(\mathsf{s}(N))$*, locality* $O(\mathsf{d}(N))$*, and read efficiency* $O(\mathsf{r}(N,\cdot))$*.*

From allocation algorithms to SSE schemes. We conclude our high-level description of the statistical-independence framework by briefly overviewing the generic transformation from allocation schemes to SSE scheme. The reader is referred to [2] for the complete formal description of this transformation.

In a nutshell, the client runs the $\mathsf{RangesGen}$ and the $\mathsf{Allocation}$ procedures as described above to obtain the actual allocation map of all elements. Then, the client encrypts each identifier from each list $\mathsf{DB}(w)$ in map with a key that is derived from the keyword w using a pseudorandom function. In addition, any unused entry in the array is filled with a uniform string of the appropriate length.

When issuing a query corresponding to a keyword w, the client asks the server to retrieve the encrypted content of all possible locations of the list $\mathsf{DB}(w)$.[10]

[10] The details here are quite subtle. The server obtains the pseudorandom key that was used to produce randomness for the relevant invocation of $\mathsf{RangesGen}$. In addition, the server stores the lengths of the lists in an encrypted manner, and can only reveal the lengths of the already-queried lists. Knowing both the pseudorandom key and the list length allows the server to compute the possible locations of the list $\mathsf{DB}(w)$.

Since these locations are chosen independently at random, this does not reveal any additional information on the structure of the database except for the length of the queried list. The client then identifies the actual locations and decrypts the data by itself.

4.2 Our Leveled Two-Choice Scheme

In this section we present our new allocation scheme from which Theorem 4.4 provides the searchable symmetric encryption schemes guaranteed by Theorem 4.1. Our scheme consists of the following three levels for storing the elements of any given database DB of size N:

- The first level, named the "two-choice array", consists of the two-choice SSE scheme of Asharov et al. [2] but with *an exponentially improved read efficiency*. In this array, each list $\mathsf{DB}(w_i)$ can be stored in one out of two possible intervals of consecutive locations, in a manner that we describe below as part of our $\mathsf{Allocation}$ algorithm. However, unlike the scheme of Asharov et al. we do not store all of the N elements of the database in this array. Instead, the key observation underlying our new scheme is that when viewing this array as a collection of bins, then by allowing a few lists to "overflow" from this level to the second level (overall at most $\widehat{N} = N/\log N$ elements will overflow with all but a negligible probability), we can reduce the maximal load of each bin from $\tilde{O}(\log\log N)$ (as in [2]) to $O(\log\log\log N)$. This then translates into improving the read efficiency in this level from $\tilde{O}(\log\log N)$ to $O(\log\log\log N)$.
- The second level, named the "cuckoo hashing level", stores the vast majority of the elements that overflow from the first level. This level consists of roughly $\log N$ cuckoo hashing tables (see Sect. 2.3), where the j hash table is designed to store at most $\widehat{N}/2^j$ values each of which of size 2^j. These values are the lists that overflow from the first level (the jth table will store overflowing lists of length roughly 2^j).
- The third level, named the "stash level", consists of a cuckoo hashing stash for each of the second-level cuckoo hashing tables. The goal of introducing this level is to reduce the failure probably of cuckoo hashing from noticeable to negligible (see Sect. 2.3), which is essential for the security of the resulting SSE scheme.

This leveled structure of our allocation scheme, and thus of our SSE scheme, guarantees that the possible locations for a list $\mathsf{DB}(w)$ of length n are its two possible intervals in the two-choice array, its two locations in the jth cuckoo hashing table for $j = \log n$, and anywhere in the stash of the jth cuckoo hashing table. In what follows we formally describe our allocation scheme (see Algorithm 4.7), which we prove to have space $O(N)$, locality 5, and read efficiency $\omega(1) \cdot \epsilon(n)^{-1} + O(\log\log\log N)$ when retrieving lists of length $n = N^{1-\epsilon(n)}$.

Theorem 4.5. *For any function $f(N) = \omega(1)$, Algorithm 4.7 describes an $(O(N), 5, \mathsf{r}(N,n))$-allocation scheme for databases of size N in which no keyword is associated with more than $N/\log^3 N$ identifiers, where $\mathsf{r}(N,n) = f(N) \cdot \epsilon(n)^{-1} + O(\log\log\log N)$ and $n = N^{1-\epsilon(n)}$.*

Proof of Theorem 4.5. We assume without loss of generality that $f(N) = o(\log\log N)$ (since otherwise, we may take $\tilde{f}(N) = \min(f(N), o(\log\log N))$ instead). For the two-choice part of the algorithm, we make use of the following theorem from [2].

Theorem 4.6 ([2] Theorem 3.5 Part 1)**.** *Let $S \geq n_1$ be a bound on the maximal length, and let m be the number of bins. Consider the two-choice allocation algorithm. Then, with probability $1 - N^{-\Omega(\log N)}$, there are at most $S\log^2 N$ elements at level greater than $\frac{4N}{m} + \log\log\frac{N}{S} + 2$, where the level of an element is the load of its bin right after inserting the element (e.g., the first element that is interested to the bin has level 1).*

In Algorithm 4.7, we set $S = N/\log^3 N$, $m = N/\log\log\log N$, and $\mathsf{BinSize} = O(\log\log\log N)$. Therefore, with an overwhelming probability there are at most $\widehat{N} = N/\log N$ overflowing elements, and in this case, we place at most $\hat{N}$ elements in the cuckoo hashing tables with the stashes.

Now we analyze the placement of the elements in the hash tables, assuming that the number of elements in $\mathsf{LeftOvers}$ is at most $\widehat{N}$. For each $0 \leq j \leq t$, we set the stash size $s_j = f(N) \cdot \epsilon_j^{-1}$ where ϵ_j is chosen such that $2^j = N^{1-\epsilon_j}$. We obtain that the algorithm fails to insert the lists into the cuckoo hash table H_j with its stash with probability at most $O((\widehat{N}/2^j)^{-s_j/2})$ (see Sect. 2.3). Note that $N^{\epsilon_j} \geq \log^3 N$, so it holds that

$$\begin{aligned}(\widehat{N}/2^j)^{-s_i/2} &= (N^{\epsilon_j}/\log N)^{-s_j/2} \\ &\leq (N^{\frac{2}{3}\epsilon_j})^{-s_j/2} \\ &= N^{-f(N)/3}.\end{aligned}$$

Thus, the insertion of overflowing elements fails with a negligible probability, and we conclude that Algorithm 4.7 fulfills the correctness requirement. Regarding read efficiency, the overhead of the 2-choice is $O(\log\log\log N)$, the overhead of the cuckoo hash table is 2, and the overhead of the stash is $f(N) \cdot \epsilon(n)^{-1}$, where $n = N^{1-\epsilon(n)}$, so in total we get an overhead of $f(N) \cdot \epsilon_i^{-1} + O(\log\log\log N)$ as claimed. Locality of 5 easily follows from the description of $\mathsf{SplitList}$. Regarding the space overhead, the bins require space of $m \cdot \mathsf{BinSize} = O(N)$, each cuckoo hash table with stash requires space of $O(\widehat{N}) = O(N/\log N)$, and there are less than $\log N$ tables. So in total, the space overhead is $O(N)$. ■

ALGORITHM 4.7 (Our Allocation Scheme (RangesGen, Allocation))

Input: A vector of integers $(n_1, \ldots, n_k)$ representing the lengths of the lists $L_1, \ldots, L_k$ in the database. We let $N = \sum_{i=1}^{k} n_i$, $\widehat{N} = N/\log N$, and assume for concreteness that the n_i's are powers of 2, and that $n_1 \geq n_2 \geq \cdots \geq n_k$.

Parameters:

- A bound $S = N/\log^3 N$ on the length of the longest list in the database.
- The number $m = N/\log\log\log N$ of bins in the two-choice array (it is chosen as a power of 2 and such that $m \geq n_1$).
- A bound $\mathsf{BinSize} = O(\log\log\log N)$ on the size of each bin in the two-choice array.
- Stash sizes $s_0, \ldots, s_t$ where $t = \log S$ and $s_j = f(N) \cdot \epsilon_j$ for every $j \in [t]$, where $2^j = N^{1-\epsilon_j}$ and $\omega(1) \leq f(N) \leq o(\log\log N)$ may be any pre-specified function.

The memory layout. The memory is partitioned into the following segments:

1. m bins $B_0, \ldots, B_{m-1}$, each of size BinSize.
2. Hash tables $H_0, \ldots, H_t$, where each hash table H_j is implemented as a cuckoo hash table for $\widehat{N}/2^j$ data items of size 2^j each with a stash of size s_j.

The RangesGen **algorithm.** On input N and n_i:

1. Uniformly sample $\alpha_{i,1}, \alpha_{i,2} \leftarrow \{0, \ldots, \frac{m}{n_i} - 1\}$.
   Consider the two super bins $\widetilde{B}_{\alpha_{i,1}} = (B_{n_i \cdot \alpha_1 + j})_{j=0}^{n_i - 1}$ and $\widetilde{B}_{\alpha_{i,2}} = (B_{n_i \cdot \alpha_2 + j})_{j=0}^{n_i - 1}$.
2. Sample two hash table locations $\beta_{i,1}, \beta_{i,2}$ for the cuckoo hash table $H_{\log n_i}$.
3. The possible ranges R_i are (1) The above two super-bins; (2) The two cells $\beta_{i,1}, \beta_{i,2}$ in the hashtable $H_{\log n_i}$; (3) The stash of the table $H_{\log n_i}$.

The Allocation **algorithm.**

1. Initialize m empty bins $B_0, \ldots, B_{m-1}$, and an empty set LeftOvers.
2. Initialize hash tables $H_0, \ldots, H_t$, where each hash table H_j is implemented as a cuckoo hash table for $\widehat{N}/2^j$ entries of size 2^j with a stash of size s_j.
3. For every list L_i with size n_i and ranges R_i, reconstruct $(\alpha_{i,1}, \alpha_{i,2})$ and $(\beta_{i,1}, \beta_{i_2})$ from R_i, and place the list L_i as follows:
   (a) Consider the two super bins $\widetilde{B}_{\alpha_{i,1}} = \left(B_{n_i \cdot \alpha_{i,1} + j}\right)_{j=0}^{n_i - 1}$ and $\widetilde{B}_{\alpha_{i,2}} = \left(B_{n_i \cdot \alpha_{i,2} + j}\right)_{j=0}^{n_i - 1}$. Let $\beta \in \{\alpha_{i,1}, \alpha_{i,2}\}$ be the index of the least loaded super bin among $\widetilde{B}_{\alpha_{i,1}}$ and $\widetilde{B}_{\alpha_{i,2}}$, where the load of a super bin is defined as the sum of loads of the bins that constitutes that super bin. If the load of the bins in $\widetilde{B}_\beta$ is BinSize, then add L_i to LeftOvers. Otherwise, place the list L_i in the super bin $\widetilde{B}_\beta$. That is, for every $j = 0, \ldots, n_i - 1$, place the jth element of the list L_i in the bin $B_{n_i \cdot \beta + j}$.
   (b) If the list was not placed, then insert L_i into the cuckoo hash table $H_{\log n_i}$ using the locations $\beta_{i,1}$ and $\beta_{i,2}$. Note that the list might be placed in the stash. If the insertion fails, then output $\perp$ and abort.

References

1. Arbitman, Y., Naor, M., Segev, G.: Backyard cuckoo hashing: constant worst-case operations with a succinct representation. In: Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science, pp. 787–796 (2010)
2. Asharov, G., Naor, M., Segev, G., Shahaf, I.: Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In: Proceedings of the 48th Annual ACM Symposium on Theory of Computing, pp. 1101–1114 (2016)
3. Asharov, G., Segev, G., Shahaf, I.: Tight tradeoffs in searchable symmetric encryption. Cryptology ePrint Archive, Report 2018/507 (2018). https://eprint.iacr.org/2018/507
4. Aumüller, M., Dietzfelbinger, M., Woelfel, P.: Explicit and efficient hash families suffice for cuckoo hashing with a stash. Algorithmica **70**(3), 428–456 (2014)
5. Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: Proceedings of the 22nd ACM Conference on Computer and Communications Security, pp. 668–679 (2015)
6. Cash, D., Jaeger, J., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M., Steiner, M.: Dynamic searchable encryption in very-large databases: data structures and implementation. In: Proceedings of the 21st Annual Network and Distributed System Security Symposium (2014)
7. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.-C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8042, pp. 353–373. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40041-4_20
8. Cash, D., Tessaro, S.: The locality of searchable symmetric encryption. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 351–368. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55220-5_20
9. Chang, Y.-C., Mitzenmacher, M.: Privacy preserving keyword searches on remote encrypted data. In: Ioannidis, J., Keromytis, A., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 442–455. Springer, Heidelberg (2005). https://doi.org/10.1007/11496137_30
10. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 577–594. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17373-8_33
11. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, pp. 79–88 (2006)
12. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: Improved definitions and efficient constructions. J. Comput. Secur. **19**(5), 895–934 (2011)
13. Demertzis, I. Papadopoulos, D., Papamanthou, C.: Searchable encryption with optimal locality: achieving sublogarithmic read efficiency. Cryptology ePrint Archive, Report 2017/749 (2017)
14. Demertzis, I., Papamanthou, C.: Fast searchable encryption with tunable locality. In: Proceedings of the 2017 ACM Special Interest Group on Management of Data (SIGMOD) Conference, pp. 1053–1067 (2017)
15. Dietzfelbinger, M., Pagh, R.: Succinct data structures for retrieval and approximate membership. In: Proceedings of the 35th International Colloquium on Automata, Languages and Programming, pp. 385–396 (2008)

16. Goh, E.: Secure indexes. Cryptology ePrint Archive, Report 2003/216 (2003)
17. Hagerup, T.: Sorting and searching on the word RAM. In: Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science, pp. 366–398 (1998)
18. Hagerup, T., Miltersen, P.B., Pagh, R.: Deterministic dictionaries. J. Algorithms **41**(1), 69–85 (2001)
19. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Proceedings of the 16th International Conference on Financial Cryptography and Data Security, pp. 258–274 (2013)
20. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Proceedings of the 19th ACM Conference on Computer and Communications Security, pp. 965–976 (2012)
21. Kirsch, A., Mitzenmacher, M., Wieder, U.: More robust hashing: cuckoo hashing with a stash. SIAM J. Comput. **39**(4), 1543–1561 (2009)
22. Kurosawa, K., Ohtaki, Y.: UC-secure searchable symmetric encryption. In: Keromytis, A.D. (ed.) FC 2012. LNCS, vol. 7397, pp. 285–298. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32946-3_21
23. Kurosawa, K., Ohtaki, Y.: How to update documents verifiably in searchable symmetric encryption. In: Proceedings of the 12th International Conference on Cryptology and Network Security, pp. 309–328 (2013)
24. Miltersen, P.B.: Cell probe complexity - a survey. In: Proceedings of the 19th Conference on the Foundations of Software Technology and Theoretical Computer Science, Advances in Data Structures Workshop (1999)
25. Pagh, A., Pagh, R.: Uniform hashing in constant time and optimal space. SIAM J. Comput. **38**(1), 85–96 (2008)
26. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms **51**(2), 122–144 (2004)
27. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceedings of the 21st Annual IEEE Symposium on Security and Privacy, pp. 44–55 (2000)
28. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: Proceedings of the 21st Annual Network and Distributed System Security Symposium (2014)
29. van Liesdonk, P., Sedghi, S., Doumen, J., Hartel, P.H., Jonker, W.: Computationally efficient searchable symmetric encryption. In: Proceedings of 7th VLDB Workshop on Secure Data Management, pp. 87–100 (2010)