SSE and SSD: Page-Efficient Searchable Symmetric Encryption

Angèle Bossuat [*]   Raphael Bost [†]   Pierre-Alain Fouque [‡]   Brice Minaud [§]

Michael Reichle [¶]

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 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,

---

[*]Quarkslab. Work done while at Université de Rennes 1. Email: angele.bossuat@irisa.fr

[†]Direction Générale de l'Armement. 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. Email: raphael_bost@alumni.brown.edu

[‡]Université de Rennes 1, France. Email: pa.fouque@gmail.com

[§]Inria, Paris, France and École Normale Supérieure, CNRS, PSL, Paris, France. Email: brice.minaud@inria.fr

[¶]Inria, Paris, France and École Normale Supérieure, CNRS, PSL, Paris, France. Email: michael.reichle@inria.fr

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.

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]
$\Pi_{\mathrm{pack}}, \Pi_{\mathrm{2lev}}$	$\mathcal{O}(1)$	$\mathcal{O}(X)$	$\mathcal{O}(1)$	$\mathcal{O}(p)$	[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

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 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 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 SSE(D) can be built generically from any DIP scheme D. The page efficiency and storage efficiency of the SSE scheme 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, 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, 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, PlutoDIP and NilusDIP$_t$. Both are variants of the main TethysDIP construction, and offer trade-offs of practical interest between storage efficiency and page efficiency. In particular, NilusDIP rests on the observation that our main analytical results, regarding the optimality and stash size bound of 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 1. Nevertheless, to our knowledge, no previous SSE scheme with linear storage has achieved page efficiency $\mathcal{O}(1)$. The $\Pi_{\sf pack}$ 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 $\Pi_{\sf 2lev}$ variant from [CJJ$^+$14] incurs the same cost, because it handles short lists in the same way as $\Pi_{\sf pack}$. 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, $\Pi_{\sf pack}$ and its variants are dynamic schemes, whereas Tethys is static.

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 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, 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, TethysDIP (resp. 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 TethysDIP (resp. 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 TethysDIP generalizes some of these results to items of variable size, we only consider the static setting. Extending TethysDIP to the dynamic setting is an interesting open problem.

Finally, some aspects of 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 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 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 DB(w). In some cases, the server does not recover DB(w) directly; instead, the server recovers some data d and sends it to the client. The client then recovers DB(w) from d.

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

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

- EDB $\leftarrow$ Setup(K, DB): the setup algorithm takes as input the master key K and a database DB, and outputs an encrypted database EDB.

- $(\tau, \rho) \leftarrow$ 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).

- $d \leftarrow$ Search(EDB, τ): The search algorithm takes as input the token τ and the encrypted database EDB and outputs some data d.

- $s \leftarrow$ Recover(ρ, d): the recovery algorithm takes as input the output d of the Search algorithm, and potentially some auxiliary information ρ, and outputs the set DB(w) of document identifiers matching the queried keyword w.

The 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 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 $\mathcal{L}_{\mathsf{Setup}}$(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 $\mathcal{L}_{\mathsf{Search}}$(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 $K \leftarrow \mathsf{KeyGen}(1^\lambda)$, $\mathsf{EDB} \leftarrow \mathsf{EDBSetup}(K, DB)$, $\tau \leftarrow \mathsf{TokenGen}(K, w)$.

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,* $|\mathsf{RdPat}(\tau, \mathsf{EDB})| \le R \cdot P$, *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*,* $|\mathsf{EDB}| \le S \cdot |DB|$.

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,* $|\mathsf{PgPat}(\tau, \mathsf{EDB})| \le aX + b$, *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,* $|\mathsf{PgPat}(\tau, \mathsf{EDB})| \le P \cdot X$, *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 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 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 1 is tight.

Proof of Theorem 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 $\sum a_i \le Rx$, 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 $\sum(a_i/p + 2) \le Rx/p + 2L$. 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 $R + 2L$. □

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)*:*

- $m \leftarrow$ Size(n)*: Takes as input a number of values n. Returns a number of buckets m.*

- $(B, S) \leftarrow$ Build(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* $m \leftarrow$ 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.*

- $\mathcal{I} \leftarrow$ Lookup(m, K, ℓ)*: 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 Lookup(m, K_i, ℓ_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 $m \leftarrow$ Size$(|\mathsf{M}|)$, and $(B, S) \leftarrow$ Build(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 $m \leftarrow$ 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* $(m, B, S) \leftarrow$ Build(M) *and any key K for which the values* 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* $(m, B, S) \leftarrow$ Build(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* $(m, B, S) \leftarrow$ Build(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 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 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 SSE(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 SSE(D) has storage efficiency $S + 1$. During the

---

Algorithm 1 SSE(D)

KeyGen(1^λ)

1: Sample keys $K_{\mathsf{PRF}}, K_{\mathsf{Enc}}$ for PRF, Enc with security parameter λ
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: $\mathsf{M} \leftarrow \{K_i, \mathsf{DB}(w_i) : 1 \le i \le k\}$
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, τ_i)

1: Initialize empty set R
2: Parse τ_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

---

search process, SSE(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 SSE(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 2 (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 SSE(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 $(e_{i,1}, \ldots, e_{i,\ell_i})$. 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 $\{1, \ldots, m\}$. H_1 is uniformly random among functions mapping into $\{1, \ldots, m/2\}$, and H_2 is uniformly random among functions mapping into $\{m/2+1, \ldots, m\}$. (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 5.)

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 $\mathsf{M} = \{(K_i, (e_{i,1}, \ldots, e_{i,\ell_i})) : 1 \leq i \leq k\}$. It outputs a dictionary B such that $B[i]$ contains the values $e_{i,j}$ that are stored in bucket number i, for $i \in \{1, \ldots, m\}$, 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

Algorithm 2 TethysDIP

$\text{Build}(m, \mathsf{M} = \{(K_i, (e_{i,1}, \ldots, e_{i,\ell_i})) : 1 \le i \le k\})$

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:     $B[v] \leftarrow \{e_{i,j} : \text{origin of edge } e_{i,j} \text{ is } v\}$
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)

$\text{Lookup}(m, K, \ell \le p)$

1: returns $\{H_1(K), H_2(K)\}$

---

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.

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 $p \cdot \omega(\log \lambda)/\log m$ 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.* $\ell_i \le p$ 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* $(e_{i,1}, \ldots, e_{i,p}), \ldots, (e_{i,p(x_i-1)+1}, \ldots, e_{i,px_i})$ 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 $K_i \parallel j$ (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 $K_i \parallel j$.

The 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 $j \le \lceil \ell_i/p \rceil$. 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 $p \cdot \omega(\log \lambda)/\log m$. 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 TethysDIP. This yields a SSE scheme Tethys := SSE(TethysDIP). As the 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 3. *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* $\mathcal{O}(p \cdot \omega(\log \lambda)/\log m)$.

The 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 TethysDIP still holds with overwhelming probability, assuming the PRF is secure.

4.1   Stash Size Analysis

We now analyze the stash size of TethysDIP. We proceed by first showing that the stash size achieved by 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 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. TethysDIP is optimal in the sense that the assignment it outputs achieves the minimum possible stash size among all valid assignments. In other words, 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 $V = \{1, \ldots, m\}$, 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 (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 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 5 (Stash size bound). *Let $\varepsilon > 0$ be an arbitrary constant, and let $p, n \geq p, m = (2 + \varepsilon)n/p$, $s = n^{o(1)}$ be integers. Let L be an arbitrary vector of integers such that $\max L \leq p$ and $\sum L = n$.*

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

In particular, a stash of $\omega(\log \lambda)/\log m$ 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, 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 $\omega(\log \lambda)/\log m$ pages suffices to ensure a negligible probability of failure. If $m = \lambda^{\Omega(1)}$, $\omega(1)$ 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 5 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 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 5, 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].

16
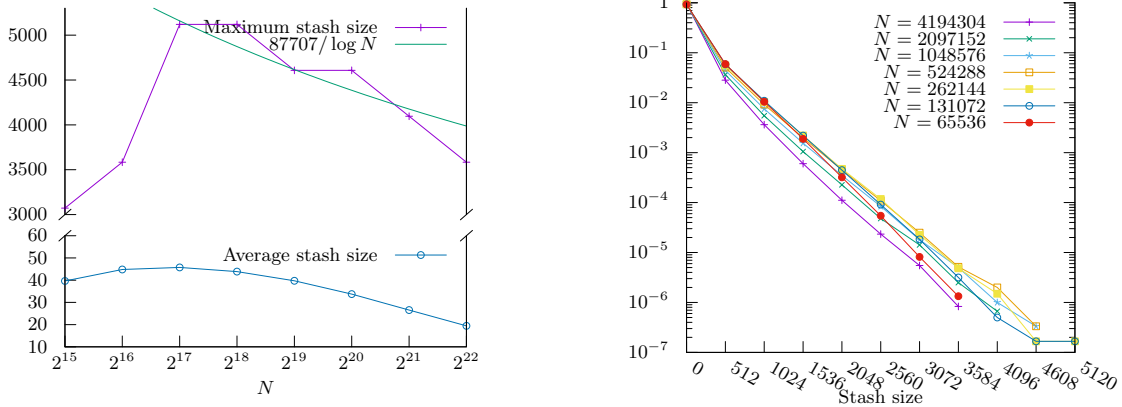
(a) Maximum and average stash size for fixed $\varepsilon = 0.1$.

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

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

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 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 $\widetilde{\mathcal{O}}(n^{3/2})$; 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
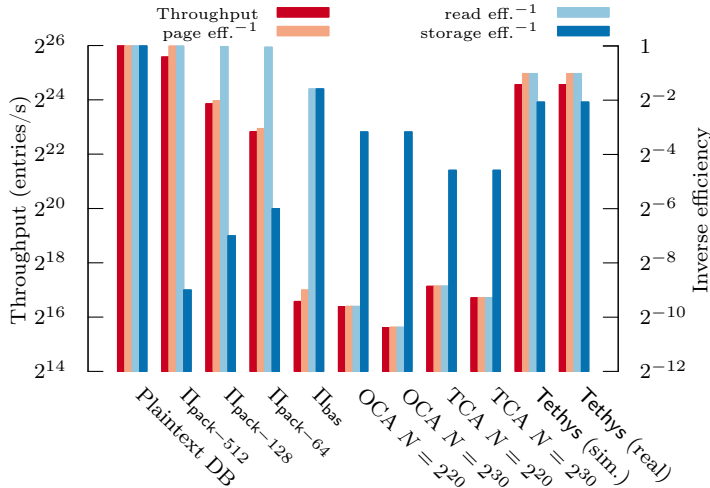
17

Figure 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.

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 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 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

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.

[Cac]   Cactus Technologies. Solid state drives 101: Everything you ever wanted to know. URL https://www.cactus-tech.com/resources/blog/details/solid-state-drives-101.

[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).

[CGLS17]   Chan, T.H.H., Guo, Y., Lin, W.K., and Shi, E. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In: T. Takagi and T. Peyrin (eds.), ASIACRYPT 2017, Part I, *LNCS*, vol. 10624, pp. 660–690. Springer, Heidelberg (Dec. 2017).

[CGPR15]   Cash, D., Grubbs, P., Perry, J., and Ristenpart, T. Leakage-abuse attacks against searchable encryption. In: I. Ray, N. Li, and C. Kruegel (eds.), ACM CCS 2015, pp. 668–679. ACM Press (Oct. 2015).

[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).

[CLZ11]   Chen, F., Lee, R., and Zhang, X. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In: Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture, HPCA '11, pp. 266–277. IEEE Computer Society, Washington, DC, USA (2011).

[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.

[FJF15]   Ford Jr, L.R. and Fulkerson, D.R. Flows in networks, vol. 54. Princeton university press (2015).

[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).

[Goo]   Goossaert, E. Coding for SSDs. URL http://codecapsule.com/2014/02/12/coding-for-ssds-part-1-introduction-and-table-of-contents/.

[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.

[MKC+12] Min, C., Kim, K., Cho, H., Lee, S.W., and Eom, Y.I. Sfs: random write considered harmful in solid state drives. In: Proceedings of the 10th USENIX conference on File and Storage Technologies, pp. 12–12. USENIX Association (2012).

[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).

[MP20]   Minaud, B. and Papamanthou, C. Note on generalized cuckoo hashing with a stash. arXiv:2010.01890 (2020). https://arxiv.org/abs/2010.01890.

[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).

[Pag01]   Pagh, R. On the cell probe complexity of membership and perfect hashing. In: 33rd ACM STOC, pp. 425–432. ACM Press (Jul. 2001).

[Pet15]   Petersen, T.K. Eulerian numbers. Springer (2015).

[PPYY19] Patel, S., Persiano, G., Yeo, K., and Yung, M. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In: L. Cavallaro, J. Kinder, X. Wang, and J. Katz (eds.), ACM CCS 2019, pp. 79–93. ACM Press (Nov. 2019).

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

[PSWW18] Pinkas, B., Schneider, T., Weinert, C., and Wieder, U. Efficient circuit-based PSI via cuckoo hashing. In: J.B. Nielsen and V. Rijmen (eds.), EUROCRYPT 2018, Part III, *LNCS*, vol. 10822, pp. 125–157. Springer, Heidelberg (Apr. / May 2018).

[Sch95]   Schoenmakers, L. A new algorithm for the recognition of series parallel graphs. Tech. rep., CWI Amsterdam (1995).

[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 world-
           wide   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.

A   Cuckoo Hashing

Cuckoo hashing was introduced by Pagh and Rodler at ESA 2001 [PR04], and proceeds as follows. We want
to insert a set S of n items into two tables T_1, T_2 of size m each. The construction is parametrized by two
hash functions $H_1, H_2 : S \to [1, m]$. Each item $s \in S$ may be inserted either in the first table at index $H_1(s)$,
or in the second table at index $H_2(s)$. Let us say that an allocation of each item to one of its two possible
locations is *suitable* if every item is allocated, and no location receives more than one item. In [Pag01], Pagh
shows that if we set $m = (1 + \varepsilon)n$ for any constant $\varepsilon > 0$, then a suitable allocation exists with probability
$1 - \mathcal{O}(1/n)$. Note that the load factor is slightly less than 50%.

In [PR04], Pagh and Rodler introduce a dynamic version of the scheme, and coin the term "cuckoo
hashing". Deletions can be implemented naturally in constant time. The term "cuckoo hashing" comes
from the insertion procedure, where item s is inserted at location $H_1(s)$ in T_1. If the location was already
occupied by some item s', that item is relocated to its other possible location $H_2(s')$ in T_2, dislodging any
item previously stored there, and so on, forming a chain reaction. Insertion succeeds if an empty location
is eventually reached, or fails if the number of relocations goes over a certain threshold T. If an insertion
fails, the entire table is rebuilt from the ground up using different hash functions, until a suitable allocation
is found. That event is called a rehash. Pagh and Rodler show that with a suitable choice of threshold
$T = \mathcal{O}(\log n)$, the insertion procedure completes in expected constant time, including the cost of rehashes.

Cuckoo hashing lookups can be performed using at most two memory accesses. This strong worst-case
guarantee makes cuckoo hashing attractive in real-time systems. Another feature of cuckoo hashing is that
if table entries are encrypted, one may look up an item s by retrieving the buckets $H_1(s)$ in T_1, $H_2(s)$ in T_2,
regardless of whether the item s exists in either table. By decrypting the bucket, one may retrieve the item if
it was present in either table, or ascertain that it was not. To an adversary observing memory accesses to the
cuckoo tables, those two cases are indistinguishable. This feature makes cuckoo hashing attractive as a basis
for oblivious hashing algorithms [CGLS17], with applications to Oblivious RAM and other cryptographic
constructions, such as differentially-private SSE [PPYY19].

In the scope of this article, we are only concerned with the static setting, so we are mainly interested
in the probability of rehash. As with many cryptographic applications in the area of oblivious algorithms,
we want this probability to be negligible. Of particular interest for that purpose is the addition of a *stash*,
introduced by Kirsch, Mitzenmacher, and Wieder [KMW10].

A stash of size s may receive up to s arbitrary items, saving the need to allocate them to either table.
Kirsch *et al.* show that with a stash of size s, the probability of failure decreases from $\mathcal{O}(n^{-1})$ to $\mathcal{O}(n^{-s})$.
Since we want the probability of failure to be negligible, and $n = \mathsf{poly}(\lambda)$, $s = \omega(1)$ is required. The original
result by Kirsch *et al.* assumes that the stash size s is constant. However, several extensions to superconstant
s exist in the literature, see Lemma 14 and the discussion that precedes.

Before closing this short introduction to cuckoo hashing, we provide a few technical clarifications that
may be helpful to the reader.

Distribution of hash functions.   Every analytical result on cuckoo hashing regarding rehash probabilities
or average-case complexity is over the randomness of the hash functions. In some of the literature, the hash
functions are picked among an explicit family, such as universal hash functions. This is in fact the case in
the original article by Pagh and Rodler. Explicit hash function families have the advantage that the analysis
does not require any cryptographic assumption, but they are slower in practice than cryptographic hash
functions. In view of this, other results in the literature, especially within cryptography, assume uniformly

random hash functions, which may be realized by efficient cryptographic hash functions, at the cost of some computational assumption. In this article, we always use uniformly random hash functions, and rely on results from the literature that do the same.

Number of tables. Some cuckoo hashing schemes in the literature use two tables, with each hash function mapping into one table, following the original article by Pagh and Rodler. Other schemes use a single table, with both hash functions mapping into the full table. To the best of our knowledge, there is no substantial difference in the behavior of standard cuckoo hashing whether one or two tables are used. However, in some extensions of cuckoo hashing, and especially when considering buckets of capacity $t > 1$ [DW05] (*i.e.* each location in the cuckoo table can receive up to t items), a single table is always used. This is because the phase transition that occurs when the load factor reaches $1/2$ disappears, so there is no longer a reason to double the storage space by considering two tables.

In this article, we rely on results that use two tables for the analysis of Tethys and Tethys′, because we ultimately relate the probability of rehash of those schemes to the probability of rehash of standard cuckoo hashing; whereas we rely on results that use a single table for the analysis of Nilus$_t$, because we ultimately relate the probability of rehash of that scheme to the probability of rehash of cuckoo hashing with buckets of size $t > 1$. Using a different number of tables for each scheme would lead to cumbersome formalism. Instead, we chose to use a single table throughout, but modify the distribution of hash functions specifically for Tethys and Tethys′, so that each hash function maps into one half of the table. This is of course equivalent to using two tables.

If a single table is used, the table size is set to $(2+\varepsilon)n/t$ (in the case of buckets of size t), for any constant $\varepsilon > 0$. The storage efficiency is $2 + \varepsilon$. When two tables are used, each table has size $m = (1+\varepsilon)n$, so that the storage efficiency is $2 + 2\varepsilon$. Of course, up to a change of variable $\varepsilon \mapsto \varepsilon/2$, this is $2 + \varepsilon$. We slightly abuse notation and write that the storage efficiency is $2 + \varepsilon$ in both cases.

B Max Flow

We present a brief overview of flows in networks (see [FJF15] for more details). A network $N = (G, c, s, t)$ consists of a directed graph $G = (V, E)$ with edge capacities $c : E \to \mathbb{R}^+$, a source $s \in V$, as well as a sink $t \in V$. For each edge $(u, v) \in E$, we assume that its corresponding back edge (v, u) is in E, with $c(v, u) = 0$ if necessary. Note that in this work, we mainly consider multi-graphs with constant capacity 1. For simplicity, we only consider graphs with at most one edge between a given pair of nodes. The extension of definitions to multi-graphs is straightforward.

A flow assigns a value, its flow, to each edge which has to satisfy two properties. First, the flow of an edge cannot exceed its capacity. Second, the sum of the flows entering a node must equal the sum of the flows exiting that node, except for the source and the sink. The value of the flow is its outgoing flow from source s. Note that this value is equal to the incoming flow at sink t, as the flow has to be conserved throughout the network.

Definition B.1 (Flows, Flow Value). *A flow $f : E \to \mathbb{R}^+$ is an assignment of edge weights satisfying the following constraints:*

- *Capacity constraint: $\forall (u, v) \in E : f(u, v) \leq c(u, v)$*

- *Conservation of flows:*

$$\forall v \in V \setminus \{s, t\} : \sum_{u \in V : \{u,v\} \in E} f(u, v) = \sum_{u \in V : \{v,u\} \in E} f(v, u).$$

The value of the flow is defined as

$$|f| = \sum_{u \in V : \{s,u\} \in E} f(s, u) = \sum_{u \in V : \{u,t\} \in E} f(t, u).$$

Table 2 – Latencies of random reads on SSDs and HDDs (in μs). Measurements were made using `fio` in `sync` mode. See Section 5 for a description of the evaluation setup.

Read width	SSD		HDD	
	Mean	Std. Dev.	Mean	Std. Dev.
4 KiB	96.03	13.55	6×10^3	4×10^3
32 KiB	153.9	19.39	6×10^3	5×10^3
64 KiB	250.4	25.11	6×10^3	4×10^3
512 KiB	1.142×10^3	60.98	8×10^3	4×10^3
1 MiB	2.084×10^3	55.31	10×10^3	4×10^3
64 MiB	121.2×10^3	400.7	300×10^3	30×10^3

A cut is a partition S, T of the nodes V such that the source resides in S and the sink resides in T. The capacity of a cut is the sum of the weights of edges leaving S.

Definition B.2 (Cuts, Capacity). *A cut $C = (S, T)$ is a partition of V such that $s \in S$ and $t \in T$. The capacity of a cut C is defined as $\mathsf{cap}(C) = \sum_{(u,v) \in E \cap (S \times T)} c(u, v)$.*

Often, we are interested in finding a cut with minimal capacity (min cut) under all possible cuts. Similarly, we often look for a flow with maximal value (max flow) under all possible flows. The latter is for example performed in TethysDIP. The following theorem shows that these problems are closely related. We later apply it in the efficiency analysis of TethysDIP in Appendix G.

Theorem 6 (Max-Flow Min-Cut). *For any network, the maximal flow value from s to t is equal to the minimal cut capacity over cuts separating s and t.*

Algorithms. Finding a MaxFlow of a network is a well-studied problem, and many efficient solutions exist. The Ford-Fulkerson algorithm [FF56] finds the max flow f_{max} of a network in running time $\mathcal{O}(|E| f_{max})$. The algorithm looks (in a greedy manner) for augmenting paths in the graph. As the name suggests, an augmenting path is a path (under capacity) that increases the max flow by sending more flow along its edges. Since the greedy decision might not be optimal, the algorithm considers the residual graph which allows for reallocation of some flow from previous rounds. If no more augmenting path can be found, the solution is optimal. Note that the Ford-Fulkerson might not terminate if the capacities are irrational. In our algorithm TethysDIP, the capacities are all set to 1, so termination is guaranteed.

There are more efficient max flow algorithms, such as [GR98] with running time $\widetilde{\mathcal{O}}(n^{3/2})$. Since the flow algorithm is only part of a (one-time) precomputation step, we implemented TethysDIP using the Ford-Fulkerson algorithm.

C   On Practical SSD Performance

In recent years, server storage for latency-sensitive applications has increasingly adopted solid-state drives (SSD) in place of hard-disk drives (HDD), whose performance has been stagnating over the last decade. Today, accessing a random 4 KiB memory block has an average latency of 96 μs on an SSD, and 5.9 ms on an HDD (*cf.* Table 2). Moreover, because of the physical and logic design of SSDs, they enjoy excellent concurrent access performance compared to HDDs (*cf.* Figure 3a). However, random write performance degrades very quickly on SSDs, even in the non-concurrent case, advocating for specifically designed data structures able to get the most out of this new technology.

The design of SSDs, as well as opportunities for leveraging their high level of parallelism, have already been well described in works such as [CLZ11]. We give here a small summary that will be useful to explain our needs when applied to SSE. We acknowledge that the actual details are a lot more complex than what follows, and refer the curious reader to existing surveys [Cac, Goo, MKC+12] for more precision.

An SSD is built around flash memory packages, which themselves include several independent dies. Each of those chips is composed of several memory planes representing multiple blocks, the atomic storage level
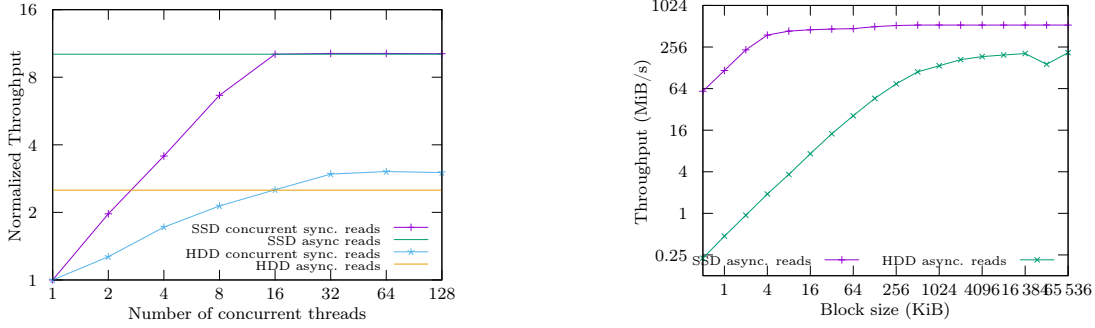
(a) Normalized concurrent throughput, function of the number of concurrent threads. Normalized throughput is the measured throughput divided by the throughput with a single thread.

(b) Throughput of asynchronous reads, function of the size of the reads, ranging from 512 B to 64 MiB.

Figure 3 – Throughput of HDD and SSD, in log-log scale. Measurements were made using `fio` in `sync` mode using parallel jobs, except for the asynchronous benchmarks, made in `libaio` mode. See Section 5 for the evaluation setup.

(from the SSD user's point of view) being the page: reads and writes have to be aligned on a page boundary, and accessing a single byte implies accessing the whole page. The size of a page ranges from 2 KiB to 16 KiB, and most SSDs have between 128 and 512 pages per block. Because of the physical workings of SSDs, the physical bits of the pages wear out every time they are erased, and a page cannot be overwritten without having been erased. The granularity of erasure in an SSD is at the block level: a single page cannot be erased. As a consequence, SSDs use a wear-leveling algorithm to evenly distribute page writes over the entire SSD, even when those writes target a single 'logical' page: in that case, the page containing the old data is marked as stale, and the new data is written to a new page. This is combined with a garbage collection algorithm that erases full blocks when the corresponding pages are all marked as stale.

Because of this complex data layout, the controller has to keep track of the physical location (or *physical block address*) of each logical page (also called *logical block address* – LBA) with a mechanism called the *Flash Translation Layer (FTL)*.[1] The exact behavior of FTL is highly dependent on the drive's controller, but the main point is that, contrary to HDD, consecutive logical pages are not guaranteed to be physically consecutive (and often guaranteed not to be consecutive), making locality of accesses less meaningful. Generally speaking, SSDs benefit from large writes of consecutive ranges, and these writes must be aligned on the page size to avoid write amplification and throughput loss. Similarly, page-aligned reads improve performance by reducing the number of pages that are physically accessed.

The end result of this behavior may be observed on Figure 3b. The figure plots the throughput when continuously reading data that is split into disjoint, randomly-located blocks, as a function of the block size. For HDDs, the throughput increases linearly with the block size, showing that storing the data into few contiguous blocks is beneficial, as expected. For blocks larger than around 1 MiB, the increase is sublinear but still significant, until very large block sizes. This shows that locality is indeed an important factor. Since for SSE schemes, optimizing locality typically comes at the expense of read efficiency, the best throughput will be reached for a certain balance between locality and read efficiency (supporting the idea of tunable locality, see Appendix K.1).

As for SSDs, the picture is straightforward. The throughput increases linearly with the block size until the block size reaches one page. This is because when the block size is less than one page, the SSD must read the entire page for every block regardless. On the other hand, once the block size is larger than one page, packing data into larger blocks provides little benefit. Thus, the most important performance criterion is to pack the data into as few pages as possible. In the concrete case of SSE workloads, the strong correlation

---

[1]The FTL of an Open-Channel SSD can be directly managed by the operating system. This allows for greater control and flexibility but requires a significant amount of RAM and CPU cycles to store and manage the translation table.

between page efficiency and throughput is also analyzed in the experiments of Section 5.

D   Tightness of Theorem 1

In this section, we show that the upper bound $R + 2L$ in Theorem 1 is tight. Since $R + 2L$ is a function of two variables, the meaning of "tightness" may not be obvious. An upper bound $f(x) \leq g(x)$ in a single variable is usually said to be tight when equality holds for arbitrarily large x's. In other words, the set $A_0 = \{x : f(x) = g(x)\}$ is *unbounded*, in the sense $\forall x \exists x' \in A_0, x' \geq x$. More loosely, the bound is also said to be tight if the sets $A_\varepsilon = \{x : f(x) > g(x) - \varepsilon\}$ are unbounded for all $\varepsilon > 0$. For our purpose, we will say that a bound $f(x, y) \leq g(x, y)$ over two variables is tight if the sets $A_\varepsilon = \{(x, y) : f(x, y) > g(x, y) - \varepsilon\}$ are unbounded, where unbounded means $\forall x, y \exists (x', y') \in A_\varepsilon, x' \geq x, y' \geq y$. In other words, the definition is unchanged, except that in order to account for having two variables, the notion of "unbounded" uses coordinate-wise order. We note that this is a natural, but rather weak notion of tightness for two variables.

We now move on to showing tightness for Theorem 1. Start from an arbitrary SSE scheme with locality L and page efficiency R. Pick (L', R') such that $L' \geq L$, $R' \geq L$, and $R'p = L'(p + 2)$ (such a pair always exists). Consider a database D consisting of a single keyword with p matches. On input D, we modify the SSE scheme to behave as follows (its behavior on other inputs is unchanged). The SSE scheme first expands the number of matching documents by a factor R' to $R'p = L'(p + 2)$ by adding dummy data. (When looking up the keyword, the scheme will read all data, including dummy data.) It then splits the data into L' chunks of $p + 2$ items each. The SSE scheme allocates 3 contiguous server pages to each chunk, and stores the chunk contiguously over those pages, starting one position before the second page. That is, the chunk overlaps all 3 pages: 1 item in the first page, p items in the second page, 1 item in the last page. Observe that the locality of the modified scheme is L', and its read efficiency is R'. Indeed, it reaches those values on input D, while its locality (resp. read efficiency) is at most L (resp. R) on other inputs. Theorem 1 implies that its page efficiency is at most $R' + 2L'$. On the other hand, its page efficiency on input D is $3L' = R'(1 + 2/p) + 2L' \to_{p \to \infty} R' + 2L'$. This shows that the sets A_ε are unbounded.

E   Encoding

In the SSE framework $\mathsf{SSE}(D)$ for some DIP scheme D (Section 3.1), after the server has processed a search query for keyword w, the client receives some data d, consisting of encrypted buckets where list elements matching w are stored. The buckets may also contain elements from unrelated lists. After decrypting the buckets, the client would like to be able to identify which elements belong to the desired list. Towards that end, some metadata must be added to the elements, so that elements belonging to a specific list can be identified unambiguously.

Adding metadata generates some storage overhead, which conflicts with the fact that we want every encrypted bucket to fit within a page exactly. To avoid this problem, we propose to add the metadata by way of preprocessing the multi-map at the input of D, *before* running the build algorithm of D, and in a way that is transparent to the D. That way, the contents of the buckets at the output of the build algorithm still benefit from the correctness guarantees of the DIP scheme: namely, there are no more than p values per bucket, including all required metadata.

A simple solution is to increase the size of each value to add a tag identifying the list it belongs to. The number p of values per page is decreased accordingly, so that a page still contains p values. The goal of this section is to present other ways to encode metadata that are more storage-efficient.

We call an algorithm that adds the required metadata an *encoding*. In the scope of this section, the input multi-map provided as input for D is written as a set of lists $\{(K_i, v_{i,1}, \ldots, v_{i,m_i})\}_{i=1,\cdots,k}$, where K_i is a key, and the $v_{i,j}$'s are its associated values. The encoding algorithm transforms each such list of the multi-map into a new list $(e_{i,1}, \ldots, e_{i,\ell_i})$. After all lists are encoded, they are given to D as input of the build algorithm.

Assume that one key K_i is stored on κ memory blocks, and one value $v_{i,j}$ is stored on ν memory blocks, where a *block* is some arbitrary memory unit of β bits, such as an octet ($\beta = 8$) or 64-bit word ($\beta = 64$).

For simplicity, one may think of $\kappa = \nu = 1$ and $\beta = 64$. The best efficiency is achieved for $\beta = 1$, although this choice may be inconvenient for practical implementations.

E.1 EncodeBasic: Basic Encoding

The simple solution mentioned earlier is that every value $e_{i,j}$ of the multi-map is a full key-value pair. That is, each input list $(K, v_1, \ldots, v_\ell)$ is encoded into $(e_1, \ldots, e_\ell) = ((K, v_1), \ldots, (K, v_\ell))$. A value of the multi-map provided as input to D is itself a key-value pair of $\kappa + \nu$ blocks. Retrieving the values corresponding to key k from a bucket is trivial. In terms of storage cost, the original input of $\kappa + \ell\nu$ blocks is transformed into $\ell(\kappa + \nu)$ blocks. Assuming $\kappa = \nu$, in the worst case (large ℓ), this solution doubles the storage cost.

E.2 EncodeSeparate: Separation Encoding

To reduce the storage cost, instead of using one key per value in the list, a more efficient solution is to use one key for each of the two buckets where values can be assigned. To do so, each value e_i of the multi-map is now a single block of β bits. Recall that a key (resp. a value) is stored in κ (resp. ν) such blocks. At the cost of increasing κ if necessary, assume that $\lceil \log(p) \rceil$ bits of a key can be set aside to store an integer in $\{1, \ldots, p\}$. Each input tuple $(K, v_1, \ldots, v_t)$ of $\kappa + t\nu$ blocks is encoded as $2\kappa + t\nu$ blocks: the first 2κ blocks contain two separate copies of the key K, and the remaining blocks contain the v_i's. Let $(e_1, \ldots, e_\kappa, e'_1, \ldots, e'_\kappa, f_1, \ldots, f_{t\nu})$ denote these blocks, in order. After the build algorithm is run, the list is split between two buckets A and B: the scheme D assigns a values to bucket A, b values to bucket B, and possibly c values to the stash, with $a + b + c = 2\kappa + t\nu$.

1. First, consider the case that no value is sent to the stash ($c = 0$). If both a and b are greater than κ, we store one copy of the key in each bucket, followed by the values containing values, in order. That is, bucket A receives $(e_1, \ldots, e_\kappa, f_1, \ldots, f_{a-\kappa})$, and bucket B receives the complement $(e'_1, \ldots, e'_\kappa, f_{a-\kappa+1}, \ldots, f_{t\nu})$. Each key also contains the number of values stored in the same bucket (using the $\lceil \log(p) \rceil$ bits set aside earlier). If $a \leq \kappa$, we do not store any value in A, and store one copy of the key plus all values in B (preserving order). This is always possible, since $a \leq \kappa$ implies $b \geq \kappa + t\nu$. Likewise, if $b \leq \kappa$, we do the opposite. Note that since $c = 0$, it is not possible to have $a \leq \kappa$ and $b \leq \kappa$ simultaneously.

2. If some values are sent to the stash ($c \geq 1$), we proceed in the same way, with a few tweaks. If $a \leq \kappa$ and $b > \kappa$, as in the previous case, we store nothing in A, store $(e'_1, \ldots, e'_\kappa, f_1, \ldots, f_{b-\kappa})$ in B, and send $(f_{b-\kappa+1}, \ldots, f_\nu)$ to the stash. (We do not detail how values are stored in the stash: since it is very small, its storage format is not critical.) This is always possible, since $a \leq \kappa$ implies $b + c \geq \kappa + t\nu$. Likewise if $b \leq \kappa$ and $a > \kappa$, we do the opposite. If $a \leq \kappa$ and $b \leq \kappa$ hold simultaneously, we send all values $(f_1, \ldots, f_\nu)$ to the stash.

In the end, each bucket receives values structured as sequences of one key followed by values, with the key containing the number of values that follow. If the bucket is not full, it is padded with zeros. It is clear that this allows the user to retrieve the values corresponding to a given key from the two buckets. In terms of storage cost, the original input of $\kappa + \ell\nu$ blocks is transformed into $2\kappa + \ell\nu$ blocks: we add an extra key. (In addition, if the key fills the full κ blocks, $\lceil \log p \rceil$ bits are also added.) In most cases, this cost of one extra key is negligible compared to the list of values that follows. That is why we adopt this approach as a default solution for our DIP scheme instantiations such as TethysDIP. On the other hand, in some edge cases, such as if $\kappa \approx \nu$, and all lists are of size 1, this solution would increase the storage by 50%. The EncodeCompact scheme below avoids this behavior, and behaves well in all cases, at the cost of less simplicity.

Many variants of the previous storage scheme are possible. For example, if $\kappa = \nu = 1$ and $\beta = 64$, and if the first bit of a key block and the first bit of a document block can be set aside, it is more efficient to proceed as follows. Set the first bit of a key to 0, and the first bit of a value to 1. The user can parse the sequences of $(key, (values))$ from the bucket unambiguously, without having to set aside $\lceil \log p \rceil$ within the key to store the number of values that follow.

E.3 EncodeCompact: Compact Encoding

Regarding storage efficiency, the previous solution adds the cost of storing an extra key compared to basic plaintext storage. This extra cost can be reduced by replacing one of the two keys by a pointer to the position to the remainder of the sequence in the other bucket, which in practice is much smaller (and only depends on p). In more detail, set aside $3\lceil \log p\rceil$ bits within each key, instead of $\lceil \log p\rceil$ in the previous solution, increasing κ as necessary. The encoding proceeds exactly the same as EncodeSeparate, except a single copy of the key is used. The only situation where this raises an issue is when the list is split into $a > \kappa$ values in bucket A and $b > \kappa$ values in bucket B. In that case, one of two buckets (picked arbitrarily) receives the key and as many values as will fit (as in EncodeSeparate). The other bucket receives the remaining values, without any key. Once all lists are assigned, in a given bucket, values that come with a key are stored first, followed by elements that are not accompanied with a key (grouped by key). The $3\lceil \log p\rceil$ set aside in a key are used as follows: the first $\lceil \log p\rceil$ bits encode the number of values of the list that follow the key. The remaining two sequences of $\lceil \log p\rceil$ bits encode the start and end position of the values of the list that are stored in the other bucket. This allows the user to retrieve all values corresponding to a given key unambiguously. Compared to the previous scheme, instead of two keys, each extended by $\lceil \log p\rceil$ bits each, we use a single key, extended by $3\lceil \log p\rceil$ bits. If we set $\beta = 1$ (which is always possible), the total overhead compared to plaintext storage is only $3\lceil \log p\rceil$ bits per list. For a typical value of $p = 512$, this is 27 bits.

Remark. The number of values n at the input of D is leaked to the server via the number of buckets $m = (1 + \varepsilon)n/p + 3$ of TethysDIP. If the basic encoding scheme EncodeBasic is used, $n = N = |DB|$, which is why in the security model of Section 3.1, $\mathcal{L}_{\mathsf{Setup}}(DB) = N$. However, it should be noted that if another encoding scheme is used, n is some linear combination of N and the number of keywords k, so the leakage profile of $\mathsf{SSE}(D)$ is slightly modified.

F Security Proof of the SSE Framework

In this section, we prove Theorem 2. The proof follows the standard game-based paradigm. Let p be the page size. Recall that the leakage during setup is $\mathcal{L}_{\mathsf{Setup}}(DB) = |DB| = N$, and the leakage during search is $\mathcal{L}_{\mathsf{Search}}(w_i) = (\ell_i, \mathsf{sp})$, where ℓ_i is the number of sublists for the list of documents matching keyword w_i, and sp is the search pattern. Our goal is to show that SSE is $\mathcal{L}$-adaptively semantically secure.

Let $\mathcal{S}$ denote the simulator. During setup, $\mathcal{S}$ receives $\mathcal{L}_{\mathsf{Setup}}(DB) = |DB| = N$, and is tasked with simulating the database EDB. To do this, $\mathcal{S}$ samples a fresh key K'_{Enc} for Enc, and generates $B[1] = \mathsf{Enc}_{K'_{\mathsf{Enc}}}(z), \ldots, B[m] = \mathsf{Enc}_{K'_{\mathsf{Enc}}}(z)$, where z is an all-zero input of length p. Note that m can be computed via N. Similarly, the simulator creates a simulated table T' obtained by generating N entries mapping κ to χ, where $\kappa \leftarrow \{0,1\}^{2\lambda}$ and $\chi \leftarrow \{0,1\}^{\lceil \log N\rceil}$ are sampled uniformly and independently for each entry. The simulator returns the simulated database $\mathsf{EDB}' = (B[1], \ldots, B[m]), T'$.

During a search query, the simulator receives $\mathcal{L}_{\mathsf{Search}}(w_i) = (\ell_i, \mathsf{sp})$, and is tasked with simulating the search token $\tau_i = (K_i, m_i)$. If the search pattern sp reveals that the same keyword has already been searched in the past, the simulator simply replies the same token. Otherwise, K'_i is chosen uniformly at random among the keys κ in the table T' that have not yet been used for a previously queried keyword. The simulated mask is set to $M'_i = T'[K'_i] \oplus \ell_i$. The simulated search token is (K'_i, M'_i). Observe that the server recovers the correct value ℓ_i.

We need to prove that the real game is indistinguishable from the ideal game. We proceed with a sequence of hybrid games.

- Hybrid 0 is the real experiment.

- Hybrid 1 is the same as Hybrid 0 except each pair (K_i, m_i) is generated uniformly at random instead of being the output of $\mathsf{PRF}_{K_{\mathsf{PRF}}}(w_i)$. The table T and the search tokens are still generated as in the real game, using this new sampling of (K_i, m_i). The PRF security of PRF implies that the advantage of an adversary trying to distinguish Hybrid 0 from Hybrid 1 is negligible.

- Hybrid 2 is the same as Hybrid 1 except the table T is replaced by the simulated table T', and the search token (K_i, m_i) by the simulated search token (K_i', M_i'). That is, those values are generated as in the ideal game. The key observation is that the view of the adversary in this hybrid is identical to its view in the previous hybrid. Indeed, every entry in the table T of Hybrid 1 is uniformly random and independent, and every search token (K_i, m_i) of Hybrid 1 is uniformly random among tokens that satisfy: (1) K_i is a fresh entry in T (not used for a previously queried keyword); and (2) $\ell_i = T[K_i] \oplus m_i$. This is precisely the same distribution as the simulated values T', (K_i', M_i'). The advantage of an adversary trying to distinguish between Hybrid 1 and Hybrid 2 is zero.

- Hybrid 3 is the same as Hybrid 2, except a flag FAIL is raised if the input of Build is not valid, which occurs if two distinct lists have the same identifier, i.e. there exist $i \neq j$ such that $K_i' = K_j'$. By a standard birthday argument, this probability is negligible. More precisely, there are at most $N = \mathsf{poly}(\lambda)$ values K_i', so there are less than N^2 pairs (i, j), and each pair has a probability of collision $2^{-2\lambda}$, hence by the union bound, the probability of collision is upper-bounded by $N^2 2^{-2\lambda} = \mathsf{negl}(\lambda)$. It follows that the advantage of an adversary trying to distinguish Hybrid 2 from Hybrid 3 is negligible.

- Hybrid 4 is the same as Hybrid 3 except the real database EDB is replaced by the simulated database EDB'. The real key K_{Enc} and the simulated key K_{Enc}' used to generated the two databases are identically distributed, so the only difference between the two databases lies in the plaintext messages being encrypted. Since Enc is IND-CPA secure, it follows that the advantage of an adversary trying to distinguish Hybrid 3 from Hybrid 4 is negligible.

- Hybrid 5 is the ideal experiment. The server's view in the ideal experiment and in Hybrid 5 are identically distributed, so the advantage of an adversary trying to distinguish Hybrid 4 from Hybrid 5 is zero.

This concludes the proof of Theorem 2.

G Proof of Optimality of TethysDIP

In this section, we prove that the data-independent packing scheme TethysDIP is optimal, in the sense that it minimizes the number of values going to the stash, among all possible ways of splitting lists between their two destination buckets. In the analysis, we restrict ourselves to lists of maximal size p. This is sufficient, as TethysDIP handles arbitrary list lengths by splitting up lists into sublists of maximal size p.

G.1 Setup and Notation

Let us recall some notation. If S is a set, $|S|$ denotes its cardinality. If L is a multiset, $|L|$ denotes its cardinality, including multiplicity, and $\sum L$ denotes the sum of its elements $\sum L = \sum_{\ell \in L} \ell$. We will often work with multisets of positive integers representing list lengths (i.e. the number of values in a given list of a multi-map M). In that context, we might call an element of L a list, and the total number of elements contained in the lists is $\sum L$.

For a given multi-map M, let n be the total number of values, let p be the bucket size, let m be the total number of buckets, and let s be the size of the stash (counted as number of values). We always assume m is a multiple of 2; otherwise, an extra bucket is added. The hash functions are chosen as follows: H_1 is uniformly random among functions mapping into $\{1, \ldots, m/2\}$; H_2 is uniformly random among functions mapping into $\{m/2 + 1, \ldots, m\}$.

We have k lists. The i-th list contains ℓ_i values, for $1 \leq i \leq k$. The total number of values is $n = \sum \ell_i$. We assume that the lists have length at most p: $\forall i, \ell_i \leq p$. The i-th list is split between buckets $H_1(K_i)$ and $H_2(K_i)$. For simplicity, compared to the main body of the article, we ignore K_i, setting $K_i = i$.

Graph notation. For v a vertex in a graph G, let $\mathsf{out}_G(v)$ denote its outdegree. Whenever a notation or statement takes as input an undirected graph, we may also apply it to directed graphs, by assimilating

the directed graph with its corresponding undirected graph (where edge orientations are forgotten). If G is undirected, we say that a directed graph D *arises* from G if its corresponding undirected graph is G. By *subgraph*, we always mean vertex-induced subgraph. That is, a *subgraph* $G' = (V', E')$ of $G = (V, E)$ is such that $V' \subseteq V$ and $E' = (V' \times V') \cap E$. Finally, since we allow multiple edges with the same start and end points, note that the "set" E of edges is actually a multiset (this type of graph is sometimes called a multigraph).

G.2 Optimality of TethysDIP

As in Section 4.1, we regard TethysDIP as an algorithm that takes as input an undirected graph $G = (V, E)$ with $|V| = m$ vertices and $|E| = n$ edges, and an integer p. TethysDIP outputs a directed graph D arising from G. The page size $p \in \mathbb{N}$ is fixed throughout the section (this avoids passing it as a parameter everywhere).

Definition G.1. *Let $G = (V, E)$ be a directed graph. The* overflow overflow(G) *is defined as:*

$$\mathsf{overflow}(G) = \sum_{v \in V} \max(0, \mathsf{out}_G(v) - p)$$

The main result of this section is Theorem 7, which states that TethysDIP is optimal, in the sense that it minimizes the overflow. This holds regardless of the hash functions: in this section, we do not care how they are picked. TethysDIP takes as input a graph and capacity p, and always orients the edges optimally for the given metric, regardless of what the graph looks like.

Definition G.2. *Let G be an undirected graph. The* optimal overflow optimal(G) *is the minimal overflow, taken over all directed graphs arising from G.*

Theorem 7 (Optimality of TethysDIP). *Let G be an undirected graph. Let D be the directed graph output by TethysDIP on input G. Then* overflow$(D) =$ optimal(G).

Before proving the theorem, we begin with a few definitions and lemmas.

Definition G.3. *Let $G = (V, E)$ be an undirected graph. The* unavoidable overflow $\mathsf{U}(G)$ *of G is defined as* $\mathsf{U}(G) = |E| - p|V|$. *If $\Delta \subseteq V$, we write $\mathsf{U}_G(\Delta)$ for $\mathsf{U}(G')$, where G' is the subgraph of G induced by Δ.*

Lemma 8. *Let $G = (V, E)$ be an undirected graph. Then*

$$\mathsf{optimal}(G) = \max_{\Delta \subseteq V} \mathsf{U}_G(\Delta).$$

Proof. Pick $\Delta \subseteq V$. Let D be a directed graph arising from G such that overflow$(D) =$ optimal(G), and let $D' = (\Delta, E')$ be the subgraph of D induced by Δ. We have:

$$\mathsf{overflow}(D) \geq \mathsf{overflow}(D') \geq \sum_{v \in \Delta} (\mathsf{out}_{D'}(v) - p) = |E'| - p|\Delta| = \mathsf{U}_G(\Delta).$$

This proves optimal$(G) \geq \max_{\Delta \subseteq V} \mathsf{U}_G(\Delta)$.

Let us prove optimal$(G) \leq \max_{\Delta \subseteq V} \mathsf{U}_G(\Delta)$. Again, let D be a directed graph arising from G such that overflow$(D) =$ optimal(G). Define $\Gamma \subseteq V$ to be the set of vertices of D whose outdegree is strictly more than p. Let $\Delta \supseteq \Gamma$ be the set of vertices that can be reached from Γ by following a directed path in D. Observe that the outdegree of every vertex v in Δ must be at least p. Otherwise, there would exist a directed path from a vertex in Γ to some v with $\mathsf{out}_D(v) < p$. Flipping the direction of every edge along this path reduces the overflow by 1. This would contradict the optimality of D. Let $D' = (\Delta, E')$ be the subgraph of D induced by Δ. We have overflow$(D) =$ overflow$(D') = \sum_{v \in \Delta} (\mathsf{out}_{D'}(v) - p) = |E'| - p|\Delta| = \mathsf{U}(D') = \mathsf{U}_G(\Delta)$. Hence optimal$(G) \leq \max_{\Delta \subseteq V} \mathsf{U}_G(\Delta)$. $\square$

Lemma 8 is structurally very similar to [SEK03, Theorem 5] and [Sch95, Theorem 1], which prove the same relationship between a notion of *maximum load* and *unavoidable load* of a graph. This structural similarity is not a coincidence: [SEK03, Theorem 5] can be deduced from Lemma 8 by observing that the maximum load is equal to the smallest p such that the overflow is zero. When $p = 1$, Lemma 8 is also closely related to the standard result in cuckoo hashing that the number of overflowing values is equal to the minimum number of edges that must be deleted so that no component has more than one cycle.

We are now ready to prove Theorem 7.

Proof of Theorem 7. Let $G = (V, E)$ be the undirected graph at the input of TethysDIP. Let $\overline{C} = (V_{st}, E_{\overline{C}})$ be the directed graph obtained within TethysDIP right after running the max flow algorithm, and before flipping the edges that carry flow. So $\overline{C}$ includes the source and sink nodes s and t ($V_{st} = V \cup \{s, t\}$), as well as a number of edges from the source to V and from V to the sink. In this proof, we never consider edges with capacity $x > 1$; instead, we use x separate edges of capacity 1, and all our graphs are unlabeled. Let $C = (V, E_C)$ denote the subgraph of $\overline{C}$ induced by V, *i.e.* we remove the source, sink, and adjacent edges.

After building $\overline{C}$, the TethysDIP algorithm computes a max flow. Let f denote the total amount of flow going from the source to the sink in this max flow. Finally, let $\overline{D} = (V_{st}, E_{\overline{D}})$ denote the directed graph obtained *after* flipping the edges that carry flow; and $D = (V, E_D)$ the subgraph of $\overline{D}$ induced by V. The graph D is the graph that is returned by TethysDIP. Our goal is to prove $\mathsf{overflow}(D) = \mathsf{optimal}(G)$.

To do so, the strategy is to use Lemma 8, and exhibit a subset of vertices $\Delta \subseteq V$ such that $\mathsf{U}_G(\Delta) = \mathsf{overflow}(D)$. This is enough to conclude, because using Lemma 8, $\mathsf{optimal}(G) \geq \mathsf{U}_G(\Delta) = \mathsf{overflow}(D) \geq \mathsf{optimal}(G)$, so $\mathsf{optimal}(G) = \mathsf{overflow}(D)$ as desired. We now build such a Δ.

By the Max-Flow Min-Cut Theorem, there exists a partition of V into disjoint subsets S and T such that $s \in S$, $t \in T$, and the number of edges going from S to T in C is equal to f (recall we only use edges of capacity 1). Further, all edges from S to T carry flow. We claim that $\Delta = S$ is our witness: that is, $\mathsf{U}_G(S) = \mathsf{overflow}(D)$. In order to prove that statement, we start with a few facts.

> *Fact 1.* $\mathsf{overflow}(D) = \mathsf{overflow}(C) - f$. When we flip the edges of $\overline{C}$ that carry flow to build $\overline{D}$, we flip edges along f disjoint paths $(s, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k), (v_k, t)$ going from source to sink. Flipping edges along such a path amounts to moving one overflowing value from v_1, which is over capacity (outdegree $> p$) to v_k, which is under capacity (outdegree $< p$). Each one of the f flipped paths reduces the overflow of the graph by 1.

> *Fact 2.* $\forall v \in V$, if $\mathsf{out}_D(v) > p$, there are exactly $\mathsf{out}_D(v) - p$ edges from the source s to v in $\overline{D}$; if $\mathsf{out}_D(v) < p$, there are exactly $p - \mathsf{out}_D(v)$ edges from v to the sink t in $\overline{D}$. This fact was true in C by construction, and it is preserved when we flip edges along a path from source to sink, so it remains true in D.

> *Fact 3.* $\forall v \in S, \mathsf{out}_D(v) \geq p$. If v was under capacity in D (outdegree $< p$), there would be an edge from v to the sink, but there is no edge going from S to T in D (they were all flipped).

> *Fact 4.* $\forall v \in V$, if $\mathsf{out}_D(v) > p$, then $v \in S$. Indeed, if $\mathsf{out}_D(v) > p$, there is an edge going from the source to v in D, but there is no edge from S to T in D, so $v \in S$.

From the last two facts, it follows that in D, all overflowing vertices are in S, and all vertices in S are over capacity (or at capacity), so $\mathsf{overflow}(D) = \sum_{v \in S} \mathsf{out}_D(v) - p|S|$. Let $G' = (S, E')$ be the subgraph of G induced by S. In D, every edge leaving from a vertex in S stays within S, since there is no edge from S to T, so $\sum_{v \in S} \mathsf{out}_D(v) = |E'|$. We conclude that $\mathsf{overflow}(D) = |E'| - p|S| = \mathsf{U}_G(S)$. $\qquad\square$

Remark. It is be possible to do a direct proof of the theorem, without going through Lemma 8 (which is only used in the easy direction), and without using the notion of unavoidable overflow. This is because by Facts 3 and 4, all overflowing buckets are in S, so the overflow of D is entirely in S, and it cannot be further reduced, because the buckets in S are already full or overflowing, and there is no edge leaving S. However, the current proof gives more insight, and the notion of unavoidable load, as well as Lemma 8, will be needed in Appendix H to prove Theorem 18.

H Bounding the Stash Size of TethysDIP

In this section, we prove that for any optimal packing scheme in the sense of Theorem 7 (including TethysDIP), a stash of size $\omega(\log \lambda)/\log m$ suffices to ensure a negligible probability of failure. A failure occurs when the stash is too small to receive all reassigned values. Again, it is enough to consider lists of size up to p (as TethysDIP handles arbitrary lists by splitting them into lists of maximal size p).

Recall that we want to store lists of length $\ell_i \leq p$ with $\sum \ell_i = n$ into $m = \mathcal{O}(n/p)$ buckets of capacity p. Fix an arbitrary stash size s. The values of list i must be stored into buckets $H_1(K_i)$, $H_2(K_i)$, or in the stash, where the hash functions are modeled as uniformly random over the ranges $\{1, \ldots, m/2\}$ and $\{m/2+1, \ldots, m\}$ respectively. We succeed if every bucket receives at most p values, and the stash receives at most s values. Algorithmically, we already know how to proceed: we have proved that TethysDIP is optimal, in the sense that it minimizes the number of values stored in the stash, among all possible assignments. This means that if it is *possible* to succeed (for a given pair of hash functions), then TethysDIP *will* succeed with probability one. What we want to do now is to determine m and s such that it is in fact possible to succeed, except with negligible probability. Note that TethysDIP already fixes m according to the following analysis, for now we assume that it is a variable. Here and in the remainder, to ease notation, we assume that n and s are multiples of p (which can be enforced by adding at most p values).

As in Section 4.1, we view TethysDIP as an algorithm with input $G = (V, E)$, an undirected graph with $|V| = m$ vertices and $|E| = n$ edges, and output D, a directed graph resulting from orienting edges in G. Recall that TethysDIP minimizes the overflow $\sum_{v \in V} \max(0, \mathsf{out}(v) - p)$, which is equal to the stash size. Our goal in this section is to find a suitable upper bound on that quantity, which should hold with overwhelming probability.

In a nutshell, our proof strategy is to reduce the analysis of the stash size to a cuckoo hashing problem with n/p balls and m buckets (of capacity 1), and stash size s/p. We divide the proof into three steps. In the first step, we show that having n/p lists of the maximum length p is a worst case for the *expected* stash size. With the right view of the problem, the result follows from a convexity argument, using a *majorization* technique. In the second step, we show that in that worst case, our problem becomes equivalent to a cuckoo hashing problem, and hence inherits the same upper bound as in [KMW10]. However, since our reduction to the worst case was only in expectancy, we are not done. In the third step, we derive an upper bound that holds with overwhelming probability from the previous results. Let us start with some notation.

Weighted graphs. We change our view on graphs to facilitate our analysis. Instead of allowing for multiple edges between two nodes, we mark each edge with a weight that indicates the number of edges between the nodes. We assume that there exists an edge between any given pair of nodes. If there was no edge in the corresponding multigraph, the weight of the edge will be 0. In this view, a graph is a tuple $G = (V, w)$, where V are the nodes and $w : V^2 \to \mathbb{N}$ is the weight function. For undirected graphs, we set $w(u, v) = w(v, u)$. We define the weight of a node $v \in V$ as $w(v) = \sum_{u \in V} w(v, u)$.

In the analysis, we often allow for graphs with real edge weights $w(u, v) \in \mathbb{R}$. In that case, when the set of vertices is fixed, undirected graphs will sometimes be viewed as vector spaces over $\mathbb{R}$, in the natural way. That is, let $G_0 = (V, w_0), G_1 = (V, w_1)$ be undirected graphs with vertices V and weights w_i. Let $x \in \mathbb{R}$. We define the addition of two graphs by $G_0 + G_1 = (V, w)$, where $\forall u, v \in V, w(u, v) = w_0(u, v) + w_1(u, v)$. Similarly, scalar multiplication is defined by $x \cdot G_0 = (V, w)$, where $\forall u, v \in V, w(u, v) = x \cdot w_0(u, v)$.

Input graph. Let $p \in \mathbb{N}$. The input graph is induced by a multi-map $\mathsf{M} = \{(K_i, (e_{i,1}, \ldots, e_{i,\ell_i})) : 1 \leq i \leq k\}$, the number of buckets m, and a pair of hash functions $H = (H_1, H_2)$ mapping into $\{1, \ldots, m/2\}$ and $\{m/2, \ldots, m\}$ respectively. As the concrete values K_i and $e_{i,j}$ of M are not important for the analysis, we will capture the relevant lengths ℓ_i in a tuple $L = (L[1], \ldots, L[k]) \in [0, p]^k$. Note that again, we allow for lengths $\ell_i \in \mathbb{R}$ during the analysis. We call $G = (V, w) = \mathsf{graph}_H^m(L)$ the *input graph* of TethysDIP, defined as follows: $V = \{1, \ldots, m\}$, and for $u, v \in V$:

$$w(u, v) = \sum_{\substack{i \in [1, k], \\ \{H_1(i), H_2(i)\} = \{u, v\}}} L[i].$$

32

The definition ensures that $w(u, v) = w(v, u)$.

Revisiting the (optimal) overflow. As in Appendix G.2, we require the notions of overflow overflow, optimal overflow optimal, and unavoidable overflow (see definitions G.1 to G.3). Even though the graph representation slightly differs in this section, the definitions can be adapted naturally (by replacing the outdegree $\text{out}_G(v)$ of a node v with its weight $w(v)$ and $|E|$ with the sum of all edge weights). Also note that Lemma 8 remains true for real weights (the proof did not rely on weights being integral, and still holds for real weights, with no change).

Probability of Failure. Given $m, p, s \in \mathbb{N}$, a tuple L of list lengths with $\max L \leq p$, and a pair of hash functions $H = (H_1, H_2)$ mapping into $\{1, \ldots, m/2\}$ and $\{m/2, \ldots, m\}$ respectively, define $\text{Fail}_{m,p,s}(L, H)$ as the event that it is *impossible* to assign the values of every list i into the buckets $H_1(K_i)$, $H_2(K_i)$, or in the stash, such that no bucket receives more than p values, and the stash receives no more than s values. Note that the number of values n is implied by the parameter L: $n = \sum L$. We want an upper bound on the probability of $\text{Fail}_{m,p,s}(L, H)$, over the random choice of the hash functions H, as a function of m, n, p, s, and independently of the choice of L (as long as L satisfies the constraints of our problem: $\max L \leq p$, and $\sum L = n$).

The probability of failure $\text{Fail}_{m,p,s}(L, H)$ can be expressed expressed differently as follows. Because TethysDIP is optimal in the sense of minimizing the stash size (see Theorem 7), $\text{optimal}(G)$ is equal to the stash size after running TethysDIP on a graph G. Thus for some fixed p, the probability of failure is equal to:

$$\Pr[\text{Fail}_{m,p,s}(L, H)] = \Pr[\text{optimal}(\text{graph}_H^m(L)) > s],$$

where the probability is over the uniformly random choice of H_1, H_2. We are ultimately interested in upper bounding this probability, but for now, we focus on the expected stash size $\mathbb{E}[\text{optimal}(\text{graph}_H^m(L))]$.

H.1 Step 1: Majorization

The main result of this step is the following lemma. Note that the list L_D corresponds to an arbitrary multi-map (that is provided as input to TethysDIP) and L_M to a multi-map, where every key maps to exactly p values. After proving this lemma for some function f_H, we show that the stash size fulfills the requirements.

Lemma 9 (Bound of Expectancy). *Let $L_M \in \{0, p\}^k, L_D \in [0, p]^k$ with $\sum L_D = \sum L_M$. Let $f_H : [0, p]^k \mapsto \mathbb{R}$ be convex and such that $\mathbb{E}[f_H(L)]$ is invariant under list permutation. It holds that:*

$$\mathbb{E}[f_H(L_M)] \geq \mathbb{E}[f_H(L_D))],$$

where the expectancy is over the (uniformly random) choice of $H = (H_1, H_2)$.

Because the hash functions (H_1, H_2) are uniform, composing their input with any fixed permutation still yields a uniform distribution. It follows that the expected stash size $\mathbb{E}[\text{optimal}(\text{graph}_H^m(\cdot))]$ is invariant under list permutation. We will later show that the stash size is a convex function, and thus, the lemma above shows that the list L_M is the worst case for the expectation of the stash size.

The proof of Lemma 9 follows the structure of [BFHM08], and uses the so-called *majorization* technique. We now introduce the notions and tools necessary for our proof. We refer to [BFHM08] for more details.

Definition H.1 (Majorization). *For two non-increasing vectors $L_1, L_2 \in \mathbb{R}^k$ with $\sum L_1 = \sum L_2$, we say that L_1 majorizes L_2, written $L_1 \succ L_2$, if*

$$\forall \kappa \in [1..k] : \sum_{i=1..\kappa} L_1[i] \geq \sum_{i=1..\kappa} L_2[i].$$

Definition H.2 (T-Transformation). *A T-transformation matrix T is a matrix of the form $\mathsf{T} = \delta \mathsf{I} + (1 - \delta)\mathsf{P}$, where $\delta \in [0, 1]$, I is the identity matrix, and P is a permutation matrix that swaps exactly two coordinates. We write $L_1 \overset{\mathsf{T}}{\Longrightarrow} L_2$, if L_2 can be derived form L_1 by applying one T-transformation.*

Lemma 10. *For $L_1, L_2 \in \mathbb{R}^k, L_1 \succ L_2$ if and only if L_2 can be derived from L_1 by successive applications of at most $k - 1$ T-Transformations.*

The above lemma allows us to transform one list into another, given that the target list is majorized by the original list. For L_M and L_D defined as in Lemma 9), it holds that L_M majorizes L_D, so we can transform L_M into L_D with $k - 1$ T-transformations. This allows us to prove Lemma 9.

Proof of Lemma 9. Without loss of generality (since f is invariant under list permutation), we assume that L_M and L_D are non-increasing vectors. Under the given constraints, it holds that $L_M \succ L_D$. Thus, L_D can be derived from L_M by $k - 1$ T-transformations:

$$L_M \xRightarrow{\mathsf{T}} L_1 \xRightarrow{\mathsf{T}} L_2 \xRightarrow{\mathsf{T}} \cdots \xRightarrow{\mathsf{T}} L_{k-2} \xRightarrow{\mathsf{T}} L_D.$$

Setting $L_M = L_0$ and $L_D = L_{k-1}$, we have $L_{i+1} = \delta_i L_i + (1 - \delta_i) L_i \mathsf{P}_i$, for all $i \in [0, k - 2]$. We receive

$$\mathbb{E}[f_H(L_{i+1})] \le \delta_i \mathbb{E}[f_H(L_i)] + (1 - \delta_i)\mathbb{E}[f_H(L_i \mathsf{P}_i)] = \mathbb{E}[f_H(L_i)],$$

since f_H is convex and $\mathbb{E}[f_H(L_i \mathsf{P}_i)] = \mathbb{E}[f_H(L_i)]$ by assumption. The result follows by induction. $\qquad\square$

We now show that the stash size $\mathsf{optimal}(\mathsf{graph}_H^m(\cdot))$ is convex. First, we show that $\mathsf{optimal}$ is convex. We will later conclude that the expected stash size is convex, since graph_H^m is linear.

Lemma 11. *The function $\mathsf{optimal}(\cdot)$ is convex, meaning that for graphs $G = (V, w), G_0 = (V, w_0)$ and $G_1 = (V, w_1)$ with $G = \delta G_0 + (1 - \delta)G_1$ for all $\delta \in [0, 1]$, it holds that:*

$$\mathsf{optimal}(G) \le \delta\mathsf{optimal}(G_0) + (1 - \delta)\mathsf{optimal}(G_1).$$

Proof. Let $G = \delta G_0 + (1 - \delta)G_1$ with weight $w = \delta w_0 + (1 - \delta)w_1$. With Lemma 8, we have that $\mathsf{optimal}(G^*) = \max_{\Delta \subseteq V} \mathsf{U}_{G^*}(\Delta)$ for $G^* \in \{G, G_0, G_1\}$. The convexity follows from the following calculation:

$$
\begin{aligned}
\mathsf{optimal}(G) = \max_{\Delta \subseteq V} \mathsf{U}_G(\Delta) &= \max_{\Delta \subseteq V} \sum_{\{u,v\} \subseteq \Delta} w(u, v) - p \cdot |V| \\
&= \max_{\Delta \subseteq V} \sum_{\{u,v\} \subseteq \Delta} \delta w_0(u, v) + (1 - \delta)w_1(u, v) - p \cdot |V| \\
&= \max_{\Delta \subseteq V} \sum_{\{u,v\} \subseteq \Delta} \delta(w_0(u, v) - p \cdot |V|) + (1 - \delta)(w_1(u, v) - p \cdot |V|) \\
&\le \max_{\Delta \subseteq V} \sum_{\{u,v\} \subseteq \Delta} \delta(w_0(u, v) - p \cdot |V|) + \max_{\Delta \subseteq V}(1 - \delta)(w_1(u, v) - p \cdot |V|) \\
&= \delta(\max_{\Delta \subseteq V} \sum_{\{u,v\} \subseteq \Delta} w_0(u, v) - p \cdot |V|) \\
&\quad + (1 - \delta)(\max_{\Delta \subseteq V} \sum_{\{u,v\} \subseteq \Delta} w_1(u, v) - p \cdot |V|)) \\
&= \delta(\max_{\Delta \subseteq V} \mathsf{U}_{G_0}(\Delta)) + (1 - \delta)(\max_{\Delta \subseteq V} \mathsf{U}_{G_1}(\Delta)) \\
&= \delta\mathsf{optimal}(G_0) + (1 - \delta)\mathsf{optimal}(G_1). \qquad\square
\end{aligned}
$$

Lemma 12. *Let $H = (H_1, H_2)$ be fixed hash functions, $L \in [0, p]^k$. Then $\mathsf{optimal}(\mathsf{graph}_H^m(L))$ is convex (as a function of L).*

Proof. We first show that graph_H^m is linear. Let $L_1, L_2 \in [0,p]^k$ be two lists and $\delta \in \mathbb{R}$. The weight w of edge $\{u,v\}$ of the graph $\mathsf{graph}_H^m(L_1 + \delta L_2)$ is

$$
\begin{aligned}
w(u,v) &= \sum_{\substack{i \in [1,k], \\ \{H_1(i),H_2(i)\}=\{u,v\}}} (L_1 + \delta L_2)[i] \\
&= \sum_{\substack{i \in [1,k], \\ \{H_1(i),H_2(i)\}=\{u,v\}}} L_1[i] + \delta \sum_{\substack{i \in [1,k], \\ \{H_1(i),H_2(i)\}=\{u,v\}}} L_2[i] \\
&= w_1(u,v) + \delta w_2(u,v),
\end{aligned}
$$

where w_1 and w_2 are the weights of $\mathsf{graph}_H^m(L_1)$ and $\mathsf{graph}_H^m(L_2)$ respectively. We conclude that graph_H^m is linear. Now let $L = \delta L_1 + (1 - \delta)L_2$ for $\delta \in [0,1]$. Linearity implies that

$$
\mathsf{graph}_H^m(L) = \delta\mathsf{graph}_H^m(L_1) + (1 - \delta)\mathsf{graph}_H^m(L_2),
$$

and Lemma 11 yields the desired result. $\qquad\square$

Thus, we can apply Lemma 9 on the stash size $\mathsf{optimal}(\mathsf{graph}_H^m(\cdot))$. As consequence, lists of the form L_M, *i.e.* multi-maps M that map keys to lists with p values, are a worst case for the expected stash size.

H.2   Step 2: Cuckoo Hashing

The previous step shows that it suffices to bound the expectancy for the particular list L_M. This means we have "gotten rid" of the parameter L, in the sense that we have reduced our problem to a version where this parameter is fixed. Next, we are going to "get rid" of the parameter p, in the same sense. The idea is that, now that both bucket capacity and list lengths are multiples of p, we can scale down the problem by a factor p. This is formalized in the next lemma.

Lemma 13. *Let $L_M \in \{0,p\}^k$ be the vector consisting of n/p copies of the value p, and denote by $L_1 \in \{0,1\}^k$ the vector consisting of n/p copies of 1. Then the distribution of the stash size is equal for both lists up to a factor p, more precisely:*

$$
\Pr[\mathsf{optimal}(\mathsf{graph}_H^m(L_M)) = s] = \Pr[p \cdot \mathsf{optimal}(\mathsf{graph}_H^m(L_1)) = s],
$$

where the probability is taken over the (uniformly random) choice of $H = (H_1, H_2)$.

Proof. We prove something stronger: for any given H, the stash sizes are equal up to a factor of p. That is, for every H, it holds that $p \cdot \mathsf{optimal}(\mathsf{graph}_H^m(L_1)) = \mathsf{optimal}(\mathsf{graph}_H^m(L_M))$.

First, let us show $p \cdot \mathsf{optimal}(\mathsf{graph}_H^m(L_1)) \geq \mathsf{optimal}(\mathsf{graph}_H^m(L_M))$. On both sides, the problem is to fit n/p lists into m buckets and a stash, where each bucket can store one list, with minimal overflow. The difference is that in case of the first problem, lists are of length 1, so the entire list i must be assigned either to $H_1(K_i)$, $H_2(K_i)$, or the stash. It cannot be split between the three. In case of the second problem, list length, bucket size, and stash size have been scaled by p. So it is the same problem, except we now have the additional freedom that each list i can be split between $H_1(K_i)$, $H_2(K_i)$, and the stash. Hence it is trivially an easier problem. Given an optimal solution for the first problem, we immediately have an optimal solution for the second problem, where in addition, no list is split. As the problem is scaled by the factor p, so will be the stash size.

Now let us show that $\mathsf{optimal}(\mathsf{graph}_H^m(L_M)) \geq p \cdot \mathsf{optimal}(\mathsf{graph}_H^m(L_1))$. Fix some H and consider the same (non-weighted) multi-graph G as in previous sections: vertices are the buckets $\{1\ldots,m\}$, and each list i for $1 \leq i \leq n/p$ gives rise to p edges between $H_1(K_i)$ and $H_2(K_i)$. A witness for the optimal overflow $s = \mathsf{optimal}(\mathsf{graph}_H^m(L_M))$ is a directed graph D arising from G such that $\mathsf{overflow}(D) = s$. We are going to progressively transform this graph D in to a graph D' arising from the graph induced by L_1 such that $\mathsf{overflow}(D') = s/p$. (Note that the argument also shows that $s/p \in \mathbb{N}$.)

First, we can assume that the outdegree of every overflowing vertex in D is a multiple of p. Otherwise, let d_i denote the outdegree of vertex i. Pick the smallest i such that $d_i > p$ and $d_i \neq 0 \mod p$. Let Γ bet the set of vertices that can be reached from i by following a directed path in D. Let D_Γ be the subgraph of D induced by Γ. By construction of G, the number of edges of D_Γ must be a multiple of p. It follows that there must exist another $j \in \Gamma$ distinct from i such that $d_j \neq 0 \mod p$. By construction of Γ, there exists a directed path from i to j. Flip all edges along this path. This does not affect the overflow of D, because it amounts to moving one value from bucket i to bucket j, and both buckets are overflowing (otherwise, flipping edges along the path would reduce the overflow, which contradicts the optimality of D). On the other hand, after flipping these edges, d_i has decreased and d_j has increased. By construction of i, $i < j$. It follows that the sequence of outdegrees $(d_1, \ldots, d_m)$ has decreased strictly for the lexicographic order. Since $\sum d_i = n$ remains constant, this process terminates after a finite number of iterations, at which point all d_i's such that $d_i > p$ are multiples of p.

Next, we repeat the same reasoning with buckets that are under capacity ($d_i < p$): pick the smallest i such that $d_i \neq 0 \mod p$, by the same argument as above there must exist a directed path to some $j > i$ such that $d_j \neq 0 \mod p$; flipping edges along this path does not affect the overflow of the graph because both vertices are under capacity; and the sequence $(d_1, \ldots, d_m)$ decreases strictly for the lexicographic order. Once this process terminates, all d_i's are multiples of p.

As a consequence, the edges of D can be partitioned into p-tuples, such that all edges in a p-tuple have the same start and end points. Let D' be the same graph as D, except every p-tuple of edges has been reduced to a single edge. If we consider that vertices in this new graph D' have a capacity of 1, the overflow of D' is exactly p times smaller than the overflow of D. $\qquad\square$

Thus, it is enough to bound the probability of failure in the case where all lists are length 1, and buckets have capacity 1. This is exactly a "cuckoo with a stash" problem, in the sense of Kirsch *et al.* [KMW10]. In that setting, Kirsch *et al.* prove an upper bound $\mathcal{O}(n^{-s})$ on the failure probability. That bound cannot be used directly, because it requires that the stash size s should be constant. In particular, it can only yield a polynomially low failure probability. A negligible failure probability is crucial in our setting, since the failure probability depends on the list length distribution, which we aim to hide. An extension of the original proof from [KMW10] to variable s is also given in [GM11], but that extension requires that the table size m should be polylogarithmic in the security parameter.

An $\mathcal{O}(n^{-s/2})$ bound for variable s was shown in [ADW14], with only the restriction that $s = \mathcal{O}(n^{1/c})$ for some constant c. The proof targets explicit hash families, but also extends to uniformly random functions, as noted in the introduction of the same article. In fact, a simpler variant of the proof directly targeting uniformly random hash functions is given in [Wie17]. For efficiency reasons, in practice, we wish to use cryptographic hash functions, modeled as uniformly random functions, so we are mainly interested in the uniform case. For that reason, we use the result from [Wie17] (extending it slightly to get a closed-form expression).

Lemma 14 (Corollary of [Wie17], Theorem 5.5)**.** *Consider the setting of cuckoo hashing where n items are inserted into two tables of size $m \geq (1 + \varepsilon)n$ each, for some constant $\varepsilon > 0$, with a stash of size s. Assume the hash functions used for cuckoo hashing are uniformly random. Then the probability that a valid cuckoo assignment fails to exist[2] is at most*

$$n^{-s} \cdot \left(\frac{2}{1 + \varepsilon} \right)^s \cdot \sum_{t=0}^{\infty} \frac{t^{8s}}{(1 + \varepsilon)^t} = \mathcal{O}\left(\sqrt{s} \cdot \left(\frac{Cs^8}{n} \right)^s \right)$$

for some constant $C \leq 2 \cdot (8(1 + 1/\varepsilon)/e)^8$.

Proof. Let f denote the probability that a valid cuckoo assignment fails to exist. The first bound on f is proved in [Wie17, Theorem 5.5]. Technically, the theorem is written in a setting where s is constant, but as already observed in [PSWW18, Appendix C], the proof is purely combinatorial, and holds for arbitrary s.

---

[2]That is, letting T_1, T_2 denote the two tables, and h_1, h_2 the hash functions, it is not possible to assign every item x into either index $h_1(x)$ in T_1, index $h_2(x)$ in T_2, or the stash, without either assigning two items to the same table location, or exceeding the stash size s.

The $\mathcal{O}\left(\sqrt{s} \cdot \left(\frac{Cs^8}{n}\right)^s\right)$ upper bound can be derived as follows. Letting A_k be the k-th Eulerian polynomial, by a classic identity [Pet15],

$$\sum_{t=0}^{\infty} t^k x^t = \frac{x \cdot A_k(x)}{(1-x)^{k+1}}.$$

Reinjecting into the first bound with $x = 1/(1+\varepsilon)$ yields

$$f \leq \left(\frac{2}{(1+\varepsilon)n}\right)^s \cdot \frac{x \cdot A_{8s}(x)}{(1-x)^{8s+1}}.$$

Since $x < 1$ and A_k is increasing, $A_{8s}(x) < A_{8s}(1) = (8s)!$. Using the upper-bound variant of Stirling's formula $k! \leq e\sqrt{k}(k/e)^k$, this yields

$$f \leq \left(\frac{2}{(1+\varepsilon)n}\right)^s \cdot \frac{x \cdot 2e\sqrt{2s}(8s/e)^{8s}}{(1-x)^{8s+1}}$$
$$\leq \left(\frac{2}{n}\right)^s \cdot \frac{x}{1-x} \cdot 2e\sqrt{2s}\left(\frac{8s}{e(1-x)}\right)^{8s}$$
$$= \mathcal{O}\left(\sqrt{s} \cdot \left(\frac{Cs^8}{n}\right)^s\right). \qquad \square$$

Lemma 14 says something about the *existence* of a solution to a cuckoo hashing problem with certain parameters. This is enough in the scope of this article, because TethysDIP outputs an optimal solution (Theorem 7). Thus, using Lemma 14, we get that, for some constant C':

$$\Pr[\mathsf{Fail}_{m,1,s}(L_1, H)] = \mathcal{O}\left(\sqrt{s} \cdot \left(\frac{C's^8}{n}\right)^s\right).$$

A straightforward computation yields the following corollary.

Corollary 15. *Let $m = (2+\varepsilon)n/p$ for some constant $\varepsilon > 0$, and $s = \mathcal{O}(n^{1/c})$ for some sufficiently large constant c. Then:*

$$\Pr[\mathsf{Fail}_{m,1,s}(L_1, H)] = \mathcal{O}\left(m^{-s/2}\right).$$

H.3 Step 3: Bounding the Probability of Failure

We will now use the results from the previous two sections to compute a bound for the probability of failure $\Pr[\mathsf{Fail}_{m,p,s}(L, H)]$, for an arbitrary list L. From Step 2, we know a bound on the probability of failure for the list L_M. From Step 1, we know that L_M is a worst case for the *expectancy* of the stash size, but not necessarily the failure probability. The remaining arguments, given in this section, are to bridge that gap. First, we recall the following elementary result.

Lemma 16. *Let X be a random variable over $\mathbb{N}$. $\mathbb{E}[X] = \sum_{i \geq 0} i \Pr[X = i] = \sum_{i \geq 0} \Pr[X > i]$.*

Corollary 17. *Let X be a random variable over $\mathbb{N}$. $\Pr[X > 0] \leq \mathbb{E}[X]$.*

We our now ready to show the main result.

Theorem 18 (Main bound). *Let $L \in [0, p]^k$ such that $\sum L = n, m = (2+\varepsilon)n/p$ for some constant $\varepsilon > 0$, $s = \mathcal{O}(n^{1/c})$ for some sufficiently large constant c. The probability that the allocation fails is bounded by:*

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

Proof. We bound the probability that $\max(\mathsf{optimal}(\mathsf{graph}_H^m(\cdot)) - s, 0) > 0$. Note that this corresponds exactly to the probability of failure $\mathsf{Fail}_{m,p,s}(L, H)$. In the following, we denote $f_H(L) = \mathsf{optimal}(\mathsf{graph}_H^m(L))$. By definition, we have:

$$\Pr[\mathsf{Fail}_{m,p,s}(L, H)] = \Pr[\max(f_H(L) - s, 0) > 0].$$

First, we transform the probability into an expression containing the expectancy using Corollary 17, such that we can apply our worst case bound of Step 1 later on:

$$\Pr[\max(f_H(L) - s, 0) > 0] \leq \mathbb{E}[\max(f_H(L) - s, 0)].$$

Lemma 12 shows that $\mathsf{optimal}(\mathsf{graph}_H^m(\cdot))$ is convex. We know that $\max(f_H - s, 0)$ is convex as composition of the increasing and convex function $g(x) = \max(x - s, 0)$ and convex function $\mathsf{optimal}(\mathsf{graph}_H^m(\cdot))$. It is also invariant under list permutation. Thus, we can apply Lemma 9 on $\max(f_H(\cdot) - s, 0)$:

$$\mathbb{E}[\max(f_H(L) - s, 0)] \leq \mathbb{E}[\max(f_H(L_M) - s, 0)],$$

where $L_M \in \{0, p\}^k$ is the vector consisting of n/p copies of the value p. We denote by $L_1 \in \{0, 1\}^k$ the vector consisting of n/p copies of 1. Lemma 13 shows that the distributions of optimal stash size are equal for L_M and L_1, up to a factor p. As consequence, this equality also holds for the expectancy. With the notation using f_H, we obtain:

$$\mathbb{E}[\max(f_H(L_M) - s, 0)] = p \cdot \mathbb{E}[\max(f_H(L_1) - s/p, 0)].$$

In the end, we get $\Pr[\mathsf{Fail}_{m,p,s}(L, H)] \leq \mathbb{E}[\max(f_H(L_1) - s/p, 0)]$. We now upper bound this expectancy. To do so, we apply the definition of the expectancy, split the resulting sum into three parts and bound each part separately. Let $P_H(i) = \Pr[\max(f_H(L_1) - (s/p + i), 0) > 0]$. By Lemma 16:

$$\begin{aligned}
\Pr[\mathsf{Fail}_{m,p,s}(L, H)] &\leq p \cdot \mathbb{E}[\max(f_H(L_1) - s/p, 0)] \\
&= p \cdot \sum_{i \geq 0} \Pr[\max(f_H(L_1) - s/p, 0) > i] \\
&= p \cdot \left(\sum_{i \geq 0} \Pr\left[\max(f_H(L_1) - (s/p + i), 0) > 0\right] \right) \\
&= p \cdot \left(\sum_{i=0}^{\log(m)} P_H(i) + \sum_{i=\log(m)+1}^{m-1} P_H(i) + \sum_{i \geq m} P_H(i) \right).
\end{aligned}$$

We consider each of the three terms above in turn. For the first term, we can apply Lemma 14, as the probability $P_H(i)$ is the probability of failure of cuckoo hashing for stash size $s/p + i - 1 \leq s/p + \log(m) = \mathcal{O}(n^{1/c})$, for some sufficiently large constant c. Thus:

$$\sum_{i=0}^{\log(m)} P_H(i) = \mathcal{O}\left(\sum_{i=0}^{\log(m)} m^{-(s/p+i)/2} \right) = \mathcal{O}\left(m^{-s/(2p)} \cdot \sum_{i=0}^{\log(m)} m^{-i/2} \right) = \mathcal{O}(m^{-s/(2p)}).$$

The last equation holds because

$$\sum_{i=0}^{\log(m)} m^{-i/2} \leq \sum_{i=0}^{\infty} m^{-i/2} = \frac{1}{1 - m^{-1/2}} = \mathcal{O}(1).$$

Let us turn to the second term. Observe that $P_H(\cdot)$ is necessarily non-increasing. In particular, $P_H(i) \leq P_H(\log m)$ for $i \geq \log m$. Applying the cuckoo hashing bound, we get $P_H(\log m) \leq m^{-(s/p+\log m)/2}$. Using

this bound in the second sum yields:

$$\sum_{i=\log(m)+1}^{m-1} P_H(i) \le m \cdot m^{-(s/p+\log m)/2}$$

$$= m^{-s/(2p)} \cdot m^{1-\log(m)/2}$$

$$= \mathcal{O}(m^{-s/(2p)}).$$

Finally, let us consider the third term. If the stash size s is larger than the total number n of values to allocate, it is trivially not possible for the stash to overflow, and the probability of failure is 0. Thus, the third term vanishes, as $P_H(i) = 0$ for $i \ge m$.

Putting everything together, we get

$$Pr[\mathsf{Fail}_{m,p,s}(L, H)] \le p \cdot (\mathcal{O}(m^{-s/(2p)}) + \mathcal{O}(m^{-s/(2p)}) + 0)$$

$$= \mathcal{O}(p \cdot m^{-s/(2p)}). \qquad \square$$

I DIP Scheme Variants

This section introduces two variants of TethysDIP: PlutoDIP and NilusDIP$_t$. These two DIP schemes yield SSE schemes Pluto = SSE(PlutoDIP) and Nilus$_t$ = SSE(NilusDIP$_t$) respectively, via the generic transformation from DIP to SSE from Section 3. (In Greek mythology, Pluto and Nilus are two of Tethys' many children.) Roughly speaking, Pluto reduces the page cost by a factor two on average, at the price of increasing server-side storage by 50%. Nilus$_t$ offers a trade-off in the other direction: page efficiency is increased by a factor t, for any integer parameter t, but storage efficiency improves from $2 + \varepsilon$ to $1 + (2/e)^{t-1}$.

I.1 The PlutoDIP Scheme

In TethysDIP, lists of values matching a given key are split into sublists $L_1, \ldots, L_{n_i}$. All but the last sublist have size exactly one page, while the last sublist may be smaller. All sublists are then stored using the TethysDIP allocation scheme.

In PlutoDIP, we proceed in the same way, but only store the last sublist L_{n_i} with the TethysDIP algorithm. All other sublists have the same size (one page), so they can be stored separately in a standard hash table HT. (A small technicality here is that some hashing schemes do not strictly fit within the DIP formalism, because their lookup procedure depends on the set of lookup keys, which is not part of the input of Lookup. However, the definition of DIP can be extended to allow this additional input. The generic conversion to SSE is unaffected, because the lookup keys used in the SSE scheme are PRF outputs, and can be simulated as uniformly random strings by the simulator.)

For Lookup, PlutoDIP first queries the hash table HT and recovers the complete pages $L_1, L_2, \ldots$ of the list, until there is a miss, meaning that there is no remaining complete page associated with the searched keyword in HT. The algorithm then makes a single query to TethysDIP, in order to fetch the last incomplete page (if any).

In the end, PlutoDIP makes n_i calls to HT and one call to TethysDIP. If we assume that HT makes close to one page access per query on average, this results in page cost $X + 2$. Thus, PlutoDIP makes only two extra page accesses beyond the minimum necessary to read the list in plaintext storage. (That statement is a little simplified, since we are neglecting the potential access overheads of the hash table HT, but similar overheads would also exist for plaintext storage.)

Unfortunately, if we wish to maintain the same minimal leakage profile as Tethys, this solution comes at the cost of a significant increase in server storage. Indeed, if we proceeded exactly as above, the number of sublists stored in TethysDIP would be equal to the number of keywords. In order to avoid leaking that information, and maintain the same leakage profile as Tethys, we must dimension both HT and TethysDIP so that they can store the entire database. As a result, the storage efficiency of PlutoDIP increases by 1 relative

to Tethys (slightly more depending of the load factor of the HT hashing scheme). In the scope of this article, it is assumed that we do not accept to leak the number of keywords. If the setting where we would accept to leak that information, the extra storage cost of Pluto disappears, and Pluto becomes the most efficient solution in practice.

I.2 The NilusDIP Scheme

Recall that TethysDIP achieves page efficiency 2, and storage efficiency $2 + \varepsilon$. NilusDIP_t is parametrized by an integer $t > 1$; it achieves page efficiency $2t$, and storage efficiency $1 + (2/e)^{t-1}$. Nilus is relevant in settings where server-side storage is more important than throughput.

NilusDIP_t uses the same algorithm as TethysDIP (Algorithm 2), with three differences: (1) the capacity of each bucket is increased from p to tp values; (2) the number of buckets is decreased from $(2 + \varepsilon)n/p + 3$ to $m = (1 + (2/e)^{t-1})n/(tp) + 3$; (3) the two hash functions H_1 and H_2 are now drawn uniformly at random among functions mapping into the set $\{1, \ldots, m\}$ of all buckets (their ranges are no longer disjoint, so the graph underlying the algorithm is no longer bipartite). The NilusDIP_t allocation algorithm is identical to Algorithm 2, except for the fact that every occurrence of "p" in Algorithm 2 is replaced by "tp". Note that the size of the lists at the input of the algorithm is still bounded by p: only the capacity of the buckets is increased. When retrieving a list, NilusDIP must read two buckets of size tp, hence lookup efficiency degrades from 2 to $2t$. On the other hand, the gap of a factor t between the size of the longest list and bucket capacity enables a improved storage efficiency $1 + (2/e)^{t-1}$. This is due to the following theorem.

Theorem 19. *Fix a constant $0 < \varepsilon \leq 0.25$. There exist constants $\alpha > 0$ and $\gamma < 1$, such that for every $t \geq 1 + \ln(1/\varepsilon)/(1 - \ln 2)$, and $1 \leq s \leq m/(10e^4)$, if we have $m \geq (1 + \varepsilon)n/(tp)$ buckets, each of capacity tp, then the probability that NilusDIP_t sends more than s values to the stash is*

$$p \cdot \mathcal{O}\left(\left(\frac{e}{m}\right)^{t+s/p} + \left(\frac{2e}{m}\right)^{2(t-1)} \left(\frac{\alpha s}{pm}\right)^{s/p+1} + m\gamma^m \right).$$

In particular, a stash of $\omega(\log \lambda)/\log(n)$ pages suffices to ensure a negligible probability of failure

The sufficient size of $\omega(\log \lambda)/\log(n)$ pages for the stash is the same as that of TethysDIP (Theorem 5). The condition $t \geq 1 + \frac{\ln(1/\eta)}{1 - \ln 2}$ is satisfied by $\varepsilon = (2/e)^{t-1}$. The resulting storage efficiency of $1 + (2/e)^{t-1}$ is essentially best possible by [KMW10, Proposition 4.4]. The proof of Theorem 19 is given in Appendix I.3. In short, the first step of the proof is to adapt the proof linking the probability of failure of TethysDIP to that of cuckoo hashing with a stash from Appendix H, establishing the same relationship between NilusDIP_t and cuckoo hashing with buckets of size t. The second step is a direct application of [MP20, Proposition 3].

I.3 Analysis of NilusDIP

This section provides a proof of Theorem 19. As noted in Appendix I.2, the first step of the proof is to adapt the proof linking the probability of failure of TethysDIP to that of cuckoo hashing with a stash from Appendix H, in order to establish the same relationship between NilusDIP_t and cuckoo hashing with buckets of size t. In short, the same reasoning applies as is, by observing that every step still holds when bucket capacity is increased from p to tp. We now go through the proof in more detail. The notation is the same as in Appendix G.2 and appendix H.

First, the optimality argument of Theorem 7 directly extends to NilusDIP_t, since NilusDIP_t is the same algorithm as TethysDIP with a different bucket capacity. (The distribution of the two hash functions H_1, H_2 has also changed, but as was noted in Section 4.1, this has no bearing on the optimality argument.) This means that the orientation of edges output by NilusDIP_t minimizes the overflow among all possible valid assignments. In other words, in order for NilusDIP_t to succeed in assigning all values to buckets without exceeding a given stash size s, it suffices that a successful orientation *exists*, as was the case with TethysDIP. This reduces the problem to a pure graph orientability problem, upper-bounding the probability that such an orientation fails to exist.

Given parameters m, n, p, t, define the multigraph G with vertices $\{1, \ldots, m\}$, where each list i of length ℓ_i gives rise to ℓ_i edges $(H_1(K_i), H_2(K_i))$. Let $H = (H_1, H_2)$. Since vertices have capacity tp, the overflow of $G = (V, E)$ is defined as $\sum_{v \in V} \max(0, \mathsf{out}_G(v) - tp)$. Given a multiset L of list lengths and a stash size s, define $\mathsf{NFail}_{m,p,s,t}(L, H)$ as the probability that the minimum overflow of G over all possible orientations of its edges is strictly more than s, computed over the random choice of H. By the earlier optimality argument, $\mathsf{NFail}_{m,p,s,t}(L, H)$ is equal to the probability of failure of $\mathsf{NilusDIP}_t$, on input lists with length distribution L.

Recall that the proof for the stash size bound of $\mathsf{TethysDIP}$ in Appendix H is divided in three steps. The first step is a convexity and majorization argument. The statements and proofs for the first step are unchanged, up to replacing "p" by "tp" whenever "p" was used to represent the size of a bucket, and replacing $\mathsf{Fail}_{m,p,s}$ by $\mathsf{NFail}_{m,p,s,t}$.

This is also true for the second step. Regarding the proof of Lemma 13, observe that for the argument to go through, what is required is that the bucket size is a multiple of p, not that it is exactly p.

In the third step, beyond the previous textual replacements, the only difference is the use of Lemma 14, which bounds the probability of failure of cuckoo hashing with a stash. For $\mathsf{NilusDIP}$, we require a counterpart that holds for cuckoo hashing with bins of size t. That counterpart is the following proposition from [MP20], which corrects the earlier result from [KMW10, Proposition 4.3].

Lemma 20 ([MP20], Proposition 3). *Fix a constant $0 < \varepsilon \leq 0.25$. There exist constants $\alpha > 0$ and $\gamma < 1$, such that for every $t \geq 1 + \ln(1/\varepsilon)/(1 - \ln 2)$, and $1 \leq s \leq m/(10e^4)$, the following holds. Consider the cuckoo graph G for n items and $m = (1 + \varepsilon)n/t$ buckets of capacity t induced by uniformly random hash functions. The probability that there does not exist a set of s edges in G whose removal makes it possible to orient the remaining edges of G so that every vertex has outdegree at most t is*

$$\mathcal{O}\left(\left(\frac{e}{m} \right)^{t+s} + \left(\frac{2e}{m} \right)^{2(t-1)} \left(\frac{\alpha s}{pm} \right)^{s+1} + m\gamma^m \right).$$

The condition $t \geq 1 + \frac{\ln(1/\eta)}{1 - \ln 2}$ is satisfied by $\eta = (2/e)^{t-1}$. Lemma 20 holds for variable t and s; if those quantities are constant, the upper bound simplifies to $\mathcal{O}(n^{-t-s})$.

J Distinguishing Length Distributions in RandomFit

As discussed in Appendix K.1, the construction from [DP17] uses a data-dependent way of packing lists of size bounded by some quantity B, into buckets of size B. Specifically, the following allocation scheme is used: each list in turn is assigned to a bucket drawn uniformly at random among buckets that have enough space left. To retrieve a list during a search, a separate position map table records where each list is stored. Let us call this allocation scheme RandomFit (by analogy with first-fit bin packing). RandomFit creates a dependency between the memory location where a list is stored (visible to the server when the list is searched), and the length distribution of other lists. The resulting leakage profile is non-standard: most SSE schemes leak no information about unqueried keywords. Concretely, it is possible for an attacker distinguish between two possible length distributions of unqueried keywords. We now sketch such a distinguisher.

The distinguisher holds in the standard SSE adversarial model from Section 2.2. Such a distinguisher does not contradict the security proof of [DP17] because it only exploits leakage information explicitly allowed by the scheme, but stresses the fact that the leakage profile is non-standard, and an attacker can infer information about the sizes of answers to unqueried keywords. Many variants of the distinguisher are possible; we only give one.

Consider the case that we want to fit lists of length 1 up to some length x into n bins of size $2x$. (Bins in [DP17] are twice as large as the longest list.) Assume that n is an integer square. Consider length distribution A, consisting of n lists of length x; and length distribution B, consisting of $\sqrt{n}$ lists of length x and $(n - \sqrt{n})x$ lists of size 1. Both distributions contain nx elements in total. To distinguish between those two list distributions, the adversary queries $\sqrt{n}$ lists of size x (which exist in both cases), and observes the resulting memory accesses. If a collision occurs, in the sense that at least one bin is accessed by two

different queries, the server guesses that the length distribution is A; otherwise, it guesses that the length distribution is B. This distinguisher has a constant advantage as x diverges. Indeed, for distribution A, the probability of a collision tends to a nonzero constant by the birthday paradox; while for distribution B, the probability of a collision tends to zero as x diverges, because every bin contains at least one list of length 1 with overwhelming probability, which precludes the possibility that two lists of length x are assigned to the same bin.

K   Other Applications of Data-Independent Packing

The motivation behind the definition of Data-Independent Packing is to build page-efficient SSE. Nevertheless, the idea of data-independent packing, as well as the TethysDIP solution we propose, may find other applications. To illustrate that point, in this section, we sketch two such applications.

K.1   Tunable Locality with Standard Leakage

In [DP17], Demertzis and Papamanthou build a family of SSE schemes with tunable locality. Their scheme is parametrized by storage efficiency s and locality L, and achieves read efficiency $\mathcal{O}(N^{1/s}/L)$. However, except in the special case where $L = N^{1/s}$, their scheme has a non-standard leakage profile, where memory accesses depend on the sizes of unqueried keywords. This is because their scheme is faced with the problem of packing lists of length varying up to some bound B, into buckets of fixed size $\geq B$. The solution adopted in [DP17], which we refer to as RandomFit, is not data-independent. This results in non-standard leakage profile, and allows for attacks that distinguish between two possible length distributions of unqueried keywords. We refer the reader to Appendix J for more details.

In hindsight, the problem faced in that part of the construction is exactly an instance of DIP. Because the notion of DIP did not exist, and building a DIP scheme is not trivial, a data-dependent solution was used. The issue can be avoided simply by replacing RandomFit with any DIP scheme, without touching the rest of the scheme.

As an example, we can replace RandomFit with TethysDIP. Doing so precludes the need for a position map. More importantly, it makes memory accesses data-independent, which avoids leaking information about the lengths of unqueried lists. Locality is doubled (but still $\mathcal{O}(L)$), since TethysDIP splits each list among two possible locations. Storage efficiency is essentially the same: 2 for RandomFit, and $2 + \varepsilon$ for TethysDIP. Regarding read efficiency, TethysDIP reads two locations instead of one, but RandomFit uses buckets that are twice as large. This amounts to the same cost. The real overhead comes from the stash. Naively, the stash incurs a factor $\omega(\log \lambda)$ in read efficiency. Thus, by inserting TethysDIP into the construction from [DP17], the trade-off achieved between storage efficiency, locality, and read efficiency, remains similar, up to a logarithmic factor in read efficiency. The point of the change is that, by using a proper DIP scheme, the security risk associated with a non-standard leakage profile disappears.

The logarithmic read efficiency overhead can be reduced using other techniques. For instance, assuming all lists in the database have size bounded by $\mathcal{O}(N/\lambda)$, it can be shown that a stash of $\omega(1)$ pages suffices. In that case, using TethysDIP in place of RandomFit recovers the exact same tradeoff as the original result from [DP17], up to a $\omega(1)$ factor, while offering a standard leakage profile. Other trade-offs are certainly possible, but are out of scope for this article. An interesting open question related to this application is the creation of an efficient DIP scheme with no stash.

K.2   Length-Hiding ORAM with Constant Storage Overhead

When accessing memory outsourced to a distant server, ORAM enables hiding the *access pattern*. That is, a honest-but-curious server does not learn which memory block was accessed. However, in the case that the client stores items of various sizes, ORAM does not hide the number of memory blocks accessed by the client, and hence, the size of the item being retrieved.

In some settings, that information alone can leak significant information, see *e.g.* [CGPR15]. A simple solution is to pad all items to the size of the largest item. This incurs an overhead in bandwidth, which is inherent: if the bandwidth consumed to retrieve an item was lower for smaller items, information about item sizes would be leaked to the server.

Naively, padding also incurs an overhead in server storage, since the padded dataset may be much larger than the original dataset. This will be the case, for example, when most items are much smaller than the largest item. That cost is *not* inherent, and can be avoided in the static setting, as follows.

Let B denote the size of the largest item in the dataset. Run a standard bin-packing algorithm on the items, with buckets of size B. Record in a separate position map where each item was stored. Initialize an arbitrary ORAM with block size B, and store the *buckets* output by the bin-packing algorithm in the ORAM. Initialize another ORAM to store the bin-packing position map. The client can retrieve an item by first querying the position map in the second ORAM, then the appropriate block in the first ORAM. The downside is that this requires two *dependent* ORAM accesses: the client needs to know the answer to the first ORAM query, before proceeding with the second query.

Using TethysDIP in place of a standard bin-packing algorithm avoids this dependency. As earlier, run TethysDIP on the dataset with buckets of size B. Each item with identifier id is stored in bucket $H_1(\mathsf{id})$, bucket $H_2(\mathsf{id})$ [3]. Then initialize an arbitrary ORAM with block size B, and store the buckets output by TethysDIP in the ORAM. In fact, since the ranges of H_1 and H_2 are disjoint, for efficiency, two separate ORAMs can be used. To lookup item id, the client simply queries the two ORAMs on blocks $H_1(\mathsf{id})$ and $H_2(\mathsf{id})$ respectively.

Compared to the previous solution, we have to make two independent accesses to smaller ORAMs (since the range of H_1 is only $1 + \epsilon/2$ larger than the original dataset, whereas standard bin-packing incurs a larger overhead). Latency is also significantly reduced, since the two accesses can be made concurrently. Remark that in this particular application, the benefits of TethysDIP do not only come from data independence (which avoids the need for a position map), but also from the fact that as a bin-packing algorithm, TethysDIP inherits the same benefits that cuckoo hashing enjoys among hashing algorithms: namely, in this case, the fact that items can be retrieved within a constant number of independent lookups.

---

[3]In this particular setting, a stash is not needed, because the adversary does not have access to the hash functions H_1 and H_2, since their outputs are hidden behind ORAM accesses. Hence, in case of failure, we can rerun the build algorithm with newly drawn hash functions. Thus, a constant probability of success is enough. The proof of Theorem 5 can be adapted, using a bound for the probability of failure of cuckoo-hashing without a stash, in order to show that TethysDIP has a constant probability of success in that setting. We omit the details.
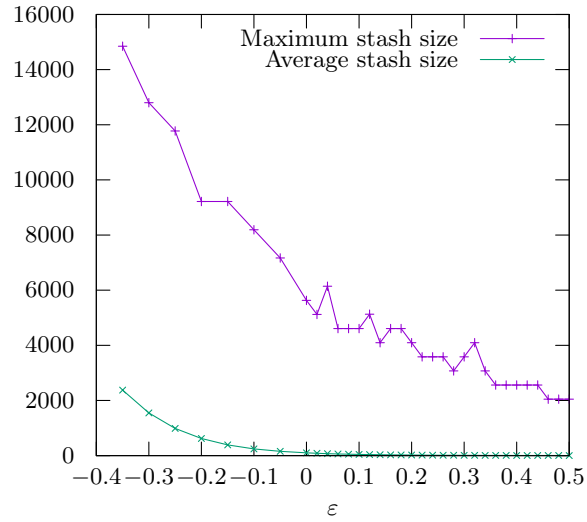
Figure 4 – Stash size function of ε. We chose $n = 2^{18}$ for $\varepsilon = 0$ and $p = 512$ (and hence $m = 2^{10}$). Each point of the curve represents respectively the mean and the maximum stash size over 6×10^6 experiments.
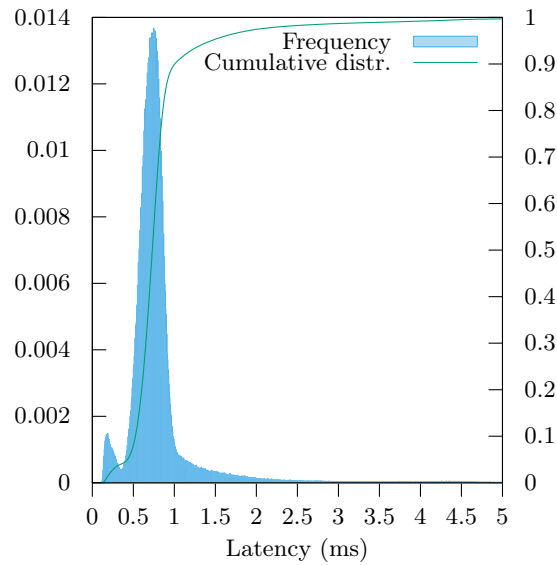

Figure 5 – End-to-end latency distribution of Tethys (including decryption and decoding of the results). The mean latency is $0.8200\,\mathrm{ms}$, while the Q1, median and Q3 latencies are respectively, $0.6313\,\mathrm{ms}$, $0.7338\,\mathrm{ms}$ and $0.8336\,\mathrm{ms}$.