SSE and SSD: Page-Efficient Searchable Symmetric Encryption

Angèle Bossuat[1], Raphael Bost[2,⋆], Pierre-Alain Fouque[1], Brice Minaud[3,4], and Michael Reichle[3,4]

[1] Quarkslab, and Université de Rennes 1, Rennes, France.
[2] Direction Générale de l'Armement, Paris, France.
[3] Inria, Paris, France.
[4] École Normale Supérieure, CNRS, PSL, Paris, France.

Abstract. Searchable Symmetric Encryption (SSE) enables a client to outsource a database to an untrusted server, while retaining the ability to securely search the data. The performance bottleneck of classic SSE schemes typically does not come from their fast, symmetric cryptographic operations, but rather from the cost of memory accesses. To address this issue, many works in the literature have considered the notion of locality, a simple design criterion that helps capture the cost of memory accesses in traditional storage media, such as Hard Disk Drives. A common thread among many SSE schemes aiming to improve locality is that they are built on top of new memory allocation schemes, which form the technical core of the constructions.

The starting observation of this work is that for newer storage media such as Solid State Drives (SSDs), which have become increasingly common, locality is not a good predictor of practical performance. Instead, SSD performance mainly depends on *page efficiency*, that is, reading as few pages as possible. We define this notion, and identify a simple memory allocation problem, *Data-Independent Packing* (DIP), that captures the main technical challenge required to build page-efficient SSE. As our main result, we build a page-efficient and storage-efficient data-independent packing scheme, and deduce the Tethys SSE scheme, the first SSE scheme to achieve at once $\mathcal{O}(1)$ page efficiency and $\mathcal{O}(1)$ storage efficiency. The technical core of the result is a new generalization of cuckoo hashing to items of variable size. Practical experiments show that this new approach achieves excellent performance.

1 Introduction

In Searchable Symmetric Encryption (SSE), a client holds a collection of documents, and wishes to store them on an untrusted cloud server. The client also wishes to be able to issue search queries to the server, and retrieve all documents

⋆ The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the DGA or the French Government.

matching the query. Meanwhile, the honest-but-curious server should learn as little information as possible about the client's data and queries. Searchable Encryption is an important goal in the area of cloud storage, since the ability to search over an outsourced database is often a critical feature. The goal of SSE is to enable that functionality, while offering precise guarantees regarding the privacy of the client's data and queries with respect to the host server.

Compared to other settings related to computation over encrypted data, such as Fully Homomorphic Encryption, a specificity of SSE literature is the focus on high-performance solutions, suitable for deployment on large real-world datasets. To achieve this performance, SSE schemes accept the leakage of some information on the plaintext dataset, captured in security proofs by a leakage function. The leakage function is composed of setup leakage and query leakage. The setup leakage is the total leakage prior to query execution, and typically reveals the size of the index, and possibly the number of searchable keywords. For a static scheme, the query leakage usually reveals the repetition of queries, and the set of document indices matching the query. Informally, the security model guarantees that the adversary does not learn any information about the client's data and queries, other than the previous leakages.

In particular, the allowed leakage typically reveals nothing about keywords that have *not yet* been queried. Although this requirement may seem natural and innocuous, it has deep implications about the storage and memory accesses of SSE schemes. At Eurocrypt 2014, Cash and Tessaro [CT14] proved an impossibility result that may be roughly summarized as follows. If an SSE scheme reveals nothing about the number of documents matching *unqueried* keywords, then it cannot satisfy the following three efficiency properties simultaneously: (1) constant storage efficiency: the size of the encrypted database is at most linear in the size of the plaintext data; (2) constant read efficiency: the amount of data read by the server to answer a query is at most linear in the size of the plaintext answer; (3) constant locality: the memory accesses made by the server to answer a query consist of a constant number of contiguous accesses. Thus, a secure SSE scheme with constant storage efficiency and read efficiency cannot be *local*: it must perform a superconstant number of disjoint memory accesses.

In practice, many SSE schemes (*e.g.* [CGKO06, CJJ$^+$13, Bos16]) make one random memory access per entry matching the search query. As explained in [CJJ$^+$14, MM17], making many small random accesses hampers performance: hard disks drives (HDD) were designed for large sequential accesses, and solid state drives (SSD) use a leveled design that does not accommodate small reads well. As discussed *e.g.* in [BMO17], this results in the fact that in many settings, the performance bottleneck for SSE is not the cost of cryptographic operations (which rely on fast, symmetric primitives), but the cost of memory accesses.

As a consequence, SSE scheme designers have tried to reduce the number of disk accesses needed to process a search query, *e.g.* by grouping entries corresponding to the same keywords in blocks [CJJ$^+$14, MM17], or by using more complex allocation mechanisms [ANSS16, ASS18, DPP18]. However, no optimal solution is possible, due to the previously mentioned impossibility result of Cash and Tessaro. In the static case, the first construction by Asharov *et al.* from STOC 2016 achieves linear server storage and constant locality, at the cost of logarithmic read efficiency (the amount of data read by the server to answer a query is bounded by the size of the plaintext answer times $\mathcal{O}(\log N)$, where N is the size of the plaintext database) [ANSS16]. The logarithmic factor was reduced to $\log^{\gamma} N$ for $\gamma < 1$ by Demertzis *et al.* at Crypto 2018 [DPP18].

An interesting side-effect of this line of research is that it has highlighted the connection between Searchable Encryption and memory allocation schemes with certain security properties. The construction from [ANSS16] relies on a two-dimensional variant of the classic balls-and-bins allocation problem. Likewise, the construction from [DPP18] uses several memory allocation schemes tailored to different input sizes.

1.1 Overview of Contributions

As discussed above, memory accesses are a critical bottleneck for SSE performance. This has led to the notion of locality, and the construction of many SSE schemes aiming to improve locality, such as [CT14, ANSS16, MM17, DP17, DPP18]. The motivation behind the notion of locality is that it is a simple criterion that captures the performance of traditional storage media such as HDDs. In recent years, other storage media, and especially SSDs, have become more and more prevalent. To illustrate that point, the number of SSDs shipped worldwide is projected to overtake HDD shipments in 2021 [Sta21].

However, locality as a design target, was proposed assuming an implementation on a HDD. The starting point of our work is that for SSDs, locality is no longer a good predictor of practical performance. This raises two questions: first, is there a simple SSE design criterion to capture SSD performance, similar to locality for HDDs? And can we design SSE schemes that fulfill that criterion?

The answer to the first question is straightforward: for SSDs, performance is mainly determined by the number of memory pages that are accessed, regardless of whether they are contiguous. This leads us to introduce the notion of page efficiency. The page efficiency of an SSE scheme is simply the number of pages that the server must access to process a query, divided by the number of pages of the plaintext answer to the query. Page efficiency is an excellent predictor of SSD performance. This is supported by experiments in Section 5. Some of the technical reasons behind that behavior are also discussed in the full version.

The main contribution of this work is to give a positive answer to the second question, by building a page-efficiency SSE scheme, called Tethys. Tethys achieves page efficiency $\mathcal{O}(1)$ and storage efficiency $\mathcal{O}(1)$, with minimal leakage. Here, $\mathcal{O}(1)$ denotes an absolute constant, independent of not only the database size, but also the page size. We also construct two additional variants, Pluto and Nilus_t, that offer practical trade-offs between server storage and page efficiency. An overview of these schemes is presented on Table 1, together with a comparison with some relevant schemes from the literature.

Similar to local SSE schemes such as [ANSS16] and its follow-ups, the core technique underpinning our results is a new memory allocation scheme. In order

Table 1 – Trade-offs between SSE schemes. Here, p is the number elements per page, k is the number of keywords, and λ is the security parameter (assuming $k \geq \lambda$). *Page cost* $aX + b$ means that in order to process a query whose plaintext answer is at most X pages long, the server needs to access at most $aX + b$ memory pages. *Page efficiency* is page cost divided by X in the worst case. *Client storage* is the size of client storage, where the unit is the storage of one element or address. *Storage efficiency* is the number of pages needed to store the encrypted database, divided by the number of pages of the plaintext database.

Schemes	Client st.	Page cost	Page eff.	Storage eff.	Source
Π_{bas}	$\mathcal{O}(1)$	$\mathcal{O}(Xp)$	$\mathcal{O}(p)$	$\mathcal{O}(1)$	[CJJ$^+$14]
$\Pi_{\text{pack}}, \Pi_{\text{2lev}}$	$\mathcal{O}(1)$	$\mathcal{O}(X)$	$\mathcal{O}(1)$	$\mathcal{O}(p)$	[CJJ$^+$14]
1-Choice	$\mathcal{O}(1)$	$\widetilde{\mathcal{O}}(\log N)\, X$	$\widetilde{\mathcal{O}}(\log N)$	$\mathcal{O}(1)$	[ANSS16]
2-Choice	$\mathcal{O}(1)$	$\widetilde{\mathcal{O}}(\log\log N)\, X$	$\widetilde{\mathcal{O}}(\log\log N)$	$\mathcal{O}(1)$	[ANSS16]
Tethys	$\mathcal{O}(p \log \lambda)$	$2X + 1$	3	$3 + \varepsilon$	Section 4
Pluto	$\mathcal{O}(p \log \lambda)$	$X + 2$	3	$3 + \varepsilon$	Full version
Nilus_t	$\mathcal{O}(p \log \lambda)$	$2tX + 1$	$2t + 1$	$1 + (2/e)^{t-1}$	Full version

to build Tethys, we identify and extract an underlying combinatorial problem, which we call *Data-Independent Packing* (DIP). We show that a secure SSE scheme can be obtained generically from any DIP scheme, and build Tethys in that manner.

Similar to standard bin packing, the problem faced by a DIP scheme is to pack items of variable size into buckets of fixed size, in such a way that not too much space is wasted. At the same time, data independence requires that a given item can be retrieved by inspecting a few buckets whose location is independent of the sizes of other items. That may seem almost contradictory at first: we want to pack items closely together, in a way that does not depend on item sizes. The solution we propose is inspired by a generalization of cuckoo hashing, discussed in the technical overview below.

We note that the DIP scheme we build in this way has other applications beyond the scope of this article. One side result is that it can also be used to reduce the leakage of the SSE scheme with tunable locality from [DP17]. Also, we sketch a construction for a length-hiding static ORAM scheme that only has constant storage overhead.

Finally, we have implemented Tethys to analyze its practical performance. The source code is publicly available (link in Section 5). The experiments show two things. First, experimental observations match the behavior predicted by the theory. Second, when benchmarked against various existing static SSE schemes, Tethys achieves, to our knowledge, unprecedented performance on SSDs: without having to rely on a very large ciphertext expansion factor (less than 3 in our experiments), we are able to stream encrypted entries and decrypt them from a medium-end SSD at around half the raw throughput of that SSD.

1.2 Technical Overview

In single-keyword SSE schemes, the encrypted database is realized as an inverted index. The index maps each keyword to the (encrypted) list of matching document indices. The central question is how to efficiently store these lists, so that accessing some lists reveals no information about the lengths of other lists.

Page efficiency asks that in order to retrieve a given list, we should have to visit as few pages as possible. The simplest solution for that purpose is to pad all lists to the next multiple of one page, then store each one-page chunk separately using a standard hashing scheme. That padding approach is used in some classic SSE constructions, such as [CJJ$^+$14]. While the approach is page-efficient, it is not storage-efficient, since all lists need to be padded to the next multiple of p.

In practice, with a standard page size of 4096 bytes, and assuming 64-bit document indices, we have $p = 512$. Regardless of the size of the database, if it is the case that most keywords match few documents, say, less than 10 documents, then server storage would blow up by a factor 50. More generally, whenever the dataset contains a large ratio of small lists, padding becomes quite costly, up to a factor $p = 512$ in storage in the worst case. Instead, we would like to upper-bound the storage blowup by a small constant, independent of both the input dataset, and the page size.

Another natural approach is to adapt SSE schemes that target locality. It is not difficult to show that an SSE scheme with locality L and read efficiency R has page efficiency $\mathcal{O}(L+R)$ (Theorem 2.1). However, due to Cash and Tessaro's impossibility result, it is not possible for any scheme with constant storage efficiency and locality $\mathcal{O}(1)$ (such as [ANSS16] and its follow-ups) to have read efficiency $\mathcal{O}(1)$; and all such schemes result in superconstant page efficiency.

Ultimately, a new approach is needed. To that end, we first introduce the notion of *data-independent packing* (DIP). A DIP scheme is a purely combinatorial allocation mechanism, which assigns lists of variable size into buckets of fixed size p. (Our definition also allows to store a few extra items in a stash.) The key property of a DIP scheme is data independence: each list can be retrieved by visiting a few buckets, whose locations are independent of the sizes of other lists.

We show that a secure SSE scheme $\mathsf{SSE(D)}$ can be built generically from any DIP scheme D. The page efficiency and storage efficiency of the SSE scheme $\mathsf{SSE(D)}$ can be derived directly from similar efficiency measures for the underlying DIP scheme D. All SSE schemes in this paper are built in that manner.

We then turn to the question of building an efficient DIP scheme. Combinatorially, what we want is a DIP scheme with constant page efficiency (the number of buckets it visits to retrieve a list is bounded linearly by the number of buckets required to read the list), and constant storage efficiency (the total number of buckets it uses is bounded linearly by the number of buckets required to store all input data contiguously). The solution we propose, $\mathsf{TethysDIP}$, is inspired by cuckoo hashing. For ease of exposition, we focus on lists of size at most one page. Each list is assigned two uniformly random buckets as possible destinations. It is required that the full list can be recovered by reading the two buckets, plus a stash. To ensure data independence, all three locations are accessed, regardless of where list elements are actually stored. Since the two buckets for each list are drawn independently and uniformly at random, data independence is immediate.

Once each list is assigned its two possible buckets, we are faced with two problems. The first problem is algorithmic: how should each list be split between its two destination buckets and the stash, so that the stash is as small as possible, subject to the constraint that the assignment is correct (all list elements are stored, no bucket receives more than p elements)? We prove that a simple max flow computation yields an optimal solution to this optimization problem. To see this, view buckets as nodes in a graph, with lists corresponding to edges between their two destination buckets, weighted by the size of the list. Intuitively, if we start from an arbitrary assignment of items to buckets, we want to find as many disjoint paths as possible going from overfull buckets to underfull buckets, so that we can "push" items along those paths. This is precisely what a max flow algorithm provides.

The second (and harder) problem we face is analytic: can we prove that a valid assignment exists with overwhelming probability, using only $\mathcal{O}(n/p)$ buckets (for constant storage efficiency), and a stash size independent of the database size? Note that a negligible probability of failure is critical for security, because the probability of failure depends on the list length distribution, which we wish to hide. Having a small stash size, that does not grow with the database size, is also important, because in the final SSE scheme, we will ultimately store the stash on the client side.

In the case of cuckoo hashing, results along those lines are known, see for example [ADW14]. However, our situation is substantially different. Cuckoo hashing with buckets of capacity $p > 1$ has been analyzed in the literature [DW05], including in the presence of a stash [KMW10]. Such results go through the analysis of the *cuckoo graph* associated with the problem: similar to the graph discussed earlier, vertices are buckets, and each item gives rise to one edge connecting the two buckets where it can be assigned. A crucial difference in our setting compared to regular cuckoo hashing with buckets of capacity p is that edges are not uniformly distributed. Instead, each list of length x generates x edges between the same two buckets.

Thus, we need an upper bound that holds for a family of non-uniform edge distributions (those that arise from an arbitrary number of lists with an arbitrary number of elements each, subject only to the total number of elements being equal to the database size n). Moreover, we want an upper bound that holds simultaneously for all members of that family, since we want to hide the length distribution. What we show is that the probability of failure for any such distribution can be upper-bounded by the case where all lists have the maximum size p, up to a polynomial factor. Roughly speaking, this follows from a convexity argument, combined with a majorization argument, although the details are intricate. We are then able to adapt existing analyses for the standard cuckoo graph.

In the end, $\mathsf{TethysDIP}$ has the following features: every item can be retrieved by visiting 2 data-independent table locations (and the stash), the storage efficiency is $2+\varepsilon$, and the stash size is $p\omega(\log\lambda)$. All those quantities are the same as regular cuckoo hashing, up to a scaling factor p in the stash size, which is unavoidable (see full version for more details). Since regular cuckoo hashing is a special case of our setting, the result is tight.

In the full version, we present two other DIP schemes, $\mathsf{PlutoDIP}$ and $\mathsf{NilusDIP}_t$. Both are variants of the main $\mathsf{TethysDIP}$ construction, and offer trade-offs of practical interest between storage efficiency and page efficiency. In particular, $\mathsf{NilusDIP}$ rests on the observation that our main analytical results, regarding the optimality and stash size bound of $\mathsf{TethysDIP}$, can be generalized to buckets of size tp rather than p, for an arbitrary integer t. This extension yields a storage efficiency $1+(2/e)^{t-1}$, which tends exponentially fast towards the information theoretical minimum of 1. The price to pay is that page efficiency is $2t$, because we need to visit two buckets, each containing t pages, to retrieve a list.

1.3 Related Work

Our work mainly relates to two areas: SSE and cuckoo hashing. We discuss each in turn.

In [ANSS16], Asharov *et al.* were the first to explicitly view SSE schemes as an allocation problem. That view allows for very efficient schemes, and is coherent with the fact that the main bottleneck is the IO and not the cryptographic overhead, as observed by Cash *et al.* [CJJ+13]. Our work uses the same approach, and builds an SSE scheme on top of an allocation scheme.

As proved by Cash and Tessaro [CT14], no SSE scheme can be optimal simultaneously in locality, read efficiency, and storage efficiency (see also [ASS18]). Since then, many papers have constructed schemes with constant locality and storage efficiency, while progressively improving read efficiency: starting from $\mathcal{O}(\log N \log\log N)$ in [ANSS16] to $\mathcal{O}(\log^\gamma N)$ in [DPP18] for any fixed $\gamma > 2/3$, and finally $\mathcal{O}(\log\log N \log^2 \log\log N)$ when all lists have at most $N^{1-1/\log\log N}$ entries [ANSS16], or $\mathcal{O}(\log\log\log N)$ when they have at most $N^{1-1/o(\log\log\log N)}$ entries [ASS18]. On the other hand, some constructions achieve optimal read efficiency, and sublinear locality, at the cost of increased storage, such as the family of schemes by Papamanthou and Demertzis [DP17].

Contrary to the previous line of work, we aim to optimize page efficiency and not locality. At a high level, there is a connection between the two: both aim to store the data matching a query in close proximity. A concrete connection is given in Theorem 2.1. Nevertheless, to our knowledge, no previous SSE scheme with linear storage has achieved page efficiency $\mathcal{O}(1)$. The Π_{pack} scheme from [CJJ+14] achieves page efficiency $\mathcal{O}(1)$ by padding all lists to a multiple of the page size, and storing lists by chunks of one page. However, this approach has storage efficiency p in the worst case. The Π_{2lev} variant from [CJJ+14] incurs the same cost, because it handles short lists in the same way as Π_{pack}. In practice, such schemes will perform well for long lists, but will incur a factor up to p when there are many small lists, which can be prohibitive, as a typical value of p is $p = 512$ (*cf.* Section 1.2). On the other hand, Π_{pack} and its variants are dynamic schemes, whereas Tethys is static.

$\mathsf{TethysDIP}$ is related to one of the allocation schemes from [DPP18], which uses results by Sanders *et al.* [SEK03]. That allocation scheme can be generalized to handle the same problem as $\mathsf{TethysDIP}$, but we see no way of doing so that would achieve storage and page efficiency $\mathcal{O}(1)$. Another notable difference is that we allow for a stash, which makes it possible to achieve a negligible probability of failure (the associated analysis being the most technically challenging part of this work). An interesting relationship between our algorithm in the algorithm from [SEK03] is discussed in Section 4.1.

As Data-Indepdent Packing scheme, $\mathsf{TethysDIP}$ is naturally viewed as a packing algorithm with oblivious lookups. The connection between SSE and oblivious algorithms is well-known, and recent works have studied SSE with fully oblivious accesses [MPC+18, KMO18].

We now turn to cuckoo hashing [PR04]. As noted earlier, $\mathsf{TethysDIP}$ (resp. $\mathsf{NilusDIP}_t$) includes standard cuckoo hashing with a stash (resp. with buckets of size $t > 1$) as a special case, and naturally extends those settings to items of variable size. Moreover, our proof strategy essentially reduces the probability of failure of $\mathsf{TethysDIP}$ (resp. $\mathsf{NilusDIP}_t$) to their respective cuckoo hashing special cases. As such, our work relies on the cuckoo hashing literature, especially works on bounding stash sizes [KMW10, ADW14]. While $\mathsf{TethysDIP}$ generalizes some of these results to items of variable size, we only consider the static setting. Extending $\mathsf{TethysDIP}$ to the dynamic setting is an interesting open problem.

Finally, some aspects of $\mathsf{TethysDIP}$ relate to graph orientability. Graph orientability studies how the edges of an undirected graph may be oriented in order to achieve certain properties, typically related either to the in- or outdegree sequence of the resulting graph, or to k-connectivity. This is relevant to our $\mathsf{TethysDIP}$ algorithm, insofar as its analysis is best formulated as a problem of deleting the minimum number of edges in a certain graph, so that every vertex has outdegree less than a given capacity p (cf. Section 4). As such, it relates to deletion orientability problems, such as have been studied in [HKL+18]. Many variants of this problem are NP-hard, such as minimizing the number of *vertices* that must be deleted to achieve the same property, and most of the literature is devoted to a more fine-grained classification of their complexity. In that respect, it seems we are "lucky" that our particular optimization target (minimizing the so-called overflow of the graph) can be achieved in only quadratic time. We did not find mention of this fact in the orientability literature.

Organization of the paper. Section 2 provides the necessary background and notation, and introduces definitions of storage and page efficiency. Section 3 introduces the notion of data-independent packing (DIP), and presents a generic construction of SSE from a DIP scheme. Section 4 gives an efficient construction of DIP. Section 5 concludes with practical experiments.

2 Background

2.1 Notation

Let $\lambda \in \mathbb{N}$ be the security parameter. For a distribution probability X, we denote by $x \leftarrow X$ the process of sampling a value x from the distribution. If $\mathcal{X}$ is a set, $x \leftarrow \mathcal{X}$ denotes the process of sampling x uniformly at random from $\mathcal{X}$. Logarithm in base 2 is denoted by $\log(\cdot)$. A function $f(\lambda)$ is *negligible* in λ if it is $\mathcal{O}(\lambda^{-c})$ for every $c \in \mathbb{N}$. If so, we write $f = \mathsf{negl}(\lambda)$.

Let $\mathsf{W} = \{w_1, \ldots, w_k\}$ be the set of keywords, where each keyword w_i is represented by a machine word, each of $\mathcal{O}(\lambda)$ bits, in the unit-cost RAM model, as in [ANSS16]. The plaintext database is regarded as an inverted index. To each keyword w_i is associated a list $\mathsf{DB}(w_i) = (\mathsf{ind}_1, \ldots, \mathsf{ind}_{\ell_i})$ of document identifiers matching the keyword, each of length $\mathcal{O}(\lambda)$ bits. The plaintext database is $\mathsf{DB} = (\mathsf{DB}(w_1), \ldots, \mathsf{DB}(w_k))$. Uppercase N denotes the total number of keyword-document pairs in DB, $N = |\mathsf{DB}| = \sum_{i=1}^{k} \ell_i$, as is usual in SSE literature.

We now introduce multi-maps. A *multi-map* M consists of k pairs $\{(K_i, \mathsf{vals}_i) : 1 \leq i \leq k\}$, where $\mathsf{vals}_i = (e_{i,1}, \ldots, e_{i,\ell_i})$ consists of ℓ_i values $e_{i,j}$. (Note that in this context, a *key* is an identification key in a key-value store, and not a cryptographic key.) We assume without loss of generality that the keys K_i are distinct. Throughout, we denote by n the total number of values $n = |\mathsf{M}| := \sum_{i=1}^{k} \ell_i$, following the convention of allocation and hashing literature. For the basic $\mathsf{TethysDIP}$ scheme, $n = N$. We assume (without loss of generality) that values $e_{i,j}$ can be mapped back unambiguously to the key of origin K_i. This will be necessary for our SSE framework, and can be guaranteed by assuming the values contain the associated key. As this comes with additional storage overhead, we discuss some encoding variants in the full version (some of these encodings result in $n > N$).

Throughout the article, the page size p is treated as a variable, independent of the dataset size n. Upper bounds of the form $f(n,p) = \mathcal{O}(g(n,p))$, where the function f under consideration depends on both n and p, mean that there exist constants C, C_n, C_p such that $f(n,p) \leq Cg(n,p)$ for all $n \geq C_n$, $p \geq C_p$.

2.2 Searchable Symmetric Encryption

At setup, the client generates an encrypted database EDB from the plaintext database DB and a secret key K. The client sends EDB to the server. To issue a search query for keyword w, the client sends a search token τ_w. The server uses the token τ_w and the encrypted database EDB to compute $\mathsf{DB}(w)$. In some cases, the server does not recover $\mathsf{DB}(w)$ directly; instead, the server recovers some data d and sends it to the client. The client then recovers $\mathsf{DB}(w)$ from d.

Formally, a static Searchable Symmetric Encryption (SSE) scheme is a tuple of algorithms $(\mathsf{KeyGen}, \mathsf{Setup}, \mathsf{TokenGen}, \mathsf{Search}, \mathsf{Recover})$.

- $K \leftarrow \mathsf{KeyGen}(1^\lambda)$: the key generation algorithm KeyGen takes as input the security parameter λ in unitary notation and outputs the master key K.

- $\mathsf{EDB} \leftarrow \mathsf{Setup}(K, \mathsf{DB})$: the setup algorithm takes as input the master key K and a database DB, and outputs an encrypted database EDB.
- $(\tau, \rho) \leftarrow \mathsf{TokenGen}(K, w)$: the token generation algorithm takes as input the master key K and a keyword w, and outputs a search token τ (to be sent to the server), and potentially some auxiliary information ρ (to be used by the recovery algorithm).
- $d \leftarrow \mathsf{Search}(\mathsf{EDB}, \tau)$: The search algorithm takes as input the token τ and the encrypted database EDB and outputs some data d.
- $s \leftarrow \mathsf{Recover}(\rho, d)$: the recovery algorithm takes as input the output d of the Search algorithm, and potentially some auxiliary information ρ, and outputs the set $\mathsf{DB}(w)$ of document identifiers matching the queried keyword w.

The $\mathsf{Recover}$ algorithm is used by the client to decrypt the results sent by the server. In many SSE schemes, the server sends the result in plaintext, and $\mathsf{Recover}$ is a trivial algorithm that outputs $s = d$.

Security Definition. We use the standard semantic security notion for SSE. A formal definition is given in [CGKO06]. Security is parametrized by a *leakage function* $\mathcal{L}$, composed of the setup leakage $\mathcal{L}_{\mathsf{Setup}}$, and the search leakage $\mathcal{L}_{\mathsf{Search}}$. Define two games, SSEReal and SSEIdeal. At setup, the adversary sends a database DB. In SSEReal, the setup is run normally; in SSEIdeal, the setup is run by calling a simulator on input $\mathcal{L}_{\mathsf{Setup}}(\mathsf{DB})$. The adversary can then adaptively issue search queries for keywords w that are answered honestly in SSEReal, and simulated by a simulator on input $\mathcal{L}_{\mathsf{Search}}(\mathsf{DB}, w)$ in SSEIdeal. The adversary wins if it correctly guesses which game it was playing.

Definition 2.1 (Simulation-Based Security). *Let Π be an SSE scheme, let $\mathcal{L}$ be a leakage function. We say that Π is $\mathcal{L}$-adaptively semantically secure if for all PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that*

$$|\Pr[\mathrm{SSEReal}_{\Pi,\mathcal{A}}(\lambda) = 1] - \Pr[\mathrm{SSEIdeal}_{\Pi,\mathcal{S},\mathcal{L},\mathcal{A}}(\lambda) = 1]| = \mathsf{negl}(\lambda).$$

2.3 Locality and Page Efficiency

The notions of locality and read efficiency were introduced by Cash and Tessaro [CT14]. We recall them, followed by our new metrics of page cost and page efficiency. We start with the definition of the *read pattern*. In the following definitions, the quantities EDB, τ are assumed to be computed according to the underlying SSE scheme, *i.e.* given a query for keyword w on the database DB, set $K \leftarrow \mathsf{KeyGen}(1^\lambda)$, $\mathsf{EDB} \leftarrow \mathsf{EDBSetup}(\mathsf{K}, \mathsf{DB})$, $\tau \leftarrow \mathsf{TokenGen}(K, w)$.

Definition 2.2 (Read Pattern). *Regard server-side storage as an array of memory locations, containing the encrypted database* EDB. *When processing the search query* $\mathsf{Search}(\mathsf{EDB}, \tau)$ *for keyword w, the server accesses memory locations $m_1, \ldots, m_h$. We call these locations the* read pattern *and denote it with* $\mathsf{RdPat}(\tau, \mathsf{EDB})$.

Definition 2.3 (Locality). *An SSE scheme has locality L if for any λ, DB, and keyword w, $\mathsf{RdPat}(\tau, \mathsf{EDB})$ consists of at most L disjoint intervals.*

Definition 2.4 (Read Efficiency). *An SSE scheme has read efficiency R if for any λ, DB, and keyword w, $|\mathsf{RdPat}(\tau, \mathsf{EDB})| \leq R \cdot P$, where P is the number of memory locations needed to store document indices matching keyword w in plaintext (by concatenating indices).*

Definition 2.5 (Storage Efficiency). *An SSE scheme has storage efficiency S if for any λ, DB, $|\mathsf{EDB}| \leq S \cdot |DB|$.*

Optimizing an SSE scheme for locality requires that each read query accesses few non-contiguous memory locations, thus making this operation efficient for HDDs. In the case of SSDs, it is sufficient to optimize for few page accesses (as SSDs efficiently read entire pages of memory). For this reason, we introduce the notions *page cost* and *page efficiency* to measure the efficiency of read queries performed on SSDs. More background is provided in the full version, together with experiments showing that page efficiency is an excellent predictor of SSD read performance (this is also supported by the experiments of Section 5).

Definition 2.6 (Page Pattern). *If server-side storage is regarded as an array of* pages*, when searching for a keyword w, the read pattern $\mathsf{RdPat}(\tau, \mathsf{EDB})$ induces a number of page accesses $p_1, \ldots, p_{h'}$. We call these pages the* page pattern*, denoted by $\mathsf{PgPat}(\tau, \mathsf{EDB})$.*

Definition 2.7 (Page Cost). *An SSE scheme has page cost $aX + b$, where a, b are real numbers, and X is a fixed symbol, if for any λ, DB, and keyword w, $|\mathsf{PgPat}(\tau, \mathsf{EDB})| \leq aX + b$, where X is the number of pages needed to store documents indices matching keyword w in plaintext.*

Definition 2.8 (Page Efficiency). *An SSE scheme has page efficiency P if for any λ, DB, and keyword w, $|\mathsf{PgPat}(\tau, \mathsf{EDB})| \leq P \cdot X$, where X is the number of pages needed to store documents indices matching keyword w in plaintext.*

A scheme with page cost $aX + b$ has page efficiency at most $a + b$. Compared to page efficiency, page cost is a more fine-grained measure that can be helpful when comparing the performance of different SSE schemes. It is clear that page efficiency is a direct counterpart of read efficiency, viewed at the page level, but it is also related to locality: a scheme with good locality and read efficiency immediately yields a scheme with good page efficiency, as formalized in the following theorem.

Theorem 2.1. *Any SSE scheme with read efficiency R and locality L has page efficiency at most $R + 2L$.*

The impossibility result of Cash and Tessaro [CT14] states (with some additional assumptions) that no SSE scheme can have simultaneously storage efficiency, read efficiency and locality $\mathcal{O}(1)$. As a consequence, no scheme with storage efficiency $\mathcal{O}(1)$ can have $R + 2L = \mathcal{O}(1)$. Nevertheless, our Tethys scheme has storage efficiency $\mathcal{O}(1)$ and page efficiency $\mathcal{O}(1)$. This shows that Theorem 2.1, while attractive in terms of genericity and simplicity, is not the best way to build a page-efficient scheme. In the full version, we show that the upper bound from Theorem 2.1 is tight.

Proof of Theorem 2.1. View server-side storage as an array of pages, without modifying the behavior of the scheme in any way. To process keyword w, the scheme makes at most L contiguous memory accesses of lengths $a_1, \ldots, a_L$. We have $\sum a_i \leq Rx$, where x denotes the amount of memory needed to store the plaintext answer (concatenation of document indices matching the query). Each memory access of length a_i covers at most $a_i/p + 2$ pages, where the two extra page accesses account for the fact that the start and end points of the access may not be aligned with server pages. Thus, the number of pages read is at most $\sum(a_i/p+2) \leq Rx/p + 2L$. It remains to observe that the number of pages needed to store the plaintext answer is at least x/p. Hence, the scheme has page cost (at most) $RX + 2L$, and page efficiency $R + 2L$. $\square$

3 SSE from Data-Independent Packing

In this section, we define data-independent packing, and based on this notion, provide a framework to construct SSE schemes. In Section 4, we will instantiate the framework with an efficient data-independent packing scheme.

3.1 Data-Independent Packing

A data-independent packing (DIP) scheme takes as input an integer m (the number of buckets), and a multi-map M (mapping keys to lists of values). Informally, it will assign the values of the multi-map into m buckets, each containing up to p values, and a stash. It provides a search functionality Lookup that, for a given key, returns the indices of buckets where the associated values are stored. In this section, p denotes the size of a bucket. To ease notation, it is implicitly a parameter of all methods. (In the concrete application to page-efficient SSE, p is the size of a page.)

Definition 3.1 (Data-Independent Packing).
A DIP *scheme is a triplet of algorithms* (Size, Build, Lookup)*:*

- $m \leftarrow \mathsf{Size}(n)$*: Takes as input a number of values n. Returns a number of buckets m.*
- $(B, S) \leftarrow \mathsf{Build}(\mathsf{M})$*: Takes as input a multi-map $\mathsf{M} = \{(K_i, (e_{i,1}, \ldots, e_{i,\ell_i})) : 1 \leq i \leq k\}$. Letting $n = |\mathsf{M}| = \sum_{1 \leq i \leq k} \ell_i$ and $m \leftarrow \mathsf{Size}(n)$, returns a pair (B, S), where B is an m-tuple of buckets $(B[1], \ldots, B[m])$, where each bucket $B[i]$ is a set of at most p multi-map values; and the stash S is another set of multi-map values.*
- $\mathcal{I} \leftarrow \mathsf{Lookup}(m, K, \ell)$*: Takes as input the total number of buckets m, a multi-map key K, and a number of items ℓ. Returns a set of bucket indices $I \subseteq [1, m]$.*

Correctness asks that all multi-map values $(e_{i,1}, \ldots, e_{i,\ell_i})$ associated with key K_i are either in the buckets whose indices are returned by $\mathsf{Lookup}(m, K_i, \ell_i)$, or in the stash. Later on, we will sometimes only ask that correctness holds with overwhelming probability over the random coins of Build.

Definition 3.2 (Correctness). *A* DIP *scheme is correct if for all multi-map* $\mathsf{M} = \{(K_i, (e_{i,1}, \ldots, e_{i,\ell_i})) : 1 \leq i \leq k\}$*, the following holds. Letting* $m \leftarrow \mathsf{Size}(|\mathsf{M}|)$*, and* $(B, S) \leftarrow \mathsf{Build}(\mathsf{M})$*:*

$$\forall i \in [1, k] : \mathsf{M}(K_i) \subseteq S \cup \bigcup_{j \in \mathsf{Lookup}(m, K_i, \ell_i)} B[j].$$

Intuitively, the definition of DIP inherently enforces data independence, in two ways. The first is that the number of buckets $m \leftarrow \mathsf{Size}(n)$ used for storage is solely a function of the number of values n in the multi-map. The second is that Lookup only depends on the queried key, and the number of values associated with that key. Thus, neither Size nor Lookup depend on the multi-map at the input of Build, other than the number of values it contains. It is in that sense that we say those two functions are *data-independent*: they do not depend on the dataset M stored in the buckets, including the sizes of the lists it contains. Looking ahead, when we use a DIP scheme, we will pad all buckets to their maximum size p, and encrypt them, so that the output of Build will also leak nothing more than the number of buckets m.

We supply Lookup with the number of values ℓ associated to the queried key. This is for convenience. If the number of values of the queried key was not supplied as input, it would have to be stored by the DIP scheme. We have found it more convenient to allow that information to be stored in a separate structure in future constructions. Not forcing the DIP scheme to store length information also better isolates the main combinatorial problem a DIP scheme is trying to capture, namely how to compactly store objects of variable size, while being data-independent. How to encode sizes introduces its own separate set of considerations.

Efficiency Measures. Looking ahead to the SSE construction, a bucket will be stored in a single page, and contain some document identifiers of the database. The goal is to keep the total number of buckets m small (quantified by the notion *storage efficiency*), and to ensure that Lookup returns small sets (quantified by the notion *lookup efficiency*). Intuitively, those goals will imply good storage efficiency (with a total storage of m pages, plus some auxiliary data), and good page efficiency (reading from the database requires few page accesses) for the resulting SSE scheme. Finally, the stash will be stored on the client side. Thus, the stash size should be kept small. These efficiency measures are formally defined in the following.

Definition 3.3 (Lookup Efficiency). *A* DIP *scheme has lookup efficiency* L *if for any multi-map* M*, any* $(m, B, S) \leftarrow \mathsf{Build}(\mathsf{M})$ *and any key* K *for which the values* $\mathsf{M}(K)$ *require a minimal number of buckets* x*, we have* $|\mathsf{Lookup}(m, K, \ell)| \leq L \cdot x$*.*

Definition 3.4 (Storage Efficiency). *A* DIP *scheme has storage efficiency E if for any multi-map* M *and any* $(m, B, S) \leftarrow \mathsf{Build}(\mathsf{M})$*, it holds that* $m \leq E \cdot (n/p)$*.*

Definition 3.5 (Stash size). *A* DIP *scheme has stash size C if for any multi-map* M *and any* $(m, B, S) \leftarrow \mathsf{Build}(\mathsf{M})$*, it holds that the stash contains at most C values.*

It is trivial to build a DIP scheme that disregards one of these properties. For example for good lookup and storage efficiency, we can store all values in the stash. For good storage efficiency and small stash size, it suffices to store all values in $m = \lceil n/p \rceil$ buckets and return all bucket indices $\{1, \cdots, m\}$ in Lookup. Lastly, for good lookup efficiency and stash size, we can pad every list to a multiple of p in size and subsequently split each list into chunks of size p. Each chunk can be stored in a bucket fixed by a hash function. But this scheme has a storage efficiency of p (this last approach is discussed in more detail in Section 1.2).

Ensuring good performance with respect to all properties at the same time turns out to be a hard problem. We refer to Section 4 for a concrete construction.

SSE *from Data-Independent Packing.* In this section, we give a framework to build an SSE scheme SSE(D) generically from a DIP scheme D with a bucket size p equal to the page size.

We now describe the construction in detail. Let PRF be a secure pseudo-random function mapping to $\{0, 1\}^{2\lambda + \lceil \log(N) \rceil}$. Let Enc be an IND-CPA secure symmetric encryption scheme (assimilated with its encryption algorithm in the notation). We split the output of the PRF into a key of 2λ bits and a mask of $\lceil \log(N) \rceil$ bits. Pseudo-code is provided in Algorithm 1.

Setup. The Setup algorithm takes as input a database DB, and the client's master secret key $K = (K_{\mathsf{PRF}}, K_{\mathsf{Enc}})$. For each keyword w_i, we have a list $\mathsf{DB}(w_i)$ of ℓ_i indices corresponding to the documents that match w_i. First, setup samples $(K_i, m_i) \leftarrow \mathsf{PRF}_{K_{\mathsf{PRF}}}(w_i)$ which will serve as token for w_i later on. To each list is associated the key K_i and the DIP scheme D is then called on the key-list pairs. Recall that D assigns the values to m buckets and a stash. Once that is done, each bucket is padded with dummy values until it contains exactly p values. Then, a table T with N entries is created which stores the length of each list in an encrypted manner. Concretely, T maps K_i to $\ell_i \oplus m_i$ and is filled with random elements until it contains N entries. Note that ℓ_i is encrypted with mask m_i and can be decrypted given m_i. The padded buckets are then encrypted using Enc with key K_{Enc}, and sent to the server in conjunction with the table T. The stash is stored on the client side.

Search. To retrieve all documents matching keyword w_i, the client generates the access token $(K_i, m_i) \leftarrow \mathsf{PRF}_{K_{\mathsf{PRF}}}(w_i)$ and forwards it to the server. The server retrieves $\ell_i \leftarrow T[K_i] \oplus m_i$ and queries D to retrieve the indices $I \leftarrow \mathsf{Lookup}(K_i, \ell_i)$ of the encrypted buckets. The server sends the respective buckets

Algorithm 1 $\mathsf{SSE}(\mathsf{D})$

```
KeyGen($1^\lambda$)
 1: Sample keys $K_{\mathsf{PRF}}$, $K_{\mathsf{Enc}}$ for PRF, Enc with security parameter $\lambda$
 2: return $K = (K_{\mathsf{PRF}}, K_{\mathsf{Enc}})$
Setup($K$, DB)
 1: Initialize empty set M, empty table $T$
 2: $N \leftarrow |\mathsf{DB}|$
 3: for all keywords $w_i$ do
 4:     $(K_i, m_i) \leftarrow \mathsf{PRF}_{K_{\mathsf{PRF}}}(w_i)$
 5:     $\ell_i \leftarrow |\mathsf{DB}(w_i)|$
 6:     $T[K_i] \leftarrow \ell_i \oplus m_i$
 7: $\mathsf{M} \leftarrow \{K_i, \mathsf{DB}(w_i) : 1 \leq i \leq k\}$
 8: $m, B, S \leftarrow \mathsf{Build}(\mathsf{M})$
 9: Fill $T$ up to size $N$ with random values
10: Store the stash $S$ on the client
11: return $\mathsf{EDB} = (\mathsf{Enc}_{K_{\mathsf{Enc}}}(B[1]), \ldots, \mathsf{Enc}_{K_{\mathsf{Enc}}}(B[m]), T)$
TokenGen($K$, $w_i$)
 1: $(K_i, m_i) \leftarrow \mathsf{PRF}_{K_{\mathsf{PRF}}}(w_i)$
 2: return $\tau_i = (K_i, m_i)$
Search(EDB, $\tau_i$)
 1: Initialize empty set $R$
 2: Parse $\tau_i$ as $(K_i, m_i)$
 3: Set $\ell_i = T[K_i] \oplus m_i$
 4: $I \leftarrow \mathsf{Lookup}(m, K_i, \ell_i)$
 5: for all $j \in I$ do
 6:     Add encrypted buckets $B[j]$ to $R$
 7: return $R$
```

back to the client, who decrypts them to recover the list elements. Finally, the client checks its own stash for any additional elements matching w_i.

Efficiency. The efficiency of $\mathsf{SSE}(\mathsf{D})$ heavily relies on the efficiency of D. The server stores the encrypted database EDB consisting of a table of size $N = |\mathsf{DB}|$ and m buckets. The concrete value of m depends on the storage efficiency S of D. By definition, the scheme $\mathsf{SSE}(\mathsf{D})$ has storage efficiency $S + 1$. During the search process, $\mathsf{SSE}(\mathsf{D})$ accesses one entry of table T and $|I|$ buckets, where I is the set of indices returned by Lookup. As each bucket is stored in a single page, a bucket access requires a single page access. The access to T requires an additional page access. In total, the page efficiency of $\mathsf{SSE}(\mathsf{D})$ is $L+1$, where L is the lookup efficiency of D. Note that we assume that Lookup does not make any additional page accesses, as is guaranteed by our construction. Lastly, the client stores the key K and the stash S locally. Thus, the client storage is $C + \mathcal{O}(1)$, where C is the stash size of D.

Security. The leakage profile of the construction is the standard leakage profile of a static SSE scheme. Recall that x_i is the minimal number of pages for the list of documents matching keyword w_i. The leakage during setup is $\mathcal{L}_{\mathsf{Setup}}(\mathsf{DB}) = |\mathsf{DB}| = N$. The leakage during search is $\mathcal{L}_{\mathsf{Search}}(\mathsf{DB}, w_i) = (\ell_i, \mathsf{sp})$, where sp is the *search pattern*, that is, the indices of previous searches for the same keyword (a formal definition is given in [CGKO06]). Let $\mathcal{L} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Search}})$.

Theorem 3.1 (SSE Security). *Let* D *be a* DIP *scheme with storage efficiency* S*, lookup efficiency* L*, and stash size* C*. Assume that* Lookup *does not make any page accesses,* Enc *is an IND-CPA secure encryption scheme and* PRF *is a secure pseudo-random function. Then* $\mathsf{SSE}(\mathsf{D})$ *is a* $\mathcal{L}$*-adaptively semantically secure SSE scheme with storage efficiency* $S+1$*, page efficiency* $L+1$*, and client storage* $C + \mathcal{O}(1)$*.*

The full proof is given in the full version. It is straightforward, and we sketch it here. For Setup, the simulator creates the required number m of buckets, derived from $N = \mathcal{L}_{\mathsf{Setup}}(\mathsf{DB})$, and fills each one with the encryption of arbitrary data using Enc. Similarly, it creates a table T mapping N random values κ to random entries χ. It then creates the simulated database EDB consisting of the buckets and the table. The IND-CPA security of Enc guarantees that the adversary cannot distinguish the simulated buckets from the real ones. Also, the simulated table is indistinguishable from the real table, since the concrete values ℓ_i are masked with a random mask m_i. Thus, the unqueried table entries appear random.

For a (new) search query, the simulator receives from the leakage function the number ℓ_i, and simulates the token $\tau_i = (K_i, \ell_i \oplus T[K_i])$ by choosing K_i uniformly from the unqueried keys κ of table T. The PRF security of PRF guarantees that the adversary cannot distinguish the simulated token from the real one. Note that the adversary recovers the correct value $\ell_i = T[K_i] \oplus (\ell_i \oplus T[K_i])$. This concludes the proof.

While the proof is simple, it relies heavily on the data independence of the DIP scheme. Namely, Lookup does not take the database as input, but only its size. As a consequence, the simulator need not simulate any of the Lookup inputs. Another subtle but important point is that the security argument requires that the correctness of the DIP scheme holds with overwhelming probability over the random coins of Build. Indeed, the probability of a correctness failure may be dependent on the dataset at the input of Build, and thus leak information. Moreover, if a correctness failure occurs, it is not acceptable to run Build again with fresh random coins, as the random coins of Build would then become dependent on the dataset. The same subtlety exists in the proofs of some Oblivious RAM constructions, and has led to flawed proofs when overlooked, as well as concrete distinguishing attacks exploiting this flaw [GM11, Appendix D], [FNO20].

4 Efficient Data-Independent Packing

In this section, we introduce an efficient DIP scheme. As a reminder, a DIP scheme allocates the values of a multi-map into m buckets or a stash. Recall that a multi-map consists of k keys K_i, where each key K_i maps to ℓ_i values

$(e_{i,1}, \dots, e_{i,\ell_i})$. At first, we restrict ourselves to at most p (one page) values per key for simplicity, *i.e.* $\ell_i \leq p$. The restriction will be removed at the end of the section.

The construction is parametrized by two hash functions H_1, H_2, mapping into the buckets, *i.e.* mapping into $\{1, \dots, m\}$. H_1 is uniformly random among functions mapping into $\{1, \dots, m/2\}$, and H_2 is uniformly random among functions mapping into $\{m/2+1, \dots, m\}$. (The distribution of H_1 and H_2, and the fact they have disjoint ranges, is not important for the description of the algorithm; it will only become relevant when bounding the stash size in Theorem 4.3.)

To the i-th key K_i are associated two possible destination buckets for its values, $H_1(K_i)$ and $H_2(K_i)$. Not all values need to be allocated to the same bucket, *i.e.* some values can be allocated to bucket $H_1(K_i)$, and other values to bucket $H_2(K_i)$. If both destination buckets are already full, some values may also be stored in the stash. In the end, for each key K_i, some a values are allocated to bucket $H_1(K_i)$, b values to bucket $H_2(K_i)$, and c values to the stash, with $a + b + c = \ell_i$.

The goal of the TethysDIP algorithm is to determine, for each key, how many values are assigned to each bucket, and how many to the stash, so that no bucket receives more than p values in total, and the stash is as small as possible. We shall see that the algorithm is optimal, in the sense that it minimizes the stash size subject to the previous constraint.

Algorithm description. Pseudo-code is provided in Algorithm 2. The algorithm takes as input the number of buckets m, and the multi-map $\mathsf{M} = \{(K_i, (e_{i,1}, \dots, e_{i,\ell_i})) : 1 \leq i \leq k\}$. It outputs a dictionary B such that $B[i]$ contains the values $e_{i,j}$ that are stored in bucket number i, for $i \in \{1, \dots, m\}$, together with a stash S.

The algorithm first creates a graph similar to the cuckoo graph in cuckoo hashing: vertices are the buckets, and for each value $e_{i,j}$, an edge is drawn between its two possible destination buckets $H_1(K_i)$ and $H_2(K_i)$. Note that there may be multiple edges between any two given vertices. Edges are initially oriented in an arbitrary way. Ultimately, each value will be assigned to the bucket at the *origin* of its corresponding edge. This means that the load of a bucket is the outdegree of the associated vertex.

Intuitively, observe that if we have a directed path in the graph, and we flip all edges along this path, then the load of intermediate nodes along the path is unchanged. Meanwhile, the load of the bucket at the origin of the path is decreased by one, and the load of the bucket at the end of the path is increased by one. Hence, in order to decrease the number of values sent to the stash, we want to find as many disjoint paths as possible going from overfull buckets to underfull buckets, and flip all edges along these paths. To find a maximal set of such paths, TethysDIP runs a max flow algorithm (see full version for more details). Then all edges along the paths are flipped. Finally, each value is assigned to the bucket at the origin of its associated edge. If a bucket receives more than p values, excess values are sent to the stash.

Algorithm 2 TethysDIP

```
Build($m, \mathsf{M} = \{(K_i, (e_{i,1}, \ldots, e_{i,\ell_i})) : 1 \leq i \leq k\}$)
1: $B \leftarrow m$ empty buckets, $S \leftarrow$ empty stash
2: Create an oriented graph $G$ with $m$ vertices numbered $\{1, \ldots, m\}$
3: for all values $e_{i,j}$ do
4:    Create an oriented edge $(H_1(K_i), H_2(K_i))$ with label $e_{i,j}$
5: Add separate source vertex $s$ and sink vertex $t$
6: for all vertex $v$ do
7:    Compute its outdegree $d$.
8:    if $d > p$ then
9:       Add $d - p$ edges from the source $s$ to $v$
10:   else if $d < p$ then
11:      Add $p - d$ edges from $v$ to the sink $t$
12: Compute a max flow from $s$ to $t$
13: Flip every edge that carries flow
14: for all vertex $v \in \{1, \ldots, m\}$ do
15:    $B[v] \leftarrow \{e_{i,j}$ : origin of edge $e_{i,j}$ is $v\}$
16: for all vertex $v \in \{1, \ldots, m\}$ do
17:    if $|B[v]| > p$ then
18:       $|B[v]| - p$ values are moved from $B[v]$ to $S$
19: return $(B, S)$

Lookup($m, K, \ell \leq p$)
1: returns $\{H_1(K), H_2(K)\}$
```

Efficiency. We now analyze the efficiency of TethysDIP. Note that each key still maps to at most p values for now. In order to store a given multi-map M, TethysDIP allocates a total number of $m = (2 + \varepsilon)n/p$ buckets. Thus, it has storage efficiency $2 + \varepsilon = \mathcal{O}(1)$. For accessing the values associated to key K, TethysDIP returns the result of the evaluation of the two hash functions at point K. Hence, TethysDIP has lookup efficiency $2 = \mathcal{O}(1)$. The analysis of the stash size is much more involved. In Section 4.1, we show that a stash size $p \cdot \omega(\log \lambda)/\log n$ suffices. In particular, the stash size does not grow with the size of the multi-map M.

Handling Lists of Arbitrary Size. The previous description of the algorithm assumes that all lists in the multi-map M are at most one page long, *i.e.* $\ell_i \leq p$ for all i. We now remove that restriction. To do so, we are going to preprocess the multi-map M into a new multi-map M' that only contains lists of size at most p.

In more detail, for each key-values pair $(K_i, (e_{i,1}, \ldots, e_{i,\ell_i}))$, we split $(e_{i,1}, \ldots, e_{i,\ell_i})$ into $x_i = \lfloor \ell_i/p \rfloor$ *sublists* $(e_{i,1}, \ldots, e_{i,p})$, …, $(e_{i,p(x_i-1)+1}, \ldots, e_{i,px_i})$ of size p, plus one sublist of size at most p containing the remaining values $(e_{i,px_i+1}, \ldots, e_{i,\ell_i})$. We associate the j-th sublist to a new key $K_i \| j$ (without loss of generality, assume there is no collision with a previous key). The new multi-map M' consists of all sublists generated in this way, with the j-th sublist of key K_i associated to key $K_i \| j$.

The $\mathsf{TethysDIP}$ algorithm is then applied to the multi-map M', which only contains lists of size at most p. In order to retrieve the values associated to key K_i in the original multi-map M, it suffices to query the buckets $H_1(K_i \parallel j)$, $H_2(K_i \parallel j)$ for $j \leq \lceil \ell_i/p \rceil$. Correctness follows trivially. (This approach can be naturally generalized to transform a DIP scheme for lists of size at most p into a general DIP scheme.)

Note that the total number of values in M' is equal to the total number of values n in M, as we only split the lists into sublists. Hence the scheme retains storage efficiency $2 + \varepsilon$ and stash size $p \cdot \omega(\log \lambda)/\log n$. Similarly, for a list of size ℓ, we require at minimum $x = \lceil \ell/p \rceil$ buckets. As we return x evaluations of each hash function, the storage efficiency remains 2.

The Tethys SSE scheme. We can instantiate the framework given in Section 3.1 with $\mathsf{TethysDIP}$. This yields a SSE scheme $\mathsf{Tethys} := \mathsf{SSE}(\mathsf{TethysDIP})$. As the $\mathsf{TethysDIP}$ has constant storage and lookup efficiency, Tethys also has constant storage and page efficiency and the same stash size. This is formalized in the following theorem. Let $\mathcal{L}_{\mathsf{Setup}}(\mathsf{DB}) = |\mathsf{DB}|$ and $\mathcal{L}_{\mathsf{Search}}(\mathsf{DB}, w_i) = (\ell_i, \mathsf{sp})$, where sp is the search pattern. Let $\mathcal{L}(\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Search}})$.

Theorem 4.1. *Assume that* Enc *is an IND-CPA secure encryption scheme,* PRF *is a secure pseudo-random function, and* H_1*,* H_2 *are random oracles. Then* Tethys *is an* $\mathcal{L}$*-adaptively semantically secure SSE scheme with storage efficiency* $\mathcal{O}(1)$*, page efficiency* $\mathcal{O}(1)$*, and client storage* $\mathcal{O}(p \cdot \omega(\log \lambda)/\log n)$*.*

The $\mathsf{TethysDIP}$ scheme inside Tethys requires two hash functions H_1 and H_2. The stash size bound analysis assumes those two functions are uniformly random. In practice, standard hash functions can be used. Formally, to avoid an unnecessary use of the Random Oracle Model, the hash functions can be realized by a PRF, with the client drawing the PRF key and sending it to the server together with the encrypted dataset. By standard arguments, the correctness of $\mathsf{TethysDIP}$ still holds with overwhelming probability, assuming the PRF is secure.

4.1 Stash Size Analysis

We now analyze the stash size of $\mathsf{TethysDIP}$. We proceed by first showing that the stash size achieved by $\mathsf{TethysDIP}$ is optimal, in the sense given below. We then prove a stash size bound that holds for any optimal algorithm.

Optimality. Given the two hash functions H_1 and H_2, and the multi-map M at the input of $\mathsf{TethysDIP}$, say that an assignment of the multi-map values to buckets is *valid* if every value associated to key K is assigned to one of its two destination buckets $H_1(K)$ or $H_2(K)$, or the stash, and no bucket receives more than p values. $\mathsf{TethysDIP}$ is optimal in the sense that the assignment it outputs achieves the minimum possible stash size among all valid assignments. In other words, $\mathsf{TethysDIP}$ optimally solves the optimization problem of minimizing the stash size, subject to the constraint that the assignment is valid. This holds true regardless of the choice of hash functions (which need not be random as far as this property is concerned), regardless of the number of buckets m, and regardless of the initial orientation of the graph before the max flow is computed. To formalize this, let us introduce some notation.

The problem solved by TethysDIP is naturally viewed as a graph orientability problem (see related work in Section 1). The input of the problem is the graph built in lines 2–4: vertices are buckets $V = \{1, \dots, m\}$, and each list i gives rise to ℓ_i edges from vertex $H_1(K_i)$ to $H_2(K_i)$. Recall that the outdegree $\mathsf{out}(v)$ of a vertex v is the load of the corresponding bucket. Define the *overflow* of the graph as the quantity $\sum_{v \in V} \max(0, \mathsf{out}(v) - p)$. Observe that this quantity is exactly the number of values that cannot fit into their assigned bucket, hence the number of values that are sent to the stash in line 18. The problem is to orient the edges of the graph so as to minimize that quantity. In the following theorem, TethysDIP is viewed as operating on graphs. Its input is the undirected graph G described just above, and its output is a directed graph D arising from G by orienting its edges according to Algorithm 2.

Theorem 4.2 (Optimality of TethysDIP). *Let G be an undirected graph. Let D be the directed graph output by* TethysDIP *on input G. Then* $\mathsf{overflow}(D)$ *is minimal among all directed graphs arising from G.*

The proof of Theorem 4.2 is given in the full version. In short, the proof uses the max-flow min-cut theorem to partition the vertices into two sets S (containing the source) and T (containing the sink), such that after flipping the direction of the flow in line 13, there is no edge going from S to T. Further, it is shown that all overflowing values are in S, and all buckets in S are at capacity or over capacity. Intuitively, the number of overflowing values cannot be decreased, because flipping edges within S can only increase the overflow, and there is no edge going from S to T. We refer to the full version for the full proof.

This shows that TethysDIP finds an optimal solution. Before continuing, we note that the max flow approach of TethysDIP was inspired by a result of Sanders *et al.* [SEK03], which uses a similar algorithm. The relationship between the algorithm by Sanders *et al.* and TethysDIP is worth discussing. The two algorithms have different optimization targets: the goal of the algorithm by Sanders *et al.* is not to minimize the overflow, but to minimize the max load (the load of the most loaded bucket). Another notable difference is that we allow for a stash, which allows us to reach a negligible probability of failure (the associated analysis is the most technically challenging part of this work). Nevertheless, if we disregard the stash, the algorithm from [SEK03] can be reinterpreted in the light of our own algorithm, as follows. Given an algorithm $\mathcal{A}$ that minimizes the overflow, one can build an algorithm $\mathcal{B}$ that minimizes the max load, using a logarithmic number of black-box calls to $\mathcal{A}$. Indeed, $\mathcal{A}$ yields an overflow of zero if and only if the capacity p of buckets is greater than or equal to the smallest attainable max load. Hence, it suffices to proceed by dichotomy until the smallest possible value of the max load is reached. Although it is not presented in this way in [SEK03], the algorithm by Sanders *et al.* can be reinterpreted as being built in that manner, with TethysDIP playing the role of algorithm $\mathcal{A}$. (As a side effect, our proof implies a new proof of Sanders *et al.*'s result.)

Stash size bound. The security of Tethys relies on the fact that memory accesses are data-independent. Data independence holds because the two buckets where a given list can be assigned are determined by the two hash functions, independently of the length distribution of other lists. In practice, we want to fix an upper bound on the size of the stash. If the bound were exceeded (so the construction fails), we cannot simply draw new random hash functions and start over. Indeed, from the perspective of the SSE security proof, this would amount to choosing a new underlying DIP scheme when some aspect of the first DIP scheme fails (namely, when the stash is too large). But the choice of DIP scheme would then become data-dependent, invalidating the security argument. It follows that we want to find a bound on the stash size that guarantees a negligible probability of failure in the cryptographic sense, and not simply a low probability of failure. We prove that this can be achieved using only $m = \mathcal{O}(n)$ buckets, and a stash size that does not grow with the size of the multi-map.

Theorem 4.3 (Stash size bound). *Let $\varepsilon > 0$ be an arbitrary constant, and let $p, n \geq p, m \geq (2+\varepsilon)n/p$, $s = n^{o(1)}$ be integers. Let L be an arbitrary vector of integers such that $\max L \leq p$ and $\sum L = n$.*

$$\Pr[\mathsf{Fail}_{m,p,s}(L,H)] = \mathcal{O}\left(p \cdot n^{-s/(2p)}\right).$$

In particular, a stash of $\omega(\log \lambda)/\log n$ pages suffices to ensure that TethysDIP *succeeds, except with negligible probability.*

In that statement, the vector L represents a multi-map with keys mapping to p or less values, H is the pair of hash functions (H_1, H_2), and s is the stash size. $\mathsf{Fail}_{m,p,s}(L,H)$ denotes the probability that it is impossible to orient the edges of the graph G discussed earlier in such a way that the overflow of the resulting orientation is less than s. By Theorem 4.2, as long as such an orientation exists, TethysDIP finds one, so $\mathsf{Fail}_{m,p,s}(L,H)$ is equal to the probability of failure of TethysDIP. The bottom line is that, under mild assumptions about the choice of parameters, a stash of $\omega(\log \lambda)/\log n$ pages suffices to ensure a negligible probability of failure. If $n \geq \lambda$, $\log \lambda$ pages suffice.

Note that the probability of failure *decreases* with n. This behavior is reflected in practical experiments, as shown in Section 5. The inverse dependency with n may seem counter-intuitive, but recall that the number of buckets $m > (2+\varepsilon)n/p$ increases with n. In practice, what matters is that the stash size can be upper-bounded independently of the size n of the database, since it does not increase with n. Ultimately, the stash will be stored on the client side, so this means that client storage does not scale with the size of the database.

The factor $2+\varepsilon$ for storage efficiency matches the cuckoo setting. Our problem includes cuckoo hashing as a special case, so this is optimal (see full version for more details). The constant ε can be arbitrarily small. However, having non-zero

ε has important implications for the structure of the cuckoo graph: there is a phase transition at $\varepsilon = 0$. For instance, if we set $\varepsilon = 0$, the probability that the cuckoo graph contains a component with multiple cycles (causing standard cuckoo hashing to fail) degrades from $\mathcal{O}(1/n)$ to $\sqrt{2/3} + o(1)$ [DK12]. Beyond cuckoo hashing, this phase transition is well-known in the theory of random graphs: asymptotically, if a random graph has m vertices and $n = cm$ edges for some constant c, its largest component has size $\log n$ when $c < 1/2$ a.s., whereas it blows up to $\Omega(n)$ as soon as $c > 1/2$ [Bol01, chapter 5]. This strongly suggests that a storage overhead factor of $2 + \varepsilon$ is inherent to the approach, and not an artifact of the proofs.

The proof of Theorem 4.3 is given in the full version. In a nutshell, the idea is to use a convexity argument to reduce to results on cuckoo hashing, although the details are intricate. We now provide a high-level overview. The first step is to prove that the expectancy of the stash size for an arbitrary distribution of list lengths is upper-bounded by its expectancy when all lists have length p (while n and m remain almost the same), up to a polynomial factor. The core of that step is a convexity argument: we prove that the minimal stash size, as a function of the underlying graph, is Schur-convex, with respect with the natural order on graphs induced by edge inclusion. The result then follows using some majorization techniques (inspired by the analysis of weighted balls-and-bins problems in [BFHM08]). In short, the first step shows that, for expectancy at least, the case where all lists have length p is in some sense a worst case (up to a polynomial factor). The second step is to show that in that worse case, the problem becomes equivalent to cuckoo hashing with a stash. The third and final step is to slightly extend the original convexity argument, and combine it with some particular features of the problem, to deduce a tail bound on the stash size, as desired. The final step of the proof reduces to stash size bounds for cuckoo hashing. For that purpose, we adapt a result by Wieder [Wie17].

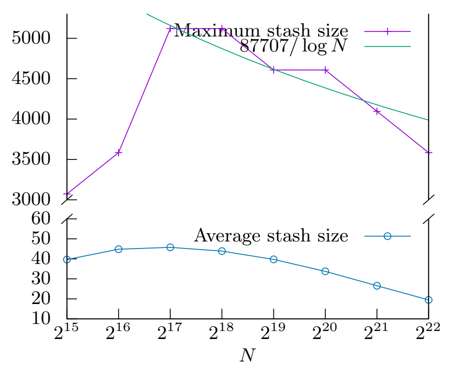
5 Experimental Evaluation

All evaluations and benchmarks have been carried out on a computer with an Intel Core i7 4790K 4.00 GHz CPU with 4 cores (8 logical threads), running Linux Debian 10.2. We used a 250 GiB Samsung 850 EVO SSD and a 4 TiB Seagate IronWolf Pro ST4000NE001 HDD, both connected with SATA, and formatted in `ext4`. The SSD page size is 4 KiB. The HDD was only used for the benchmarks (see full version), and we use the SSD for the following evaluation.

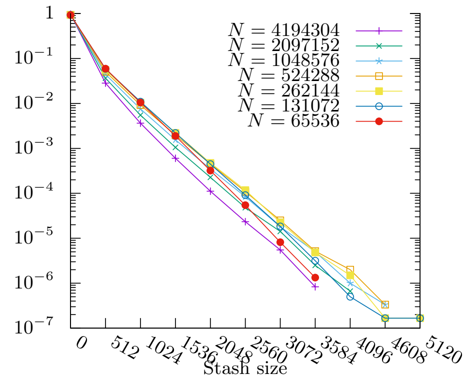
We chose the setting where document identifiers are encoded on 8 bytes and tags on 16 bytes. This allows us to support databases with up to 2^{64} documents and 2^{48} distinct keywords, with a probability of tag collision at most 2^{-32}. A page fits $p = 512$ entries.

5.1 Stash Size

Although the theory in Section 4.1 gives the asymptotic behavior of the size of the stash in TethysDIP, concrete parameters are not provided. We implemented



(a) Maximum and average stash size for fixed $\varepsilon = 0.1$.



(b) Experimental probability masses of the stash size for fixed $\varepsilon = 0.1$.

Fig. 1 – Experimental evaluation of the stash size made over 6×10^6 worst-case random TethysDIP allocations.

TethysDIP in Rust in order to run a large number of simulations, and evaluate the required stash size in practice. We want an evaluation of the stash size for page size p and an input multi-map with total value count N and bucket count m. A multi-map M_M that maps N/p keys to exactly p values is the worst-case for the stash size (see section 4.1). Thus, we evaluate the stash size of TethysDIP on the input M_M for given p, N, m.

In Figure 1a, we fix the parameter $\varepsilon = 0.1$ and look at the maximum size of the stash for various values of N. We can see that it fits a $C/\log N$ curve (except for low values of N, where the asymptotic behavior has not kicked in yet), as predicted by the theory. This confirms that the stash size does not increase (and in fact slightly decreases) with N, hence does not scale with the size of the database. In Figure 1b, for the same experiments, we plot the probability of having a stash of a given size. As was expected from Theorem 4.3, we can see that this probability drops exponentially fast with the size of the stash.

In the full version, we present data that clearly shows the transition phase at $\varepsilon = 0$, also predicted by the theory. The code of these experiments is publicly available [Bos21b].

5.2 Performance

We implemented Tethys in C++, using `libsodium` as the backend for cryptographic operations (HMAC-Blake2 for PRF and ChaCha20 for Enc), and using Linux' `libaio` library for storage accesses. Using `libaio` makes it possible to very efficiently parallelize IOs without having to rely on thread pools: although it does bring a few constraints in the way we access non-volatile storage, it allows for the performance to scale very cheaply, regardless of the host's CPU. As a consequence, our implementation uses only two threads: one for the submission

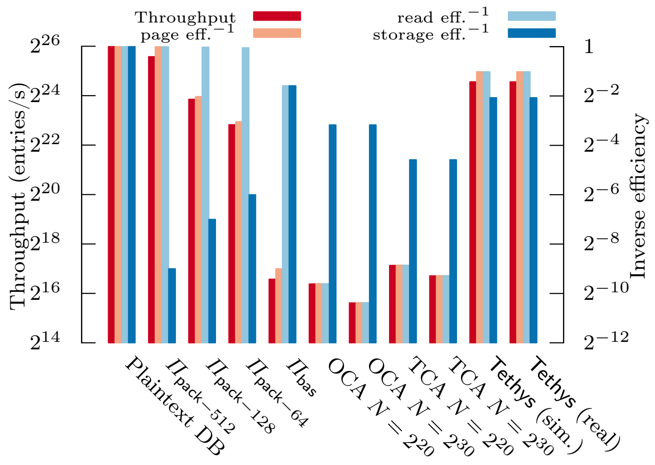


Fig. 2 – Throughput, inverse page efficiency, inverse read efficiency, and inverse storage efficiency for various SSE schemes, in log scale. Higher is better. $\Pi_{\mathsf{pack}-n}$ corresponds to Π_{pack} with n entries per block.

of the queries, and the other one to reap the completion queue, decrypt and decode the retrieved buckets.

At setup, the running time of TethysDIP is dominated by a max flow computation on a graph with n edges and $m = (1+\varepsilon)n/p$ vertices. We use a simple implementation of the Ford-Fulkerson algorithm [FF56], with running time $\mathcal{O}(nf)$, where $f \leq n$ is the max flow. This yields a worst-case bound $\mathcal{O}(n^2)$. Other max flow algorithms, such as [GR98] have running time $\widetilde{\mathcal{O}}\left(n^{3/2}\right)$; because this is a one-time precomputation, we did not optimize this step. We have experimented on the English Wikipedia database, containing about 140 million entries, and 4.6 million keywords. TethysDIP takes about 90 minutes to perform the full allocation. This is much slower than other SSE schemes, whose setup is practically instant. However, it is only a one-time precomputation. Using Pluto rather than Tethys makes a dramatic difference: most of the database ends up stored in HT (see full version), and TethysDIP completes the allocation in about 4 seconds.

Regarding online performance, comparing our implementation with available implementations of SSE schemes would be unfair: the comparison would be biased in our favor, because our implementation is optimized down to low-level IO considerations, whereas most available SSE implementations are not. To provide a fair comparison, for each SSE scheme given in the comparison, we analyzed its memory access pattern to deduce its IO workload. We then replayed that workload using the highly optimized `fio` Flexible I/O Tester (version 3.19) [Axb20]. While doing so, we have systematically advantaged the competition. For example, we have only taken into account the IO cost, not the additional cryptographic operations needed (which can be a significant overhead for some schemes, *e.g.* the One/Two Choice Allocation algorithms). Also, we have completely ignored the overhead induced by the storage data structure: for Π_{bas} and Π_{pack}, we assume a perfect dictionary, that only makes a single access per block of retrieved entries. Although this is technically possible, it would very costly, as it requires either a Minimal Perfect Hash Function, a very small load factor, or a kind of position map that is small enough to fit into RAM (that last option does not scale). Similarly, for the One-Choice Allocation (OCA) and Two-Choice Allocation (TCA) algorithms, we used the maximum read throughput achieved on our evaluation hardware, and assumed that this throughput was attained when reading consecutive buckets of respective size $\Theta(\log N)$ and $\Theta(\log \log N)$ required by the algorithms. In practice, we fixed the read efficiency of OCA to $3 \log N \log \log N$ and the one of TCA to $8 \log \log N (\log \log \log N)^2$, following [ANSS16]. The code is OpenSource and freely accessible [Bos21a].

We also computed the expected performance of Tethys using the same workload replay technique. The resulting performance measures are very close to our optimized full implementation (less than 0.1% difference on 2^{20} queries over 2^{19} distinct keywords). As that result illustrates, we argue that using simulated IO workloads to compare the performance of SSE schemes is quite accurate. The comparison between Tethys and other SSE schemes is given on Figure 2, including both the full implementation of Tethys, and its simulated workload.

We observe that Tethys compares very well with previous schemes. It vastly outperforms the One-Choice and Two-Choice allocation algorithms, as well as Π_{bas}, with over 170 times higher throughput. It also competes with all the Π_{pack} variants, its throughput being only exceeded by $\Pi_{\mathsf{pack}-512}$ with a twofold increase, due to the fact that Tethys needs to read two pages for every query. However, Π_{pack} incurs a huge storage cost in the worst case (up to a factor $p = 512$), leaving Tethys as the only scheme that performs well in both metrics. In addition, as explained earlier, our simulation of Π_{pack} does not account for the cost of the hash table implementation it relies on. For example, if we were to choose cuckoo hashing as the underlying hash table in Π_{pack}, the throughputs of $\Pi_{\mathsf{pack}-512}$ and of Tethys would be identical. The Π_{2lev} variant from [CJJ+14] is not included in the comparison, because its worst-case storage efficiency is the same as Π_{pack} (it handles short lists in the same way), and its throughput is slightly lower (due to indirections).

Our experiments show that Tethys is competitive even with insecure, plaintext databases, as the throughput only drops by a factor 2.63, while increasing the storage by a factor $4 + 2\varepsilon$ in the worst case (a database with lists of length 2 only, using the encoding $\mathsf{EncodeSeparate}$ from the full version). When sampling lists length uniformly at random between 1 and the page size, the storage efficiency is around 2.25 for $\varepsilon = 0.1$ and a database of size 2^{27}. For the encryption of Wikipedia (4.6 million keywords and 140 million entries), the storage efficiency is 3. (The extra cost beyond $2 + \varepsilon$ is mainly due to using the simpler, but suboptimal $\mathsf{EncodeSeparate}$ scheme from the full version.) In the full version, we further present the end-to-end latency of a search query on Tethys.

Finally, we have also plotted inverse read efficiency and inverse page efficiency for each scheme. As is apparent on Figure 2, inverse page efficiency correlates very strongly with throughput. When computing the correlation between the two across the various experiments in Figure 2, we get a correlation of 0.98, indicating a near-linear relationship. This further shows the accuracy of page efficiency as a predictor of performance on SSDs.

6 Conclusion

To conclude, we point out some problems for future work. First, like prior work on locality, Tethys only considers the most basic form of SSE: single-keyword queries, on a static database. A generalization to the dynamic setting opens up a number of interesting technical challenges. (A generic conversion from a static to a dynamic scheme may be found in [DP17], but would incur a logarithmic overhead in both storage efficiency and page efficiency.) A second limitation is that the initial setup of our main DIP algorithm, TethysDIP, has quadratic time complexity in the worst case. This is only a one-time precomputation, and practical performance is better than the worst-case bound would suggest, as shown in Section 5. Nevertheless, a more efficient algorithm would be welcome. Lastly, when querying a given keyword, Tethys returns entire pages of encrypted indices, some of which might not be associated to the keyword. Using an appropriate encoding, the matching keywords can be identified. While reducing volume leakage, this induces an overhead in communication, unlike other schemes such as Π_{bas} from [CJJ$^+$14], where only matching identifiers are returned. Due to the practical relevance of page efficiency, the intent of this work is that the notion will spur further research.

Acknowledgments

The authors thank Jessie Bertanier for his explanations on the inner workings of SSDs, which motivated our investigation. This work was supported by the ANR JCJC project SaFED and ANR Cyberschool ANR-18-EURE-0004.

References

ADW14. Aumüller, M., Dietzfelbinger, M., and Woelfel, P. Explicit and efficient hash families suffice for cuckoo hashing with a stash. Algorithmica, vol. 70(3):(2014), pp. 428–456.

ANSS16. Asharov, G., Naor, M., Segev, G., and Shahaf, I. Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In: D. Wichs and Y. Mansour (eds.), 48th ACM STOC, pp. 1101–1114. ACM Press (Jun. 2016).

ASS18. Asharov, G., Segev, G., and Shahaf, I. Tight tradeoffs in searchable symmetric encryption. In: H. Shacham and A. Boldyreva (eds.), CRYPTO 2018, Part I, *LNCS*, vol. 10991, pp. 407–436. Springer, Heidelberg (Aug. 2018).

Axb20. Axboe, J. Flexible I/O Tester (2020). URL `https://github.com/axboe/fio`.

BFHM08. Berenbrink, P., Friedetzky, T., Hu, Z., and Martin, R. On weighted balls-into-bins games. Theoretical Computer Science, vol. 409(3):(2008), pp. 511–520.

BMO17. Bost, R., Minaud, B., and Ohrimenko, O. Forward and backward private searchable encryption from constrained cryptographic primitives. In: B.M. Thuraisingham, D. Evans, T. Malkin, and D. Xu (eds.), ACM CCS 2017, pp. 1465–1482. ACM Press (Oct. / Nov. 2017).

Bol01. Bollobás, B. Random Graphs. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2 ed. (2001).

Bos16. Bost, R. *Σοφος*: Forward secure searchable encryption. In: E.R. Weippl, S. Katzenbeisser, C. Kruegel, A.C. Myers, and S. Halevi (eds.), ACM CCS 2016, pp. 1143–1154. ACM Press (Oct. 2016).

Bos21a. Bost, R. Implementation of Tethys, and Pluto (2021). URL `https://github.com/OpenSSE/opensse-schemes`.

Bos21b. Bost, R. Supplementary materials (2021). URL `https://github.com/rbost/tethys-sim-rs`.

CGKO06. Curtmola, R., Garay, J.A., Kamara, S., and Ostrovsky, R. Searchable symmetric encryption: improved definitions and efficient constructions. In: A. Juels, R.N. Wright, and S. De Capitani di Vimercati (eds.), ACM CCS 2006, pp. 79–88. ACM Press (Oct. / Nov. 2006).

CJJ$^+$13. Cash, D., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M.C., and Steiner, M. Highly-scalable searchable symmetric encryption with support for Boolean queries. In: R. Canetti and J.A. Garay (eds.), CRYPTO 2013, Part I, *LNCS*, vol. 8042, pp. 353–373. Springer, Heidelberg (Aug. 2013).

CJJ$^+$14. Cash, D., Jaeger, J., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M.C., and Steiner, M. Dynamic searchable encryption in very-large databases: Data structures and implementation. In: NDSS 2014. The Internet Society (Feb. 2014).

CT14. Cash, D. and Tessaro, S. The locality of searchable symmetric encryption. In: P.Q. Nguyen and E. Oswald (eds.), EUROCRYPT 2014, *LNCS*, vol. 8441, pp. 351–368. Springer, Heidelberg (May 2014).

DK12. Drmota, M. and Kutzelnigg, R. A precise analysis of cuckoo hashing. ACM Transactions on Algorithms - TALG, vol. 8.

DP17. Demertzis, I. and Papamanthou, C. Fast searchable encryption with tunable locality. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 1053–1067. ACM (2017).

DPP18. Demertzis, I., Papadopoulos, D., and Papamanthou, C. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In: H. Shacham and A. Boldyreva (eds.), CRYPTO 2018, Part I, *LNCS*, vol. 10991, pp. 371–406. Springer, Heidelberg (Aug. 2018).

DW05. Dietzfelbinger, M. and Weidling, C. Balanced allocation and dictionaries with tightly packed constant size bins. In: L. Caires, G.F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung (eds.), ICALP 2005, *LNCS*, vol. 3580, pp. 166–178. Springer, Heidelberg (Jul. 2005).

FF56. Ford, L.R. and Fulkerson, D.R. Maximal flow through a network. Canadian journal of Mathematics, vol. 8:(1956), pp. 399–404.

FNO20. Falk, B.H., Noble, D., and Ostrovsky, R. Alibi: A flaw in cuckoo-hashing based hierarchical oram schemes and a solution. Cryptology ePrint Archive, Report 2020/997 (2020). `https://eprint.iacr.org/2020/997`.

GM11. Goodrich, M.T. and Mitzenmacher, M. Privacy-preserving access of outsourced data via oblivious RAM simulation. In: L. Aceto, M. Henzinger, and J. Sgall (eds.), ICALP 2011, Part II, *LNCS*, vol. 6756, pp. 576–587. Springer, Heidelberg (Jul. 2011).

GR98. Goldberg, A.V. and Rao, S. Beyond the flow decomposition barrier. Journal of the ACM (JACM), vol. 45(5):(1998), pp. 783–797.

HKL^{+}18. Hanaka, T., Katsikarelis, I., Lampis, M., Otachi, Y., and Sikora, F. Parameterized orientable deletion. In: SWAT (2018).

KMO18. Kamara, S., Moataz, T., and Ohrimenko, O. Structured encryption and leakage suppression. In: H. Shacham and A. Boldyreva (eds.), CRYPTO 2018, Part I, *LNCS*, vol. 10991, pp. 339–370. Springer, Heidelberg (Aug. 2018).

KMW10. Kirsch, A., Mitzenmacher, M., and Wieder, U. More robust hashing: Cuckoo hashing with a stash. SIAM Journal on Computing, vol. 39(4):(2010), pp. 1543–1561.

MM17. Miers, I. and Mohassel, P. IO-DSSE: Scaling dynamic searchable encryption to millions of indexes by improving locality. In: NDSS 2017. The Internet Society (Feb. / Mar. 2017).

MPC^{+}18. Mishra, P., Poddar, R., Chen, J., Chiesa, A., and Popa, R.A. Oblix: An efficient oblivious search index. In: 2018 IEEE Symposium on Security and Privacy, pp. 279–296. IEEE Computer Society Press (May 2018).

PR04. Pagh, R. and Rodler, F.F. Cuckoo hashing. Journal of Algorithms, vol. 51(2):(2004), pp. 122–144.

SEK03. Sanders, P., Egner, S., and Korst, J. Fast concurrent access to parallel disks. Algorithmica, vol. 35(1):(2003), pp. 21–55.

Sta21. Statista. Shipments of hard and solid state disk (hdd/ssd) drives worldwide from 2015 to 2021. `https://www.statista.com/statistics/285474/hdds-and-ssds-in-pcs-global-shipments-2012-2017/` (2021).

Wie17. Wieder, U. Hashing, load balancing and multiple choice. Foundations and Trends in Theoretical Computer Science, vol. 12(3–4):(2017), pp. 275–379.