Hermes: I/O-Efficient Forward-Secure Searchable Symmetric Encryption

Brice Minaud, Michael Reichle

Hermes: I/O-Efficient Forward-Secure Searchable Symmetric Encryption

Brice Minaud[1] and Michael Reichle[2]

[1] École Normale Supérieure, PSL University, CNRS, Inria, Paris, France.
[2] ETH Zürich, Switzerland.**

Abstract. Dynamic Symmetric Searchable Encryption (SSE) enables a user to outsource the storage of an encrypted database to an untrusted server, while retaining the ability to privately search and update the outsourced database. The performance bottleneck of SSE schemes typically comes from their I/O efficiency. Over the last decade, a line of work has substantially improved that bottleneck. However, all existing I/O-efficient SSE schemes have a common limitation: they are not forward-secure. Since the seminal work of Bost at CCS 2016, forward security has become a de facto standard in SSE. In the same article, Bost conjectures that forward security and I/O efficiency are incompatible. This explains the current status quo, where users are forced to make a difficult choice between security and efficiency.

The central contribution of this paper it to show that, contrary to what the status quo suggests, forward security and I/O efficiency can be realized simultaneously. This result is enabled by two new key techniques. First, we make use of a controlled amount of client buffering, combined with a deterministic update schedule. Second, we introduce the notion of SSE supporting *dummy updates*. In combination, those two techniques offer a new path to realizing forward security, which is compatible with I/O efficiency. Our new SSE scheme, Hermes, achieves sublogarithmic I/O efficiency $\widetilde{\mathcal{O}}\left(\log\log\frac{N}{p}\right)$, storage efficiency $\mathcal{O}(1)$, with standard leakage, as well as backward and forward security. Practical experiments confirm that Hermes achieves excellent performance.

1 Introduction

Encrypted databases are an attractive proposition. A business or hospital may want to outsource its customer database for higher availability, scalability, or persistence, without entrusting plaintext data to an external service. An end-to-end encrypted messaging service may want to store and search user messages, without decrypting them. In a different direction, even if a sensitive database is stored locally, a company may want to keep it encrypted to provide a layer of protection against security breaches and data theft. The adoption by MongoDB of searchable encryption techniques is another recent illustration of the growing demand for encrypted databases [31].

When outsourcing the storage of an encrypted database, a minimal desirable functionality is the ability to search the data. Powerful techniques such as Fully Homomorphic Encryption and Multi-Party Computation allow arbitrary computation on encrypted data, but those approaches incur large overheads, and become prohibitive at scale. At the other end of the spectrum, efficient historical approaches based on structure-preserving or order-preserving encryption are subject to severe attacks, due to the large amount of information leaked to the server [32, 24]. In view of this situation, modern research on searchable encryption seeks to offer workable trade-offs between performance, functionality, and security, suited for real-world deployment.

Searchable Symmetric Encryption (SSE). The promise of SSE is to allow a client to outsource an encrypted database to an untrusted server, while retaining the ability to search the data [35]. At minimum, the client is able to issue a search query to retrieve all document identifiers that match a given keyword. In the case of *Dynamic* SSE, the client is also able to modify the contents of the database by issuing update queries, for example to insert or remove entries. The server must be able

** This work was carried out while the second author was employed by Inria, Paris. The project was supported by ANR project SaFED, ANR-19-CE39-0012.

to correctly process the queries, while learning as little information as possible about the client's data and queries. The security proof accompanying an SSE scheme provides formal guarantess regarding what information is leaked to the server during searches and updates. Typically, this information includes the total size of the database, the repetition of queries, and an identifier (such as the memory address) of the documents that match a query.

Forward Security. In short, forward security asks that updates should leak no information to the server [36]. The first efficient forward-secure SSE was proposed by Bost at CCS 2016 [9]. Since then, forward security has become a de facto standard in SSE [34, 11, 27, 21, 18]. One motivation for forward security cited in [9] is that it mitigates certain attacks: the more severe attacks from [38] exploit update leakage, and fail on forward-secure schemes. Forward security is also attractive in update-heavy workloads, for instance a private messaging app. In that setting, the encrypted database is initially empty, and its entire contents are added online through updates. Forward security guarantees that building the database online leaks no information.

I/O Efficiency. Another important design goal for SSE that has emerged in recent years is I/O efficiency [15]. For performance reasons, most SSE designs rely exclusively on symmetric cryptographic primitives. The overhead of symmetric encryption is very small on modern hardware. As a result, the main performance bottleneck is instead determined by how quickly the data can be accessed on disk [14, 9, 8]. In a nutshell, I/O efficiency asks that lists of identifiers matching the same query should be stored close together in memory, at least at the page level. This is because reading many disjoint locations on disk is much more expensive in latency and throughput than contiguous reads.

The need to store related data in close proximity is not at all innocuous for security. Asking that related data items should be stored in close proximity creates a correlation between the *location* of an encrypted data item in memory, and its *contents*. Since the server can observe the location of data it is asked to retrieve, and we do not want the server to infer information about the contents of that data, this creates a tension between security and efficiency. That is, security asks that there is no correlation between the location of data and its content, while I/O efficiency asks for the opposite.

This tension was captured in an impossibility result by Cash and Tessaro at Eurocrypt 2014 [15]. To measure I/O efficiency, [15] introduces notions of *locality* (number of non-adjacent memory locations accessed per query) and *read efficiency* (ratio between the total amount of memory read by the server to process a query, and the size of the plaintext answer). In brief, Cash and Tessaro show that a secure SSE scheme with linear server storage cannot have both constant locality, and constant read efficiency. This holds true even for static SSE. In another seminal work, at STOC 2016, Asharov *et al.* build an SSE with constant locality and $\widetilde{\mathcal{O}}(\log N)$ read efficiency — even $\widetilde{\mathcal{O}}(\log \log N)$ with a mild restriction on the input database [3].

A different measure of I/O efficiency, called page efficiency, was introduced by Bossuat *et al.* at Crypto 2021 [8] (the same metric was implicit in prior work [29]). Page efficiency is the ratio of memory pages read by the server to process a query, divided by the number of pages necessary to hold the plaintext answer. From a practical standpoint, page efficiency is a very good predictor of performance on modern Solid State Drives (SSDs), whereas locality is mainly relevant for older Hard Disk Drives [8]. From a theoretical standpoint, constant page efficiency is a weaker requirement than the combination and constant locality and constant read efficiency. Interestingly, this weaker requirement sidesteps the impossibility result of Cash and Tessaro: Bossuat *et al.* build a scheme with linear server storage and constant page efficiency [8].

Since the work of Cash and Tessaro [15], many I/O-efficient schemes have been proposed. Of these, the vast majority hold in the *static* setting [15, 3, 4, 20, 19, 8], where the database is fixed at setup, and does not support updates. To our knowledge, only two constructions have been proposed in the *dynamic* setting, where the database can be updated. The first was presented by Miers and Mohassel at NDSS 2017 under the name IO-DSSE [29]. The second is a recent result presented by Minaud and Reichle at Crypto 2022 [30]. IO-DSSE has non-standard leakage, and incurs a significant performance overhead due to its reliance on an ORAM approach. Further, the security proof is flawed and we give a concrete attack in this work. On the other hand, the main

Table 1 – An overview of relevant dynamic SSE schemes. N is an upper bound on database size, W is an upper bound on the number of keywords, p is the page size.

SSE	Page Efficiency	Storage Efficiency	Client Storage	Forward Sec.
$\Sigma o\phi o\varsigma$, Diana [9, 11]	$\mathcal{O}(p)$	$\mathcal{O}(1)$	$\mathcal{O}(W)$	✓
Π_{pack}, Π_{2lev} [13]	$\mathcal{O}(1)$	$\mathcal{O}(p)$	$\mathcal{O}(W)$	✗
IO-DSSE [29]	$\mathcal{O}(\log W)$	$\mathcal{O}\left(1+\frac{p\cdot W}{N}\right)$	$\mathcal{O}(W)$	✗
LayeredSSE [30]	$\widetilde{\mathcal{O}}\left(\log\log\frac{N}{p}\right)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	✗
Hermes	$\widetilde{\mathcal{O}}\left(\log\log\frac{N}{p}\right)$	$\mathcal{O}(1)$	$\mathcal{O}(W)$	✓

construction of Minaud and Reichle is a theoretical feasibility result, with no instantiation. Neither scheme is forward-secure.

1.1 Our Contributions

Both SSE design goals, I/O efficiency and forward security, date back to 2014 [15, 36]. Almost a decade later, there is no satisfactory solution to achieving both at once. This is not a coincidence: the two goals seem to be fundamentally at odds with each other, and it was even conjectured in [9] that both notions are incompatible. While [9] argues only about locality for SSE (page efficiency did not exist at the time), the same argument directly translates to page efficiency. To sum up the argument: a correlation between the identifiers $\mathsf{DB}(w)$ matching keyword w and a newly inserted identifier also matching w breaks forward security. On the other hand, identifiers matching the same keyword w need to be stored in close proximity to satisfy page efficiency. Note that accessing correlated pages still breaks forward security. Consequently, an identifier cannot be written to the server, unless $\mathsf{DB}(w)$ is partially rewritten for each access, in an ORAM-like manner.

This state of affairs raises some troubling questions for SSE. Since I/O efficiency is the main performance bottleneck, if it is mutually exclusive with forward security, then forward security comes at a heavy performance price — hinting at a stronger form of Cash and Tessaro's impossibility result when forward security comes into play. It would also imply that the rich literature on I/O-efficient SSE will have to remain confined to static SSE, or at least non-forward-secure SSE. So far, the conjecture is supported by current work: [9] conjectures that I/O efficiency and forward security are "irreconcilable notions", except via expensive constructions such as ORAM; [29] builds the only dynamic I/O-efficient SSE scheme to date with low update leakage (although still not quite forward-secure), but relies on ORAM; and the only other dynamic I/O-efficient SSE in [30] justifies its high update leakage by stating that forward security is likely expensive for I/O-efficient SSE.

As a first contribution, we give an explicit attack that contradicts the security claim of IO-DSSE [29] (cf. Appendix A). Our attack leverages a subtle update leakage in IO-DSSE, which was introduced through an optimization of the underlying ORAM. This would seem to further support the conjecture of [9].

As our main contribution, we go against the prevailing wisdom, and show that the previous conjecture is incorrect. We build the first *forward-secure* SSE scheme, Hermes, with linear server storage and sublogarithmic page efficiency $\widetilde{\mathcal{O}}(\log\log(N/p))$. Notably, Hermes is the first I/O-efficient SSE scheme with forward security. We note that this holds with regard to page efficiency: the question of building a similar scheme with respect to locality remains an intriguing open question. A brief comparison with existing schemes is given in Table 1. Hermes does not rely on ORAM, or any ORAM-like structure. Instead, it makes use of two new technical ideas.

First, we introduce the notion of SSE schemes supporting *dummy updates*. A dummy update can be triggered by the client at any time, and must look indistinguishable from a real update in the server's view. On the other hand, we require that server storage should only grow with an upper bound N on the number of *real* entries in the database: dummy updates create no storage overhead. We present a simple framework to build an SSE scheme $\mathsf{Dummy}(\Sigma)$ supporting dummy

updates, based on an underlying suitable SSE scheme Σ. The framework is based on an application of the two-choice allocation process.

The second main technical idea is a form of "deamortized" trivial ORAM. An explanation of this technique is deferred to the technical overview below. For now, we note that it involves buffering $\mathcal{O}(W)$ updates on the client side before pushing them to the server, where W is an upper bound on the number of searchable keywords. It was proved in [10] that single-round forward-secure SSE requires $\Omega(W)$ client storage. As a consequence, the new buffer does not increase client storage beyond a constant factor[3]. Interestingly, it is is the use of client storage that circumvents the proof sketched in [9] for the incompatibility of forward security and I/O efficiency.

In Section 7, we run experiments to choose concrete parameters for Hermes. Although our main contribution is qualitative (showing that forward security and I/O efficiency are compatible), practical experiments show that Hermes performs very well in practice, being outpaced only by purely static schemes.

1.2 Related Work

Several research directions are active within Searchable Encryption. Here, we focus on I/O efficiency. A brief overview of other directions is given in Appendix B.

The importance of I/O efficiency, and the relevant literature, were presented in the introduction. We do not repeat them here. We simply recall that most prior work on I/O-efficient SSE is limited to *static* SSE [15, 3, 4, 20, 19, 8], where the database is known in advance and immutable. We are interested in *dynamic* I/O-efficient SSE, which allows for updates to the database. Despite the obvious practical importance of mutable databases, there is still very little work in that area — likely owing to both recency and technical difficulty. We are aware of only two constructions of dynamic I/O-efficient SSE. Since they are directly comparable with this work, they deserve special attention.

The first construction of I/O-efficient Dynamic SSE was presented by Miers and Mohassel at NDSS 2017, under the name IO-DSSE [29]. IO-DSSE opened the way in a new area, but suffers from significant limitations. Intuitively, I/O efficiency requires that document identifiers that match the same keyword should be stored in close proximity. For that purpose, IO-DSSE groups identifiers matching the same keyword into *blocks* of a fixed size p. Since the number of documents matching a given keyword need not be a multiple of p, it is usually the case that one of the blocks is *incomplete*, *i.e.* it contains less than p identifiers. As in other works on I/O-efficient SSE, the main technical issue is how to efficiently handle incomplete blocks. Especially, when a new document matching a given keyword is added to the database, the document identifier needs to be appended to the incomplete block associated with the keyword. In that process, it is unclear how to hide to the server which incomplete block is being modified.

The solution proposed by IO-DSSE is to store incomplete blocks in an optimized Oblivious RAM (ORAM) construction. ORAM is a generic solution to hide memory access patterns, but is notoriously expensive in practice. In IO-DSSE, this approach is viable, because IO-DSSE focuses on use cases with few searchable keywords: they target a messaging app scenario, and assume in their experiments that no more than 350 keywords are searchable. Because the number of keywords is small, and there can be at most one incomplete block per keyword, the ORAM overhead remains manageable. By contrast, if we were to run IO-DSSE on the English Wikipedia database (a classic target in SSE literature), which contains millions of keywords, IO-DSSE blows up the size of the database by a factor more than 100 (Section 7). A second limitation of IO-DSSE is that its security claim is incorrect. It does not appear that the issue can be fixed without a significant penalty in performance (cf. Appendix A). A third limitation of IO-DSSE is that, independently of the previous security issue, IO-DSSE is not forward-secure. This puts it at odds with most of the rest of modern (non-I/O-efficient) SSE literature, where forward security has become a standard requirement [9, 34, 11, 27, 21, 18].

A recent work by Minaud and Reichle from Crypto 2022 also targets dynamic I/O-efficient SSE [30]. Like IO-DSSE, [30] introduces useful techniques to build I/O-efficient SSE. However,

[3] Although $\mathcal{O}(W)$ client storage is inherent to sublogarithmic forward-secure SSE as just noted, in practice, this cost may be too high for some applications. Practical tradeoffs to reduce client storage are discussed in Appendix E.3.

it is a theoretical work, which only considers asymptotic performance. No practical evaluation is offered. Moreover, none of the constructions is forward-secure. In practice, all the constructions of [30] directly leak which keyword is being updated, which is the worst case with regard to certain file-injection attacks [38]. In contrast, Hermes is efficient in practice, and achieves forward security in the strongest sense: updates leak no information to the server.

Before closing the section, we note that dynamic I/O-efficient SSE could in principle be built using a folklore hierarchical construction, which generically builds dynamic SSE from a static SSE scheme. That construction was sketched in [36, 20], and studied in more detail in [18]. Following that approach, one could theoretically build dynamic I/O-efficient SSE by using a static I/O-efficient SSE as underlying static scheme. However, this quickly proves impractical. First, the approach inherently incurs a logarithmic factor in both locality and page efficiency, on top of the I/O cost of the underlying static SSE. As a result, it cannot hope to match the page efficiency of Hermes, which is sublogarithmic. The most natural candidate for the underlying static SSE is the Tethys scheme from [8]: it is the only one to achieve constant page efficiency, hence the only one that would not further deteriorate page efficiency beyond the log factor inherent to the approach. However, Tethys has a quadratic $\mathcal{O}\left(N^2\right)$ setup time. This is problematic, because every N insertions, the generic construction requires building a fresh static SSE instance of size N. This implies that the average computational cost of *one* update would be $\mathcal{O}\left(N^2/N\right) = \mathcal{O}(N)$. Last but not least, the hierarchical approach requires periodically rebuilding a static SSE scheme of size N. Generically, this implies storing the *entire* database on the client side during the rebuilding phase.

1.3 Technical Overview

I/O efficiency and forward security are two important goals of SSE research, but seemingly incompatible. On an intuitive level, this is because I/O efficiency requires that identifiers matching the same keyword should be stored close to each other, so that they can be read together efficiently when the keyword is queried. Forward security requires that when a *new* identifier matching some keyword is added, it cannot be stored close to previous identifiers for the same keyword, since that would leak information about the new identifier to the server. One way to resolve this apparent contradiction is to use ORAM, as was suggested in [9], and later realized in [29]. We start by sketching the ORAM approach, which will serve to explain what Hermes does differently.

A natural way to build page-efficient SSE is to maintain an array of bins of capacity one page each, one bin for each keyword. When the client wishes to insert a new document identifier matching some keyword w, the new document identifier is added to the bin associated with w. Once the bin for keyword w is full, *i.e.* it contains a full page of identifiers matching w, the page of identifiers is inserted into a separate SSE scheme Σ that only contains full pages, and the bin is emptied. Because Σ only contains *full* pages, each page can be stored in an arbitrary location, and the scheme remains page-efficient; hence page efficiency is easy to realize. In order to search for the list of document identifiers that match keyword w, the client needs only to fetch the bin associated with w, and query Σ on w.

The problem with this naive approach is that it is not forward-secure: during an update, the server can see which bin is accessed, hence which keyword is being updated. A generic way to circumvent this leakage is to store the bins in an ORAM, which completely hides access patterns. (Roughly speaking, this approach is the one of IO-DSSE.) Despite the use of ORAM, the approach still has two issues. First, it is *still* not forward-secure: when a page becomes full, the server can observe that a new element is inserted into Σ, hence upates are not leakage-free (this leakage suffices to break the security game of forward-secure SSE). Second, ORAM incurs an $\Omega(\log W)$ overhead, where W is the number of searchable keywords, and is costly in practice.

Challenge 1: achieving forward security. To avoid leaking when a bin becomes full, we could insert a new item in Σ for every client update, regardless of whether the bin is actually full. A bin becomes full after it receives p client updates, where p is the page size (counted in number of memory words, identified here with the size of a document identifier). Asymptotically, we would have storage efficency $\mathcal{O}(p)$ instead of the desired $\mathcal{O}(1)$. In practice, for 64-bit identifiers and 4kB memory pages, $p = 512$: blowing the size of Σ by a factor p is not acceptable. Instead, we introduce the idea of SSE supporting dummy updates. When a bin is full, the full page is inserted into Σ; when it is not full, the client issues a dummy update to Σ. The promise of dummy updates is

that they should be indistinguishable from real updates in the server's view. At the same time, we arrange that they cost nothing in storage: in our actual construction dummy updates do not alter the contents of the encrypted database.

Challenge 2: realizing dummy updates. At a high level, building SSE with dummy updates comes down to building a key-value store that is amenable to fake key queries. For that purpose, we use the two-choice allocation process. In a two-choice process, n values are stored in m bins of fixed capacity c. Each value for key k is stored in the bin $H_1(k)$ or $H_2(k)$, where H_1, H_2 are hash functions mapping into $[1, m]$. We pad the bins to their full capacity c, and encrypt them with an IND-CPA scheme. Intuitively, simulating a dummy query is simple: the client fetches two uniformly random bins, re-encrypts them, and re-uploads them to the server. Setting $c = \widetilde{\mathcal{O}}(\log \log n)$, $m = \widetilde{\mathcal{O}}(n/\log \log n)$ suffices to ensure a negligible probability of overflow, with constant storage efficiency. This idea can be adapted to the SSE setting. We realize it as a framework that builds a scheme $\mathsf{Dummy}(\Sigma)$ supporting dummy updates, based on an underlying suitable forward-secure scheme Σ.

Challenge 3: dispensing with ORAM. Recall that our scheme uses W bins, each of capacity one page. As noted in the introduction, single-round forward-secure SSE requires $\Omega(W)$ client storage [10]. If we buffer W updates on the client before pushing them to the server, we can afford to scan all W bins. This costs W page accesses per W updates, hence $\mathcal{O}(1)$ *amortized* page efficiency. Another way to view this process is that we are performing a trivial ORAM (*i.e.* reading the entire array of bins), amortized over W updates. This basic idea can be deamortized, in such a way that each client update generates $\mathcal{O}(1)$ page accesses in the worst case. The deamortization is somewhat subtle, and proceeds differently depending on the regime of global parameters N, W and p. For that reason, we introduce two schemes, $\mathsf{BigHermes}$ and $\mathsf{SmallHermes}$, respectively for the case $N \geq pW$ and $N \leq pW$. Hermes is the combination of those two schemes, one for each regime.

In the end, the storage and page efficiency of Hermes reduce to those of the underlying scheme supporting dummy updates. Because that scheme relies on the two-choice process, this works out to $\widetilde{\mathcal{O}}(\log \log(N/p))$ page efficiency, and constant storage efficiency. While the ideas of buffering and deamortization, as well as dummy updates, may be natural in hindsight, we view them as the key contributions of this work: to our knowledge, they realize forward security in a way that is fundamentally different from how it was realized in prior works — and one that happens to be compatible with I/O efficiency. We note that both new techniques (the use of dummy updates, for forward security; and the replacement of standard ORAM with "deamortized" trivial ORAM, for efficiency), are modular: we could have introduced intermediate schemes that realize one without the other, although we did not see a compelling reason for it.

2 Preliminaries

2.1 Notation

Let $\lambda \in \mathbb{N}$ be the security parameter. If X is a probability distribution, $x \leftarrow X$ means that x is sampled from X. If $\mathcal{X}$ is a set, $x \leftarrow \mathcal{X}$ means that x is sampled uniformly at random from $\mathcal{X}$. A function $f(\lambda)$ is *negligible* in λ if it is $\mathcal{O}(\lambda^{-c})$ for every $c \in \mathbb{N}$. We write $f = \mathsf{negl}(\lambda)$ for short.

Protocols. Let $\mathsf{prot} = (\mathsf{prot}_A, \mathsf{prot}_B)$ be a protocol between two parties A and B. We denote an execution of protocol prot between A and B with input in_A and in_B respectively by $\mathsf{prot}_\mathsf{A}(\mathsf{in}_A) \longleftrightarrow \mathsf{prot}_\mathsf{B}(\mathsf{in}_B)$. We may write $\mathsf{prot}(\mathsf{in}_A; \mathsf{in}_B)$ for short, if both executing parties can be inferred from the context.

Data Structures. For concision, in algorithmic descriptions, tables and arrays are implicitly assumed to be initialized with 0's (if they contain integers) or $\perp$'s, unless stated otherwise. Our algorithms will frequently make use of *bins*, which can be thought of as disjoint memory segments of some fixed size $s = f(p, \lambda)$. Bins can contain arbitrary data up to their capacity s. Bins are always implicitly assumed to be padded with 0's up to their full capacity, so that their size remains fixed. In particular, the encryption of a bin reveals no information about the amount of real data contained in the bin.

Cryptographic Primitives. Throughout the article, we use the following cryptographic primitives: (1) Enc is an IND-CPA secure symmetric encryption scheme (assimilated with its encryption algorithm in the notation); (2) PRF is a secure pseudo-random function; (3) H is a collision-resistant hash function (which will be modeled as a random oracle in most security statements).

2.2 Searchable Symmetric Encryption

We recall the notion of searchable symmetric encryption (SSE). A *database* $\mathsf{DB} = \{(w_i, (\mathsf{id}_1, ..., \mathsf{id}_{\ell_i}))\}_{i=1}^{K}$ is a set of K pairs $(w_i, (\mathsf{id}_1, ..., \mathsf{id}_{\ell_i}))$, where w_i is a *keyword*, and $(\mathsf{id}_1, ..., \mathsf{id}_{\ell_i})$ is a tuple of ℓ_i document identifiers matching keyword w_i. The number of distinct keywords is K. We write $\mathsf{DB}(w_i) = (\mathsf{id}_1, ..., \mathsf{id}_{\ell_i})$ for the list of identifiers matching keyword w_i. The *size* of the database DB is the number of distinct keyword-identifier pairs (w_i, id_j), with $\mathsf{id}_j \in \mathsf{DB}(w_i)$. It is equal to $\sum_{i=1}^{K} \ell_i$.

We will usually assume that there is an upper bound W on the total number of keywords: $K \leq W$; and an upper bound N on the size of the database: $\sum_{i=1}^{k} \ell_i \leq N$. Throughout the article, the integer p denotes the page size. We treat p as a variable independent of the size of the database N, in line with previous work. Our complexity analysis holds in the RAM model of computation, where accessing a random memory word costs unit time. Memory is counted in number of memory words, which are assumed to be of size $\mathcal{O}(\lambda)$ bits, as is common in the literature.

Before giving the formal definition, let us sketch how a dynamic SSE scheme operates. The client stores a small state st, while the server stores the encrypted database EDB. The client calls $\Sigma.\mathsf{Setup}$ to initialize both states, on input an initial plaintext database DB. To search the database on keyword w, the client initiates the protocol $\Sigma.\mathsf{Search}$ on input w, and eventually obtains the list of matching identifiers $\mathsf{DB}(w)$. As a side effect, $\Sigma.\mathsf{Search}$ may also update the client and server states. Similarly, to add a new keyword-identifier pair (w, id) to the encrypted database, the client initiates $\Sigma.\mathsf{Update}$ on the corresponding input. The following formal definition follows [8, 30], with minor tweaks.

A *dynamic searchable symmetric encryption* scheme Σ is a 4-tuple of PPT algorithms (KeyGen, $\mathsf{Setup}, \mathsf{Search}, \mathsf{Update}$):

- $\Sigma.\mathsf{KeyGen}(1^\lambda)$: Takes as input the security parameter λ and outputs client secret key K.
- $\Sigma.\mathsf{Setup}(\mathsf{K}, N, W, \mathsf{DB})$: Takes as input the client secret key K, upper bounds N on the database size and W on the number of keywords, and a database DB. Outputs an encrypted database EDB and client state st.
- $\Sigma.\mathsf{Search}(\mathsf{K}, w, \mathsf{st}; \mathsf{EDB})$: The client receives as input the secret key K, keyword w and state st. The server receives as input the encrypted database EDB. Outputs some data d, an updated state st' for the client, and an updated encrypted database EDB' for the server.
- $\Sigma.\mathsf{Update}(\mathsf{K}, w, \mathsf{id}, \mathsf{op}, \mathsf{st}; \mathsf{EDB})$: The client receives as input the secret key K, a keyword-identifier pair (w, id), an operation $\mathsf{op} \in \{\mathsf{del}, \mathsf{add}\}$, and state st. The server receives the encrypted database EDB. Outputs an updated state st' for the client, and an updated encrypted database EDB' for the server.

We denote by Search_C (resp. Update_C) the client side of the protocol Search (resp. Update), and by Search_S (resp. Update_S) its server counterpart. We may omit W in the input of Setup if it is not used.

For concision, in the remainder, the client state st will be omitted in the notation. As is standard in SSE literature, we assume that keywords are preprocessed via a PRF by the client. That is, $w = \mathsf{PRF}_K(\mathsf{H}(k))$ where k is the actual keyword, for some PRF key K known only to the client.

Epochs. To facilitate the description of our schemes, it is convenient to conceptually partition update queries issued by the client into sequences of W consecutive updates. The time frame corresponding to one such sequence of W updates is called an *epoch*. More precisely, the k-th epoch comprises all updates and searches that are performed between the $((k-1) \cdot W + 1)$-th and $(k \cdot W)$-th update. Looking ahead, update queries belonging to the current epoch will typically be preprocessed together on the client side, then progessively pushed to the server during the next epoch.

Correctness. Informally, correctness asks that at the outcome of a Search protocol on keyword w, the client should obtain exactly the identifiers of documents matching w. The following definition asks for perfect correctness. In some cases, we may allow correctness to fail as long as the probability of failure is negligible.

Definition 1 (Correctness). *An* SSE *scheme* Σ *is* correct *if for all sufficiently large* $W, N \in \mathbb{N}$*, for all databases* DB*, and all sequences of search and update operations, provided at most* K *keywords are used, and the size of the database remains at most* N *at all times, letting* $\mathsf{K} \leftarrow \Sigma.\mathsf{KeyGen}(1^\lambda)$ *and* $\mathsf{EDB} \leftarrow \Sigma.\mathsf{Setup}(\mathsf{K}, N, W, \mathsf{DB})$*, at the outcome of a search query on keyword* w*, the client obtains exactly the identifiers of documents matching keyword* w *at query time. (That is, documents matching* w *in the initial database* DB *or added by an update query matching* w*, and not subsequently deleted.)*

Security. We use the standard semantic security notion from [17]. The server is modeled as a honest-but-curious adversary. Intuitively, security asks that the information learned by the server in the course of the scheme's execution is no more than a specified leakage. The allowed leakage is expressed by a *leakage function*, composed of setup leakage $\mathcal{L}_{\mathsf{Stp}}$, search leakage $\mathcal{L}_{\mathsf{Srch}}$, and update leakage $\mathcal{L}_{\mathsf{Updt}}$. The intent is that, when executing Setup on input x, the server should learn no more than $\mathcal{L}_{\mathsf{Stp}}(x)$. To formally capture that requirement, the security definition asks that there exists a PPT simulator that can simulate the view of the server, taking as input only $\mathcal{L}_{\mathsf{Stp}}(x)$. The same goes for Search and Update.

Formally, we define two games, SSEReal and SSEIdeal. In both games, the adversary first chooses a database DB. In SSEReal, the encrypted database EDB is then generated by $\mathsf{Setup}(\mathsf{K}, N, \mathsf{DB})$. In SSEIdeal, EDB is instead generated by a stateful PPT algorithm Sim called the *simulator*, on input $\mathcal{L}_{\mathsf{Stp}}(\mathsf{DB}, N)$. After receiving EDB, the adversary adaptively issues search and update queries. In SSEReal, all queries are answered honestly. In SSEIdeal, search queries on keyword w are simulated by Sim on input $\mathcal{L}_{\mathsf{Srch}}(w)$, and update queries for operation op, keyword w, and identifier id are simulated by Sim on input $\mathcal{L}_{\mathsf{Updt}}(\mathsf{op}, w, \mathsf{id})$. At the end of the game, the adversary outputs a bit b.

Definition 2 (Semantic Security). *Let* Σ *be an* SSE *scheme and let* $\mathcal{L} = (\mathcal{L}_{\mathsf{Stp}}, \mathcal{L}_{\mathsf{Srch}}, \mathcal{L}_{\mathsf{Updt}})$ *be a leakage function. The scheme* Σ *is* $\mathcal{L}$-adaptively secure *if for all PPT adversaries* $\mathcal{A}$*, there exists a PPT simulator* Sim *such that:*

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

Leakage Patterns. To facilitate the description of leakage functions, we make use of the following standard notions from the litereature [9]. The *search pattern* $\mathsf{sp}(w)$ for keyword w is the sequence of identifiers of previous search queries on w. The *update pattern* $\mathsf{up}(w)$ for keyword w is the sequence of identifiers of previous update queries on w. The *query pattern* $\mathsf{qp}(w) = (\mathsf{sp}(w), \mathsf{up}(w))$ is the combination of search and update patterns.

Throughout the article, our schemes will be secure with regard to the following leakage function: $\mathcal{L}^{\mathsf{fs}} = (\mathcal{L}^{\mathsf{fs}}_{\mathsf{Stp}}, \mathcal{L}^{\mathsf{fs}}_{\mathsf{Srch}}, \mathcal{L}^{\mathsf{fs}}_{\mathsf{Updt}})$, with $\mathcal{L}^{\mathsf{fs}}_{\mathsf{Stp}}(\mathsf{DB}, W, N) = (W, N)$, $\mathcal{L}^{\mathsf{fs}}_{\mathsf{Srch}}(w_i) = (\mathsf{qp}, a_i, d_i)$ where a_i (resp d_i) is the number of additions (resp deletions) for w_i, and $\mathcal{L}^{\mathsf{fs}}_{\mathsf{Updt}}(\mathsf{op}, w, \mathsf{id}) = \perp$. In words, Setup leaks an upper bound on the size of the database and on the number of keywords; Search reveals the query pattern, and the number of additions and deletions for the searched keyword; and updates leak nothing.

Forward Security and Backward Security. Forward security was introduced and formalized in [36] and [9], respectively. In this work, we consider a strictly stronger notion of forward security where updates leak no information.

Definition 3 (Forward Security). *An SSE scheme with leakage* $\mathcal{L} = (\mathcal{L}_{\mathsf{Stp}}, \mathcal{L}_{\mathsf{Srch}}, \mathcal{L}_{\mathsf{Updt}})$ *is* forward-secure *if* $\mathcal{L}_{\mathsf{Updt}}(\mathsf{op}, w, \mathsf{id}) = \perp$.

The notion of backward security was formalized in [11], and restricts the leakage incurred by deletions. Since backward security is not the main focus of this work, we refer the reader to [11] for the formal definition. In this work we consider type-II backwards security, which requires that search queries leak the documents currently matching w, when they were inserted, and when all the updates on w happened (but not their content). Note that schemes with leakage $\mathcal{L}^{\mathsf{fs}}$ are forward-secure, and type-II backward-secure.

Remark on Deletions. The concrete SSE constructions presented in this article are involved. Thus, we present them without deletions to improve readability. This however does not reduce their functionality, as all schemes can be extended to support deletions with the generic framework of [9]. Intuitively, given an SSE scheme Σ_{add} that supports additions, it allows to construct a scheme Σ that supports additions and deletions with the same leakage. For this, two instantiations of Σ_{add} are used, Σ_0 for additions and Σ_1 for deletions. Added identifiers are inserted into Σ_0 while deleted keyword-identifier pairs are inserted into Σ_1. Each search request, the client queries Σ_0 and Σ_1 and retrieves identifier lists L_0 and L_1 respectively. The result of the search request then is $L_0 \setminus L_1$. Clearly, if Σ_{add} has leakage $\mathcal{L}^{\mathsf{fs}}$, then Σ also has leakage $\mathcal{L}^{\mathsf{fs}}$.

Efficiency Measures. We recall the notions of storage efficiency [15], and page efficiency [8]. To reduce clutter, the following notation is common to all definitions. Let $\mathsf{K} \leftarrow \mathsf{KeyGen}(1^\lambda)$. Given a database DB, an upper bound W on the number of keywords, and an upper bound N on the size of the database, let $\mathsf{EDB} \leftarrow \mathsf{Setup}(\mathsf{K}, N, \mathsf{DB})$. Let $S = (\mathsf{op}_i, \mathsf{in}_i)_{i=1}^s$ be a sequence of search and update queries, where $\mathsf{op}_i \in \{\mathsf{add}, \mathsf{del}, \mathsf{srch}\}$ is an operation and $\mathsf{in}_i = (\mathsf{op}_i, w_i, \mathsf{id}_i, \mathsf{st}_i; \mathsf{EDB}_i)$ its input. (If $\mathsf{op}_i = \mathsf{srch}$, the query is a search query, and id_i is not provided.) After executing all operations op_j in S up to $j \leq i$, let DB_i denote the state of the database, st_i the client state, and EDB_i the encrypted database. The following definitions are borrowed from [30].

Definition 4 (Storage Efficiency). *An SSE scheme has storage efficiency E if for any λ, DB, N, sequence S, and any i, $|\mathsf{EDB}_i| \leq E \cdot |DB_i|$.*

Definition 5 (Page Pattern). *Regard server-side storage as an array of* pages*, containing the encrypted database* EDB*. When processing search query* $\mathsf{Search}(\mathsf{in}_i)$ *or update query* $\mathsf{Update}(\mathsf{in}_i)$*, the server accesses a number of pages $p_1, \ldots, p_{h'}$. We call these pages the* page pattern*, denoted by* $\mathsf{PgPat}(\mathsf{op}_i, \mathsf{in}_i)$.

Definition 6 (Page Efficiency). *An SSE scheme has page efficiency P if for any λ, DB, N, sequence S, and any i, $|\mathsf{PgPat}(\mathsf{op}_i, \mathsf{in}_i)| \leq P \cdot X$, where X is the number of pages needed to store the document identifiers $\mathsf{DB}_i(w_i)$ matching keyword w_i in plaintext.*

2.3 Allocation Schemes

Two-Choice. Insert n balls into m bins according to the standard two-choice (2C) process [6]. That is, to insert each ball, pick two bins B_α, B_β uniformly at random, and insert the ball into the least loaded bin $B_\gamma \in \{B_\alpha, B_\beta\}$.

Lemma 1 (2C [30]). *Let $\delta(m) = \log\log\log m$ and $m \geq \lambda^{1/\log\log\lambda}$. At the outcome of a sequence of n insertions, the most loaded bin contains $\mathcal{O}(n/m + \delta(m)\log\log m)$ balls, except with negligible probability.*

Layered Two-Choice. Layered two-choice (L2C) is a weighted variant of the two-choice process, introduced in [30]. We sketch a simplified version that suffices for our purpose. Let $p \in \mathbb{N}$. Consider a sequence of insertions of *weighted* balls into m bins. Each weight is an integer in $[1, p]$ and each bin is split into $1 + \log\log m$ independent layers, numbered from 0 to $\log\log m$. (For simplicity, assume $\log\log m$ is an integer.) A ball of weight w is stored in layer 0 if $w \in [0, p/\log m]$, and in layer $i \geq 1$ if $w \in (p2^{i-1}/\log m, p2^i/\log m]$. An insertion of a ball of weight w is similar to a two-choice insertion: pick two bins B_α, B_β uniformly at random, and insert the ball into the bin $B_\gamma \in \{B_\alpha, B_\beta\}$ that contains the fewest balls at layer i. In other words, the difference with

the standard two-choice process is that when determining the "least loaded bin", we only look at the number of balls in the same layer as the newly inserted ball, rather than looking at the total number.

The weight of balls can be increased after insertion. Assume the weight w of a ball is increased by v, *i.e.* the ball has weight $w + v$ after the update. The layer of the ball after the update may increase as a result. Let i be the old layer and j be the new layer after the update. If $i \neq j$, the ball of weight $w + v$ is reinserted into layer j. The old ball of weight w at layer i is marked as *residual*, and is still considered for the load computation of its bin.

Define the *load* of a bin to be the sum of the weights of the balls it contains (including residual balls, as noted above). Define the *total weight* to be the sum of the weights of all balls inserted so far, using their current weights if it has been updated. We require that the total weight remains upper-bounded by $w_{\mathsf{max}} = \mathsf{poly}(\lambda)$.

Lemma 2 ([30]). *Let* $\delta(\lambda) = \log\log\log\lambda$, $m = \lceil \frac{w_{\mathsf{max}}}{\delta(\lambda)\log\log w_{\mathsf{max}}} \rceil$, *and assume* $m = \Omega(\lambda^{\frac{1}{\log\log\lambda}})$. *After a sequence of insertions and updates, the load of the most loaded bin is* $\mathcal{O}(p\delta(\lambda)\log\log w_{\mathsf{max}})$, *except with negligible probability.*

In this article, the role of balls will be played by lists of (at most p) identifiers. The weight of a list L is its number of identifiers. When some identifiers I are added to L, if the new weight $|L \cup I|$ is in the same layer as $|L|$, then the new identifiers I are simply added to L, in the same bin. Otherwise, I is inserted in one of the two chosen bins, as if it was a list of weight $|L \cup I|$. (In that case, the list $L \cup I$ is split between both bins.)

3 SSE with Dummy Updates

A first key technique for our results is the introduction of the notion of *dummy updates*. An SSE schemes supports dummy updates if its interface is equipped with a new operation $\mathsf{DummyUpdate}$, taking as input only the client's master key K. For technical reasons, we also require that Setup receives an additional parameter D, an upper bound on the total number of dummy updates.

3.1 Security Definition

Informally, an SSE scheme supporting dummy updates is said to be secure if it is secure in the same sense as a normal SSE scheme, with the added requirement that dummy updates should be indistinguishable from real updates from the server's perspective. (A subtle but important point is that later constructions will ensure that server storage does not depend on D: dummy updates will look indistinguishable from real updates, without actually affecting server storage.)

The security definition for SSE supporting dummy updates is given in Definition 7, with the associated security game in Algorithm 1. Note that this definition naturally extends the standard definition (Definition 2).

Definition 7 (Adaptive Semantic Security with Dummy Updates). *Let* Σ *be an SSE scheme supporting dummy updates,* $\mathcal{A}$ *a stateful PPT adversary, and* Sim *a stateful PPT simulator. Let* $q \in \mathbb{N}$, *and let* $\mathcal{L} = (\mathcal{L}_{\mathsf{Stp}}, \mathcal{L}_{\mathsf{Srch}}, \mathcal{L}_{\mathsf{Updt}})$ *be a leakage function. The games* $\text{SSEREAL}^{\mathsf{dum}}_{\Sigma,\mathcal{A}}$ *and* $\text{SSEIDEAL}^{\mathsf{dum}}_{\Sigma,\mathcal{A},\mathcal{L},\mathcal{A}}$ *are defined in Algorithm 1.* Σ *is* $\mathcal{L}$-adaptively secure with support for dummy updates *if* $\mathcal{L}_{\mathsf{Updt}}$ *does not depend on its first input* op, *and if for all PPT adversaries* $\mathcal{A}$, *there exists a PPT simulator* Sim *such that:*

$$|\Pr[\text{SSEREAL}^{\mathsf{dum}}_{\Sigma,\mathcal{A}}(\lambda) = 1] - \Pr[\text{SSEIDEAL}^{\mathsf{dum}}_{\Sigma,\mathsf{Sim},\mathcal{L},\mathcal{A}}(\lambda) = 1]| = \mathsf{negl}(\lambda)\,.$$

3.2 A Framework to Build SSE with Dummy Updates

We now describe a framework that constructs an SSE scheme $\mathsf{Dummy}(\Sigma)$ supporting dummy updates, based on an underlying SSE scheme Σ. Provided Σ is *suitable* in a sense that will be defined shortly, the resulting scheme $\mathsf{Dummy}(\Sigma)$ achieves the following features:

Algorithm 1 Security games for SSE supporting dummy updates.

$\text{SSEReal}^{\text{dum}}_{\Sigma,\mathcal{A}}$

$\mathsf{K} \leftarrow \Sigma.\mathsf{KeyGen}(1^\lambda)$
$(\mathsf{DB}, N, D, W, \mathsf{st}_\mathcal{A}) \leftarrow \mathcal{A}(1^\lambda)$
$\mathsf{EDB} \leftarrow \Sigma.\mathsf{Setup}(\mathsf{K}, N, D, W, \mathsf{DB})$
send EDB to $\mathcal{A}$
for all $1 \le i \le q$ **do**
  $(\mathsf{op}_i, \mathsf{in}_i, \mathsf{st}_\mathcal{A}) \leftarrow \mathcal{A}(\mathsf{st}_\mathcal{A})$
  if $\mathsf{op}_i = \mathsf{srch}$ **then**
    Parse $\mathsf{in}_i = w_i$
    $\Sigma.\mathsf{Search}_C(\mathsf{K}, w_i) \leftrightarrow \mathcal{A}(\mathsf{st}_\mathcal{A})$
  else if $\mathsf{op}_i \in \{\mathsf{add}, \mathsf{del}\}$ **then**
    Parse $\mathsf{in}_i = (w_i, \mathsf{id}_i)$
    $\Sigma.\mathsf{Update}_C(\mathsf{K}, w_i, \mathsf{id}_i, \mathsf{op}_i) \leftrightarrow \mathcal{A}(\mathsf{st}_\mathcal{A})$
  else
    $\Sigma.\mathsf{DummyUpdate}_C(\mathsf{K}) \leftrightarrow \mathcal{A}(\mathsf{st}_\mathcal{A})$
output bit $b \leftarrow \mathcal{A}(\mathsf{st}_\mathcal{A})$

$\text{SSEIdeal}^{\text{dum}}_{\Sigma,\mathsf{Sim},\mathcal{L},\mathcal{A}}$

$(\mathsf{DB}, N, D, W, \mathsf{st}_\mathcal{A}) \leftarrow \mathcal{A}(1^\lambda)$
$\mathsf{EDB} \leftarrow \mathsf{Sim}(\mathcal{L}_{\mathsf{Stp}}(\mathsf{DB}, N, D, W))$
send EDB to $\mathcal{A}$
for all $1 \le i \le q$ **do**
  $(\mathsf{op}_i, \mathsf{in}_i, \mathsf{st}_\mathcal{A}) \leftarrow \mathcal{A}(\mathsf{st}_\mathcal{A})$
  if $\mathsf{op}_i = \mathsf{srch}$ **then**
    $\mathsf{Sim}(\mathcal{L}_{\mathsf{Srch}}(\mathsf{in}_i)) \leftrightarrow \mathcal{A}(\mathsf{st}_\mathcal{A})$
  else
    $\mathsf{Sim}(\mathcal{L}_{\mathsf{Updt}}(\mathsf{op}_i, \mathsf{in}_i)) \leftrightarrow \mathcal{A}(\mathsf{st}_\mathcal{A})$
output bit $b \leftarrow \mathcal{A}(\mathsf{st}_\mathcal{A})$

- It is a secure SSE scheme supporting dummy updates (cf. Definition 7).
- Server storage grows only with the number of *real* updates; in particular, it does not depend on the upper bound D on the number of dummy updates.
- Relative to the base scheme Σ, $\mathsf{Dummy}(\Sigma)$ only incurs a $\widetilde{\mathcal{O}}(\log\log N)$ overhead in communication.

The definition of a *suitable* SSE is given next. Essentially, it requires that server storage should behave like a key-value store. This is how most forward-secure SSE schemes operate. It also requires that running Setup on a non-empty initial database DB is equivalent to running Setup on an empty database, then performing updates to add the contents of DB. Here, "equivalent" means that the client and server states at the outcome of either process are distributed identically. This condition can be fulfilled trivially by any SSE scheme. It is not strictly necessary, but makes the description of the framework, and its security proof, more consise.

Definition 8 (Suitable SSE). *We say that an* SSE *scheme* Σ *is* suitable *if there exist a key space* $\mathcal{K}$*, a token space* $\mathcal{T}$*, and a map* $\mathsf{keys} : \mathcal{T} \mapsto \mathcal{K}$ *such that: (1)* $\Sigma.\mathsf{Setup}(\mathsf{K}, N, W, \mathsf{DB})$ *outputs an encrypted database* EDB *in the form of an encrypted key-value store that maps a key to encrypted identifiers. (2)* $\Sigma.\mathsf{Search}(K, w; \mathsf{EDB})$ *is a two-step protocol in which the client first sends a token* $\tau \in \mathcal{T}$ *and the server responds with* $\mathsf{EDB}[k_1], \ldots, \mathsf{EDB}[k_q]$ *for* $k_1, \ldots, k_q \leftarrow \mathsf{keys}(\tau)$*. (3)* $\Sigma.\mathsf{Update}(K, (w, \mathsf{id}), \mathsf{op}; \mathsf{EDB})$ *is a one-step protocol in which the client sends a key* $k \in \mathcal{K}$ *and value* $v = \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(\mathsf{id})$ *and the server stores* v *in* EDB *at position* k*,* i.e. *sets* $\mathsf{EDB}[k] = v$*. (4) Running the setup routine* $\mathsf{Setup}(\mathsf{K}, N, W, \mathsf{DB})$ *is equivalent to running the setup* $\mathsf{Setup}(\mathsf{K}, N, W, \emptyset)$ *with an empty database and subsequently performing an update operation for each keyword-identifier pair* $(w, \mathsf{id}) \in \mathsf{DB}$ *locally. (5)* Σ *is forward-secure.*

Construction. A detailed description is given in Algorithm 2. Let us explain it here in text. Let Σ be a suitable SSE scheme. First, observe that it is easy to add dummy updates to Σ if we are willing to let server storage grow linearly with the number of dummy updates: we could simply let $\mathsf{DummyUpdate}$ perform a real update with a fresh keyword-identifier pair. Because Σ is forward-secure, the leakage of either type of update would be $\perp$. The problem is that the server would have to store the keyword-identifier pairs arising from dummy updates, potentially blowing up storage overhead. This is what we wish to avoid.

Instead, the idea is to wrap Σ inside an encrypted two-choice allocation scheme (cf. Section 2.3). First, $\mathsf{Dummy}(\Sigma)$ initializes a two-choice scheme with $m = N/\widetilde{\mathcal{O}}(\log\log N)$ bins $B_1, \ldots, B_m$, each of capacity $\widetilde{\mathcal{O}}(\log\log N)$ (where one unit corresponds to the storage cost of one identifier in Σ). Because Σ is suitable, we know server storage in Σ behaves like a key-value store: for each update with token τ, the server stores the corresponding data items under the keys $\mathsf{keys}(\tau)$. Let $\mathsf{H} : \mathcal{K} \to \{1, \ldots, m\}^2$ map keys to pairs of bins. In $\mathsf{Dummy}(\Sigma)$, whenever Σ would store a data item under

a key k, the same item is instead stored in one of the two bins B_α, B_β, where $(\alpha, \beta) \leftarrow \mathsf{H}(k)$. The destination bin is chosen among B_α, B_β according to the two-choice process: the item is inserted into whichever bin currently contains fewer items.

In $\mathsf{Dummy}(\Sigma)$, the client always downloads and sends back full bins, padded to their maximal capacity $\widetilde{\mathcal{O}}(\log \log N)$, and encrypted under a key $\mathsf{K}_{\mathsf{Enc}}$ known only to the client. This is where the $\widetilde{\mathcal{O}}(\log \log N)$ overhead in communication comes from. On the other hand, thanks to the properties of the two-choice process, dummy updates can be realized easily: the client simply asks to access two bins $(\alpha, \beta) \leftarrow \mathsf{H}(k)$ for a fresh key k, re-encrypts them, and re-uploads them. This matches the behavior of real updates (up to the IND-CPA security of Enc, and the pseudo-randomness of H, modeled as a random oracle). In more detail, the key k for dummy updates is generated by $\Sigma.\mathsf{Update}$ using a reserved dummy keyword w_{dum}, and a fresh identifier id chosen by the client.

Remark. Although the outline given above is natural, the detailed construction in Algorithm 2 involves some subtlety. During setup, Σ receives as upper bound for the size of the database $N+D$, rather than just N. This is useful for the security proof. It also means that the encrypted database *generated by* Σ may scale with D. However, that database is not needed to run $\mathsf{Dummy}(\Sigma)$, so this has no impact on storage efficiency. To see that the database generated by Σ is not needed by $\mathsf{Dummy}(\Sigma)$, observe in Algorithm 2 that $\mathsf{Dummy}(\Sigma)$ crucially only makes use of the *client-side* part of the protocols of Σ; the only relevant aspect of the server-side part of those protocols is the keys map. This is made possible by the suitability assumption on Σ, which is used very strongly.

Algorithm 2 $\mathsf{Dummy}(\Sigma)$

$\mathsf{Dummy}(\Sigma).\mathsf{Setup}(\mathsf{K}, N, W, D, \mathsf{DB})$

1: Pick dummy keyword w_{dum}
2: $\mathsf{EDB}_\Sigma \leftarrow \Sigma.\mathsf{Setup}(\mathsf{K}_\Sigma, N + D, W + 1, \emptyset)$
3: Initialize $m = N/\widetilde{\mathcal{O}}(\log \log N)$ empty bins $B_1, \ldots, B_m$ of capacity $\widetilde{\mathcal{O}}(\log \log N)$.
4: **for all** $(w, \mathsf{id}) \in \mathsf{DB}$ **do**
5: $(k, v) \leftarrow \Sigma.\mathsf{Update}_C(\mathsf{K}, (w, \mathsf{id}), \mathsf{add})$
6: $(\alpha, \beta) \leftarrow \mathsf{H}(k)$
7: Insert id into the bin B_γ with fewest items among B_α, B_β
8: **return** $\mathsf{EDB} = (\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(B_i))_{i=1}^m$

$\mathsf{Dummy}(\Sigma).\mathsf{KeyGen}(1^\lambda)$

1: Sample $\mathsf{K}_\Sigma \leftarrow \Sigma.\mathsf{KeyGen}(1^\lambda)$ and encryption key $\mathsf{K}_{\mathsf{Enc}}$
2: **return** $\mathsf{K} = (\mathsf{K}_\Sigma, \mathsf{K}_{\mathsf{Enc}})$

$\mathsf{Dummy}(\Sigma).\mathsf{Search}(\mathsf{K}, w; \mathsf{EDB})$

Client:
1: $\tau \leftarrow \Sigma.\mathsf{Search}_C(\mathsf{K}_\Sigma, w)$
2: **send** τ

Server:
1: $k_1, \ldots, k_q \leftarrow \mathsf{keys}(\tau)$
2: $(\alpha_i, \beta_i) \leftarrow \mathsf{H}(k_i)$ for $i \in [1, q]$
3: **send** $\{B_{\alpha_i}^{\mathsf{enc}}, B_{\beta_i}^{\mathsf{enc}}\}_{i=1}^q$

$\mathsf{Dummy}(\Sigma).\mathsf{Update}(\mathsf{K}, (w, \mathsf{id}), \mathsf{op}; \mathsf{EDB})$

Client:
1: $(k, v) \leftarrow \Sigma.\mathsf{Update}_C(\mathsf{K}_\Sigma, (w, \mathsf{id}), \mathsf{add})$
2: **send** k

Server:
1: $(\alpha, \beta) \leftarrow \mathsf{H}(k)$
2: **send** $B_\alpha^{\mathsf{enc}}, B_\beta^{\mathsf{enc}}$

Client:
1: Decrypt $B_\alpha^{\mathsf{enc}}, B_\beta^{\mathsf{enc}}$ to B_α, B_β
2: Insert id into the bin B_γ with fewest identifiers among B_α, B_β
3: **send** re-encrypted B_α, B_β

Server:
1: Replace $B_\alpha^{\mathsf{enc}}, B_\beta^{\mathsf{enc}}$ with received bins

$\mathsf{Dummy}(\Sigma).\mathsf{DummyUpdate}(\mathsf{K}; \mathsf{EDB})$

Client:
1: Pick fresh identifier id
2: $(k, v) \leftarrow \Sigma.\mathsf{Update}_C(\mathsf{K}_\Sigma, (w_{\mathsf{dum}}, \mathsf{id}), \mathsf{add})$
3: **send** k

Server:
1: $(\alpha, \beta) \leftarrow \mathsf{H}(k)$
2: **send** $B_\alpha^{\mathsf{enc}}, B_\beta^{\mathsf{enc}}$

Client:
1: **send** re-encrypted B_α, B_β

Server:
1: Replace $B_\alpha^{\mathsf{enc}}, B_\beta^{\mathsf{enc}}$ with received bins

Security. The security of $\mathsf{Dummy}(\Sigma)$ follows naturally from that of Σ. Indeed, $\mathsf{Dummy}(\Sigma)$ essentially amounts to running Σ inside encrypted bins, with the difference that Σ is scaled for size $N + D$. As a result, $\mathsf{Dummy}(\Sigma)$ intuitively has at most the same leakage as Σ for Setup, Search and Update, except that D is additionally leaked. Regarding $\mathsf{DummyUpdate}$, the only difference

between real and dummy updates is that a new identifier is inserted in the bin with the former, while the bins are re-encrypted without modifying their content with the latter. Since Enc is IND-CCA-secure, and bins are always padded to ther full size (cf. Section 2.1), the two behaviors are indistinguishable.

Theorem 1. *Let Σ be a suitable, $\mathcal{L}^{\mathsf{fs}}$-adaptively secure SSE scheme. Let Enc be an IND-CPA secure encryption scheme. Let H be a random oracle. Let $N \geq \lambda$, and $D = \mathsf{poly}(N)$. $\mathsf{Dummy}(\Sigma)$ is a correct and secure SSE scheme supporting dummy updates with respect to leakage $\mathcal{L}_{\mathsf{dum}} = (\mathcal{L}_{\mathsf{Stp}}, \mathcal{L}_{\mathsf{Srch}}, \mathcal{L}_{\mathsf{Updt}})$, where $\mathcal{L}_{\mathsf{Stp}}(\mathsf{DB}, N, D, W) = (N, D, W), \mathcal{L}_{\mathsf{Srch}}(w) = (\mathsf{qp}, \ell)$, and $\mathcal{L}_{\mathsf{Updt}}(\mathsf{op}, w, \mathsf{id}) = \bot$.*

The full proof is postponed to Appendix F. We sketch it here. Because $\mathsf{2C}$ guarantees a maximum load of $\widetilde{\mathcal{O}}(\log\log N)$ with overwhelming probability if $N \geq \lambda$ (Lemma 1), correctness follows from the correctness of Σ. We turn to security. Let Sim_Σ be a simulator for Σ. During setup, the client initializes the encrypted database EDB_Σ of Σ and outputs $m = N/\widetilde{\mathcal{O}}(\log\log N)$ encrypted bins. The output leaks nothing but N, since the bins are encrypted. Note that for subsequent updates and searches, the state of Sim_Σ still needs to be initialized by simulating EDB_Σ. (Here, we use the fourth property of Definition 8.) This potentially leaks N, D and W. For search queries, the search token can be sampled via Sim_Σ. This requires the query pattern qp_Σ of Σ and the length ℓ of the identifier list matching the searched keyword. Note that each search and update query induces a corresponding query on Σ. Thus, the query pattern of $\mathsf{Dummy}(\Sigma)$ and Σ are equivalent. Consequently, ℓ and qp are leaked. Similarly, all updates (including dummy updates) can be simulated via Sim_Σ. As updates leak nothing in Σ, neither dummy nor real updates have any leakage.

3.3 Efficient Instantiations

Definition 9. *A suitable SSE scheme Σ is said to be* efficient *if:*

- *Σ has $\mathcal{O}(1)$ storage efficiency;*
- *If EDB contains ℓ values matching keyword w, then for $\tau \leftarrow \Sigma.\mathsf{Search}_C(\mathsf{K}, w)$, $|\mathsf{keys}(\tau)| = \ell$.*

Later, we will only use Σ to store full pages. That is, the atomic items stored in Σ will be identifier lists of size p, rather than single identifiers. Each key will map to one list of size p. Each access to the encrypted database EDB then translates to one page access, and retrieves p identifiers. On the other hand, if Σ is efficient in the sense above, the number of accesses is minimal, hence Σ with full-page items has page efficiency $\mathcal{O}(1)$. $\mathsf{Dummy}(\Sigma)$ then has page efficiency $\widetilde{\mathcal{O}}(\log\log N)$. Moreover, if Σ has $\mathcal{O}(1)$ storage efficiency, so does $\mathsf{Dummy}(\Sigma)$.

Putting both remarks together, we see that if Σ is efficient per Definition 9, then $\mathsf{Dummy}(\Sigma)$ has storage efficiency $\mathcal{O}(1)$, and when used to store full-page items as outlined above, it has page efficiency $\widetilde{\mathcal{O}}(\log\log N)$. To instantiate this idea, the $\Sigma o\phi o\varsigma$ [9] and Diana [11] schemes are good choices: they are both suitable, efficient, and $\mathcal{L}^{\mathsf{fs}}$-adaptively secure (assuming identifiers are encrypted before being stored on the server).

Remark. The $\mathsf{Dummy}(\Sigma)$ framework technically does not exclude the possibility that the sizes of individual search tokens and keys could scale with D, insofar as they are produced by Σ, and $N + D$ is an input of $\Sigma.\mathsf{Setup}$. In practice, the overhead is at most constant for $\Sigma o\phi o\varsigma$ and Diana, and can be eliminated entirely with a careful instantiation. To simplify the presentation, we have not added formal requirements for that purpose at the framework level.

4 BigHermes: the Big Database Regime

We are now ready to present our main construction, Hermes. The construction differs depending on whether $N \geq pW$ or $N < pW$. This section presents $\mathsf{BigHermes}$, which deals with the case $N \geq pW$. Throughout the section, we assume $N \geq pW$, and let Σ_{dum} be an efficient forward-secure SSE supporting dummy queries, in the sense of Section 3.

The final $\mathsf{BigHermes}$ construction is rather involved. To simplify the explanation, we build $\mathsf{BigHermes}$ progressively. We introduce three variants: $\mathsf{BigHermes}_0$, $\mathsf{BigHermes}_1$, $\mathsf{BigHermes}_2$. The

three variants are gradually more complex, but achieve gradually stronger properties. The difference lies in the efficiency guarantees. $\mathsf{BigHermes}_0$ uses the idea of dummy updates from Section 3 to achieve sublogarithmic page efficiency, communication and time complexity overheads; but only in an amortized sense. $\mathsf{BigHermes}_1$ shows how $\mathsf{BigHermes}_0$ can be deamortized in page efficiency and communication, notably without the use of ORAM found in prior work [29]. Finally, $\mathsf{BigHermes}_2$ builds on $\mathsf{BigHermes}_1$ to deamortize time complexity, and completes the construction.

4.1 $\mathsf{BigHermes}_0$: Amortized BigHermes

If a list of identifiers matching the same keyword contains exactly p identifiers, let us say that the list is *full*. If it contains less than p identifiers, it is *underfull*. In $\mathsf{BigHermes}_0$, for each keyword w, the server stores exactly one underfull list of identifiers matching w. For that purpose, the server stores W bins $(B_{w_i})_{i=1}^{W}$ of capacity p, one for each keyword. Full lists are stored separately in an instance of Σ_{dum}. When searching for keyword w, the client will retrieve the corresponding bin, and call $\Sigma_{\mathsf{dum}}.\mathsf{Search}$ to fetch full lists matching w (if any).

Naively, to perform an update on keyword w, the client could simply fetch the corresponding bin and add the new identifier, emptying the bin into Σ_{dum} if it is full. However, this would trivially break forward security, because the server would learn information about which keyword is being updated. Instead, the scheme proceeds in epochs (cf. Section 2). Each epoch corresponds to W consecutive updates. The client buffers all updates arising during the current epoch, until the buffer contains W updates, and the epoch ends. At the end of the epoch, the client downloads all bins $(B_{w_i})_{i=1}^{W}$ from the server, updates them with the W new identifiers from its buffer, and pushes the updated bins back to the server.

If one of the bins becomes full during this end-of-epoch update, it would be tempting to immediately insert the full list into Σ_{dum}. However, this would again break forward security, as the server would learn how many bins became full during the epoch. To hide that information, $\mathsf{BigHermes}_0$ takes advantage of dummy updates. Observe that during an end-of-epoch update, at most W lists can become full. To hide how many bins become full, $\mathsf{BigHermes}_0$ always performs exactly W updates on Σ_{dum} at the end of the epoch, padding real updates with dummy updates as necessary. From a security standpoint, this approach works because real updates and dummy updates are indistinguishable. From an efficiency standpoint, dummy updates have no impact on the efficiency of Σ_{dum}, as discussed in Section 3 (in short, while dummy updates are indistinguishable from real ones for the server, they effectively do nothing). The only cost of dummy updates is in communication complexity, and page efficiency. For both quantities, note that W updates to Σ_{dum} are performed per epoch of W client updates, thus at most one dummy update per client update. Hence, we can instantiate Σ_{dum} for at most $D_{\mathsf{dum}} = N$ dummy updates, and $\mathsf{BigHermes}_0$ directly inherits the same page efficiency and communication overhead as Σ_{dum}.

It remains to discuss the order of (real and dummy) updates on Σ_{dum} at the end of an epoch. This order has an impact on the security of the scheme. To see this, suppose that the following process is used: at the end of an epoch, push full lists to Σ_{dum} for each keyword where this is needed, taking keywords in a fixed order $w_1, w_2, \ldots$; then pad with dummy updates. Now imagine that during an epoch, the client fills a list for keyword w_2, but not for w_1. When the client subsequently performs a search on w_2, the server can see that the locations accessed in Σ_{dum} during the search (partially) match the locations accessed during the first update on Σ_{dum} at the end of the previous epoch (since the first update was for w_2). Since it was the first update, this implies that no update was needed for w_1. The server deduces that no list for w_1 was full at the end of the previous epoch. This breaks security. To avoid that issue, at the end of an epoch, $\mathsf{BigHermes}_0$ first computes all W updates that will need to be issued to Σ_{dum}, then permutes them uniformly at random, before sending the updates to the server. As the security proof will show, this is enough to obtain security.

In the end, $\mathsf{BigHermes}_0$ achieves storage efficiency $\mathcal{O}(1)$, inherited from the same property of Σ_{dum}, and because of the assumption $N \geq pW$, the storage cost of the W bins is $\mathcal{O}(N)$. As noted earlier, $\mathsf{BigHermes}_0$ also inherits the page efficiency and communication overhead of Σ_{dum}. Because the number of entries in the database of Σ_{dum} is at most N/p, it follows that $\mathsf{BigHermes}_0$ has page efficiency and communication overhead $\widetilde{\mathcal{O}}(\log\log(N/p))$. However, this is only in an amortized sense, since batches of W updates are performed together at the end of each epoch.

4.2 $\mathsf{BigHermes}_1$: BigHermes with Deamortized Communication

The reason $\mathsf{BigHermes}_0$ successfully hides which underfull list requires an update when the client wishes to insert a new keyword-document pair is simple: all underfull lists (bins) are updated at the same time, at the end of an epoch. This approach may be interpreted as hiding the access pattern to bins using a trivial ORAM: the entire set of bins is downloaded, updated locally, and uploaded back to the server. Dummy updates are then used to hide how many full bins are pushed to Σ_{dum}. This approach is possible due to amortization: a trivial ORAM access only occurs once every W client updates, and updates all W bins simultaneously.

In short, $\mathsf{BigHermes}_1$ deamortizes $\mathsf{BigHermes}_0$ by no longer updating bins all at once at the end of an epoch, and instead updating them one by one over the course of the next epoch. Thus, $\mathsf{BigHermes}_1$ may be understood as a "deamortized" trivial ORAM, which turns out to be much more efficient in our setting than directly using a standard ORAM, as in prior work [29]. Among other benefits, this is what allows Hermes to achieve sublogarithmic efficiency, avoiding the logarithmic overhead inherent in ORAM [28]. Let us now explain the algorithm. Pseudo-code is available in Algorithm 3. A visual representation of the update procedure is also given in Figure 1.

At a high level, at the end of an epoch, the client pre-computes where the W new identifiers from the epoch should be stored on the server, without actually pushing them to the server. To that end, the client maintains a (client-side) table T_{len}, that maps each keyword to the number of matching identifiers currently in the server-side database. Using T_{len}, at the end of an epoch, for each keyword w, the client splits the list of new identifiers matching the keyword into three (possibly empty) sublists: (1) a sublist that completes the content of B_w to a full list (if possible); (2) full sublists of size p; and (3) an underfull sublist of remaining identifiers (if any). Let $\mathsf{CB}_{\mathsf{out}}$ be a (client-side) buffer that maps each keyword to sublists (1) and (3). All sublists of type (2) are stored in another buffer CFP that maps an integer in $[1, W/p]$ to either a full list or $\perp$. (Note that there are at most W/p such sublists in total.) Once all keywords are processed in that manner, the content of CFP is shuffled randomly.

Over the course of the next epoch, the contents of CFP and $\mathsf{CB}_{\mathsf{out}}$ are pushed to the server according to a fixed schedule. In more detail, during the k-th update operation of the next epoch, the client inserts the new keyword-identifier pair into $\mathsf{CB}_{\mathsf{new}}$. This new keyword-identifier pair will not be processed until the end of the current epoch. The client then moves on to pushing updates that were buffered from the end of the previous epoch, proceeding as follows. She downloads the bin B_{w_k} for the k-th keyword from the server. The client then retrieves from $\mathsf{CB}_{\mathsf{out}}[w_k]$ the list L_1 that completes the content of B_{w_k} to a list of size p, and the new underfull list L_x. If there are enough new identifiers in L_1 to complete the content of B_{w_k} to a full list, the new full list is written to Σ_{dum}, and the contents of B_{w_k} is replaced with L_x. Otherwise, the client performs a dummy update to Σ_{dum}, and adds the identifiers of L_1 to B_{w_k}. In either case, B_{w_k} is then re-encrypted and uploaded to the server. Finally, if $k \leq W/p$, the client also retrieves $L_S \leftarrow \mathsf{CFP}[k]$. Recall that L_S is either a full list buffered from the previous epoch, or $\perp$. If $L_S = \perp$ the client performs a dummy update, otherwise she writes L_S to Σ_{dum}. In total, from the point of view of the server, during the k-th client update in a given epoch, the bin B_{w_k} is accessed, and if $k \leq W/p$ (resp. $k > W/p$), two (resp. one) updates are performed in Σ_{dum}. Thus, the access pattern during a client update is fully predictable, and reveals no information to the server. Also note that during each epoch, at most $2W$ dummy updates are performed. Hence, a number of at most $D_{\mathsf{dum}} = 2N$ dummy updates are performed on Σ_{dum}.

4.3 $\mathsf{BigHermes}_2$: Fully Deamortized BigHermes

Observe that in $\mathsf{BigHermes}_1$, the client performs most computation at the end of each epoch. We now sketch how we deamortize even the client computation, and refer to Appendix C for details.

Recall that the client needs to assign each identifier to a sublist of type (1), (2) or (3). Given this assignment, the sublists can be moved to $\mathsf{CB}_{\mathsf{out}}$ and CFP. Observe that after an identifier is assigned to a sublist, it is easy to deamortize the process of moving it into $\mathsf{CB}_{\mathsf{out}}$ or CFP. It is less straightforward to assign a fresh identifier id to the correct sublist "on-the-fly" when it is added. Note that when (w, id) is added, it is not possible to look at the list L of all buffered identifiers matching w, as this list might be of size $\mathcal{O}(W)$. We show that a pipeline pre-computation can

resolve this problem, with a constant overhead of additional data structures. These are either copies of existing data structures, or tables to store intermediate information, all of size $\mathcal{O}(W)$.

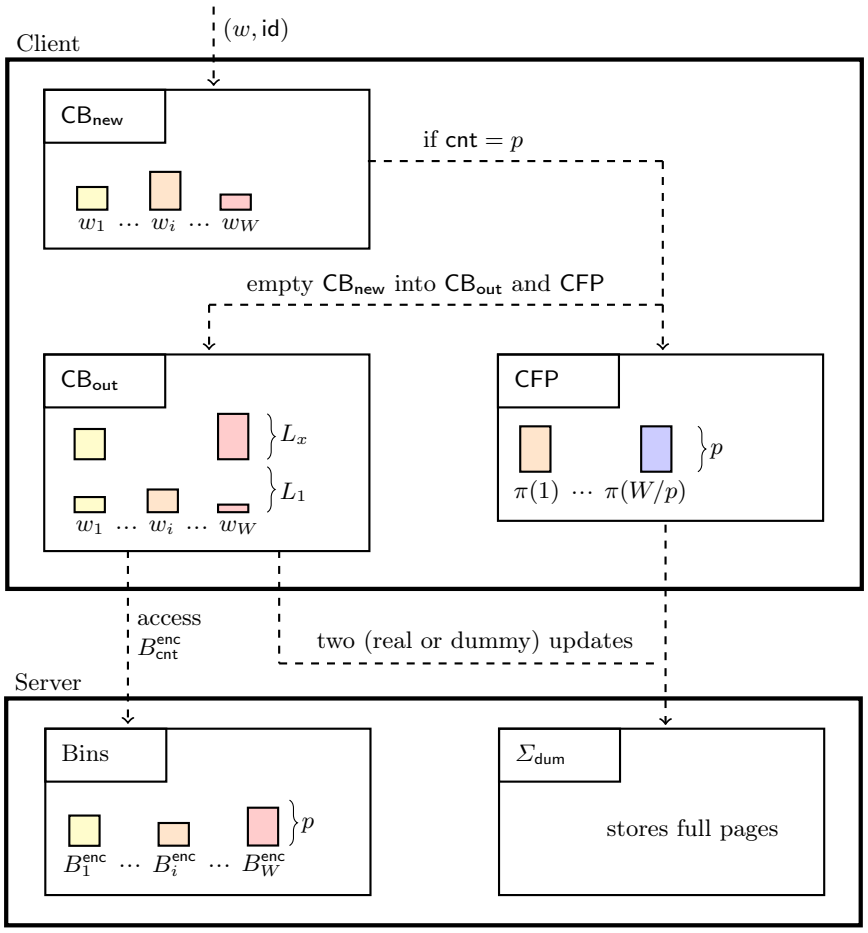

Fig. 1 – Sketch of the update schedule of $\mathsf{BigHermes}_1$. (The data structure T_{len} are omitted for clarity.) Each update, the variable cnt is incremented and the dotted lines are executed. See Algorithm 3 for pseudo-code.

4.4 Security

BigHermes is forward-secure with standard leakage $\mathcal{L}^{\mathsf{fs}}$, as stated next.

Theorem 2. *Let N be an upper bound on the size of the database, let p be the page size, and let W be an upper bound on the number of keywords. Let $N \geq pW$, and assume $N/p \geq \lambda$. Let Σ_{dum} be a forward-secure* SSE *supporting dummy updates, let* Enc *be an IND-CPA secure encryption scheme, and let* PRF *be a secure pseudorandom function (for the preprocessing of keywords). Then* BigHermes *is correct and $\mathcal{L}^{\mathsf{fs}}$-adaptively semantically secure.*

We now sketch the security proof, and refer to Appendix G for the full proof. BigHermes stores all full identifier lists of size p in Σ_{dum}, and one underfull sublist per keyword w in the bin B_w. Each search, the corresponding bin is accessed, and the client searches for all full lists on the server in Σ_{dum}. All identifiers that are not retrieved are contained in a buffer on the client. Thus, correctness follows immediately from the correctness of Σ_{dum}.

The setup only leaks W and N to the server, as the bins are encrypted, and the security of Σ_{dum} guarantees that $\mathsf{EDB}_{\Sigma_{\mathsf{dum}}}$ leaks no other information. Further, updates leak no information which

Algorithm 3 $\mathsf{BigHermes}_1$

$\mathsf{BigHermes}_1.\mathsf{Setup}(\mathsf{K}, N, W, \mathsf{DB})$

Initialize W empty bins $B_{w_1}, \ldots, B_{w_W}$ of capacity p
Initialize empty database $\mathsf{DB}_{\mathsf{dum}}$
for all keywords w **do**
  Split $\mathsf{DB}(w)$ into x lists L_i such that L_x has size at most $p-1$, and the remaining lists have size p
  Insert pairs $\{(w, L_i)\}_{i=1}^{x-1}$ into $\mathsf{DB}_{\mathsf{dum}}$
  Insert list L_x into B_w
  $T_{\mathsf{len}}[w] \leftarrow |\mathsf{DB}(w)|$
$\mathsf{cnt} \leftarrow 0$
$(N_{\mathsf{dum}}, D_{\mathsf{dum}}, W_{\mathsf{dum}}) = (N/p, 2N, W)$
$\mathsf{EDB}_{\Sigma_{\mathsf{dum}}} \leftarrow \Sigma_{\mathsf{dum}}.\mathsf{Setup}(\mathsf{K}_{\Sigma_{\mathsf{dum}}}, N_{\mathsf{dum}}, D_{\mathsf{dum}}, W_{\mathsf{dum}}, \mathsf{DB}_{\mathsf{dum}})$
$\mathsf{EDB} \leftarrow (\mathsf{EDB}_{\Sigma_{\mathsf{dum}}}, \{\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(B_{w_i})\}_{i=1}^{W})$ **return** EDB

$\mathsf{BigHermes}_1.\mathsf{KeyGen}(1^\lambda)$

Sample $\mathsf{K}_{\mathsf{Enc}}$ for Enc with security parameter λ and $\mathsf{K}_{\Sigma_{\mathsf{dum}}} \leftarrow \Sigma_{\mathsf{dum}}.\mathsf{KeyGen}(1^\lambda)$
return $\mathsf{K} = (\mathsf{K}_{\mathsf{Enc}}, \mathsf{K}_{\Sigma_{\mathsf{dum}}})$

$\mathsf{BigHermes}_1.\mathsf{Search}(\mathsf{K}, w; \mathsf{EDB})$

Client:

Perform $\Sigma_{\mathsf{dum}}.\mathsf{Search}(\mathsf{K}_{\Sigma_{\mathsf{dum}}}, w; \mathsf{EDB})$
send w

Server:

send B_w^{enc}

$\mathsf{BigHermes}_1.\mathsf{Update}(\mathsf{K}, (w, \mathsf{id}), \mathsf{add}; \mathsf{EDB})$

Client:

if $\mathsf{cnt} = p$ **then**
  $\pi \leftarrow$ uniformly random permuation of $[1, W/p]$
  Initialize empty set S of full pages
  for all keywords w **do**
    $L \leftarrow \mathsf{CB}_{\mathsf{new}}[w]$
    $r \leftarrow T_{\mathsf{len}}[w] \bmod p$
    $T_{\mathsf{len}}[w] = T_{\mathsf{len}}[w] + |L|$
    Split L into x lists L_i such that L_1 has size at most $p-r$ (exactly size $p-r$ if $x > 1$), L_x has size at most p, and the remaining lists all have size p
    $\mathsf{CB}_{\mathsf{out}}[w] \leftarrow \mathsf{CB}_{\mathsf{out}}[w] \cup \{(L_1, L_x)\}$
    $S \leftarrow S \cup \{(w, L_2), \ldots, (w, L_{x-1})\}$
  $\mathsf{CFP}[\pi(i)] \leftarrow S[i]$ for $1 \le i \le |S|$
  Empty $\mathsf{CB}_{\mathsf{new}}$ and set $\mathsf{cnt} = 0$
$\mathsf{CB}_{\mathsf{new}}[w] \leftarrow \mathsf{CB}_{\mathsf{new}}[w] \cup \{\mathsf{id}\}$
$\mathsf{cnt} \leftarrow \mathsf{cnt} + 1$
send cnt

Server:

send $B_{\mathsf{cnt}}^{\mathsf{enc}}$

(continue description of update)

Client:

Retrieve list L of identifiers from $B_{\mathsf{cnt}} = \mathsf{Dec}_{\mathsf{K}_{\mathsf{Enc}}}(B_{\mathsf{cnt}}^{\mathsf{enc}})$
$(L_1, L_x) \leftarrow \mathsf{CB}_{\mathsf{out}}[w_{\mathsf{cnt}}]$
$\mathsf{CB}_{\mathsf{out}}[w_{\mathsf{cnt}}] \leftarrow \perp$
if $L_1 \neq L_x$ **then** ▷ $x > 1$
  Run $\Sigma_{\mathsf{dum}}.\mathsf{Update}(\mathsf{K}, (w_{\mathsf{cnt}}, L_1 \cup L), \mathsf{add}; \mathsf{EDB})$
  $B_{\mathsf{cnt}} \leftarrow L_x$
else ▷ $x = 1$
  Run $\Sigma_{\mathsf{dum}}.\mathsf{DummyUpdate}(\mathsf{K}; \mathsf{EDB})$
  $B_{\mathsf{cnt}} \leftarrow L \cup L_1$
if $\mathsf{cnt} \le W/p$ **then**
  if $\mathsf{CFP}[\mathsf{cnt}] \neq \perp$ **then**
    $(w, L_S) \leftarrow \mathsf{CFP}[\mathsf{cnt}]$
    Run $\Sigma_{\mathsf{dum}}.\mathsf{Update}(\mathsf{K}, (w, L_S), \mathsf{add}; \mathsf{EDB})$
    $\mathsf{CFP}[\mathsf{cnt}] \leftarrow \perp$
  else
    Run $\Sigma_{\mathsf{dum}}.\mathsf{DummyUpdate}(\mathsf{K}; \mathsf{EDB})$
send $B_{\mathsf{cnt}}^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(B_{\mathsf{cnt}})$

Server:

Update $B_{\mathsf{cnt}}^{\mathsf{enc}}$

follows from two facts: (1) exactly one bin is accessed each update via a fixed schedule known in advance; (2) dummy updates and real updates leak no information, and are indistinguishable in the view of the server, owing to the security of Σ_{dum}.

It remains to consider searches. Each search on keyword w only leaks the query pattern qp, and the length ℓ of the identifier list for w. (Recall from Section 2.2 that the query pattern qp is equal to the search pattern sp and update pattern up.) To establish this, we need to show that the view of the server can be simulated using only qp and ℓ. Based on the search pattern and the fact that bins are encrypted with IND-CPA encryption, simulating access to the bin B_w is straightforward. Thus, security reduces to the simulation of Σ_{dum}. Because Σ_{dum} is secure, we know that its behavior can be simulated as long as we can compute the query pattern $\mathsf{qp}_{\Sigma_{\mathsf{dum}}}$ and answer length $\ell_{\Sigma_{\mathsf{dum}}}$ for Σ_{dum}. To see that this is the case, first observe that the number of Σ_{dum}-updates on w and the load of B_w per epoch can be recomputed given only up and ℓ. From there, the simulator can compute the number ℓ_i of full lists for keyword w that were pushed to Σ_{dum} during a given epoch i. Clearly, $\sum_i \ell_i = \ell_{\Sigma_{\mathsf{dum}}}$. To deduce the update pattern $\mathsf{up}_{\Sigma_{\mathsf{dum}}}$ for Σ_{dum}, it remains to determine when each update was performed during the epoch. Intuitively, because updates to Σ_{dum} occurring in a given epoch are permuted uniformly at random, it suffices to choose x updates to Σ_{dum} uniformly at random among updates issued during the epoch (excluding updates already chosen for the same purpose on a different keyword). This yields $\mathsf{up}_{\Sigma_{\mathsf{dum}}}$. On the other hand, each search query to BigHermes triggers exactly one search query to Σ_{dum}: the search pattern of Σ_{dum} matches the search pattern of BigHermes. Thus, the simulator can compute $\mathsf{qp}_{\Sigma_{\mathsf{dum}}}$, and we are done.

4.5 Efficiency

We now analyze the efficiency of BigHermes, when Σ_{dum} is efficient (in the sense of Section 3.3). Each client-side data structure, including identifier buffers and tables, has size $\mathcal{O}(W)$. Since there is a constant number of such structures, overall client storage is $\mathcal{O}(W)$. On the server side, Σ_{dum} has storage efficiency $\mathcal{O}(1)$, and the bins require $pW = \mathcal{O}(N)$ storage, hence overall storage efficiency is $\mathcal{O}(1)$. We turn to page efficiency. During each update, one bin of size p is read, and two updates on Σ_{dum} are performed. Since Σ_{dum} only stores full pages, of which there can be at most N/p, $\Sigma_{\mathsf{dum}}.\mathsf{Update}$ has page efficiency $\widetilde{\mathcal{O}}(\log\log(N/p))$. Similarly, a search on a keyword w with ℓ matching identifiers induces one bin access and (at most) $\lfloor \ell/p \rfloor$ accesses of $\widetilde{\mathcal{O}}(\log\log(N/p))$ pages each. We conclude that BigHermes has page efficiency $\widetilde{\mathcal{O}}(\log\log(N/p))$.

5 SmallHermes: Overview

In Section 4, we have built BigHermes, under the assumption $N \geq pW$. We now build a scheme SmallHermes with the same efficiency and security properties as BigHermes, but in the regime $N \leq pW$. SmallHermes uses some of the same ideas as BigHermes, and combines them with techniques from the recent LayeredSSE scheme [30]. Similar to BigHermes, we present SmallHermes in three steps, each building on the previous one. To provide a concise overview, we only sketch the construction here. The full description can be found in Appendix D.

$\mathsf{SmallHermes}_0$*: Amortized* SmallHermes. SmallHermes stores identifiers in $m = o(N)$ encrypted bins according to L2C (Section 2.3). Recall that each bin has capacity $\widetilde{\mathcal{O}}(p\log\log(N/p))$, and is conceptually divided into $\log\log(N/p)$ layers. Each list L of (at most p) identifiers matching keyword w is mapped to two bins B_α, B_β. The list L is stored in either B_α or B_β, depending on which bin has fewer items at layer κ, where κ is determined by the size of L. For a search on keyword w, the bins B_α and B_β are retrieved from the server. The client can read the matching identifiers from the bins after decryption. When a list has more than p identifiers, it is split into sublists of size p, which are treated independently.

So far, SmallHermes behaves like LayeredSSE [30]. The main difficulty is how to achieve forward security, that is, how to perform updates with *no* leakage. Here, SmallHermes borrows from BigHermes. Client updates are buffered over the course of an epoch in a client-side buffer $\mathsf{CB}_{\mathsf{new}}$ of capacity W. At the end of an epoch, the client downloads the entire encrypted database EDB from

the server, and performs updates locally. Because we are in the "small database" regime $N \leq pW$, this process has amortized $\mathcal{O}(1)$ page efficiency. Of course, it is desirable to deamortize the algorithms, so that the client does not need to download the entire database at the end of an epoch. This is presented next.

$\mathsf{SmallHermes}_1$*:* SmallHermes *with Deamortized Communication.* The communication of $\mathsf{SmallHermes}_0$ can be deamortized via client-side pre-computation (similar to Section 4.2), yielding a scheme $\mathsf{SmallHermes}_1$ with $\widetilde{\mathcal{O}}(\log\log(N/p))$ page efficiency and $\mathcal{O}(1)$ storage efficiency. To that end, the client stores an additional table T_{load}, which records the load information of each bin on all $\log\log m$ layers. (This requires $\mathcal{O}(W \cdot \log\log m)$ client storage but can be improved to $\mathcal{O}(W)$ with a more complex pre-computation.) Whenever the client decides to store an identifier in a bin, she updates T_{load} accordingly. Using T_{load}, the client can pre-compute locally in which bin a new identifier should be added. Each epoch, she accesses every bin on a fixed schedule, and inserts identifiers according to the local pre-computation.

In the end, a fresh identifier is written to the server after at most $2W$ client updates. During a client update, at most one bin of size $p\mathcal{O}(\log\log(N/p))$ is downloaded. During a search matching ℓ identifiers, exactly $2\lceil \ell/p \rceil$ bins are downloaded. Thus, SmallHermes has $\mathcal{O}(\log\log(N/p))$ page efficiency. It also inherits $\mathcal{O}(1)$ storage efficiency from L2C. Note that a figure visualizing the update schedule of SmallHermes is given (cf. Figure 4).

$\mathsf{SmallHermes}_2$*: Fully Deamortized* SmallHermes. With a complex but efficient pipeline pre-computation, we can remove the additional $\log\log m$ factor in client storage and deamortize both the communication and computation of SmallHermes. The optimization is based on the observation that for inserting the $\mathcal{O}(W)$ new identifiers of $\mathsf{CB}_{\mathsf{new}}$, we do not require the entire load information of L2C, only the load of relevant bin-layer pairs. For W updates, even if each new identifier requires the load of some bin B_γ at some layer κ, there are at most $2W$ such pairs (γ, κ). The load information of these pairs can be precomputed in a pipeline with three steps, where each step is performed in one epoch. The client can then decide where to insert each of the new identifiers based on the load information, and subsequently push the identifiers to the server. This approach yields an update operation with at most $\mathcal{O}(p\log\log(N/p))$ communication *and* computation. In the end, $\mathsf{SmallHermes}_2$ achieves the same worst-case sublogarithmic performance in page efficient, communication and computation, as $\mathsf{BigHermes}_2$.

It is interesting to note that SmallHermes can be simplified if we are willing to make a natural conjecture about the behavior of the weighted two-choice process. This point is discussed in Appendix D.4. The relevant conjecture seems to be a long-standing open problem. If true, a simpler and more elegant variant of SmallHermes can be built, although asymptotic performance remains the same.

5.1 Security and Efficiency

Theorem 3 (Sketch). *If $N \leq pW$, then* SmallHermes *is correct and $\mathcal{L}^{\mathsf{fs}}$-adaptively semantically secure.*

The full formal statement and proof are deferred to Appendix H. Let us outline the proof here. Semantic security follows from the following facts. (1) The bins are encrypted, hence only the upper bound N is leaked during setup. (2) During search, $2 \cdot \lceil \ell/p \rceil$ bins are accessed, where ℓ is the number of identifiers matching the searched keyword w. These bins are re-accessed if w is searched again, but look random random to the server during the initial search. Thus, a search leaks the search pattern, and the number of sublists. (3) Updates are performed by accessing each bin via a fixed schedule (which solely depends on the number of updates). Hence, updates leak no information.

Each data structure on the client requires $\mathcal{O}(W)$ storage. As there are only a constant number of data structures and pipeline steps, total client storage is $\mathcal{O}(W)$. At most one bin is downloaded each update; and each search on keyword w, exactly $2\lceil \ell/p \rceil$ bins are retrieved, where ℓ is the length of the list of identifiers matching w. Because each bin is of size $\widetilde{\mathcal{O}}(p\log\log(N/p))$, page efficiency is $\widetilde{\mathcal{O}}(\log\log(N/p))$, and storage efficiency is $\mathcal{O}(1)$.

6 The Hermes Scheme: Putting Everything Together

We have constructed BigHermes (Section 4), an I/O-efficient SSE scheme with forward security in the big database regime, *i.e.* $N \geq pW$. Similarly, we built SmallHermes (Section 5) in the regime $N \leq pW$. We combine them to construct Hermes.

Hermes. The Hermes scheme simply uses either BigHermes or SmallHermes, depending on which regime the global parameters N, W and p are in. That is, Setup, Search, and Update for Hermes behave exactly as in BigHermes, if $N \geq pW$, and as in SmallHermes otherwise. Clearly, Hermes has $\widetilde{\mathcal{O}}(\log\log(N/p))$ page efficiency and $\mathcal{O}(1)$ storage efficiency. Further, because both sub-schemes are forward-secure with leakage $\mathcal{L}^{\mathsf{fs}}$ (Theorems 2 and 3), the same holds for Hermes. This is formalized in Theorem 4.

Theorem 4. *Let N be an upper bound on the size of the database, and let W be an upper bound on the number of keywords. Let p be the page size. Assume $p \leq N^{1-1/\log\log\lambda}$, and $N/p \geq \lambda$. Let* Enc *be an* IND-CPA*-secure encryption scheme, and let* PRF *be a secure pseudo-random function. The* Hermes *is correct and $\mathcal{L}^{\mathsf{fs}}$-adaptively semantically secure.*

The scheme Hermes has $\mathcal{O}(W)$ client storage, $\mathcal{O}(1)$ storage efficiency (i.e., $\mathcal{O}(N)$ server storage) and $\widetilde{\mathcal{O}}(\log\log(N/p))$ page efficiency. Note that storage is given in memory words (of size $\mathcal{O}(\lambda)$ bits) and that a document identifier can be stored in a single memory word [4].

7 Experimental Evaluation

All evaluations and benchmarks have been carried out on a computer with an Intel Core i7 8550U 1.80 GHz CPU with 8 cores and an 512 GiB PCIe SSD, running Ubuntu 20.04. The SSD page size is 4 KiB. We chose the setting where document identifiers are encoded on 8 bytes. This allows us to support databases with up to 2^{64} documents, where each page fits $p = 512$ entries. We set $\lambda = 128$.

7.1 Evaluation of 2C and L2C

Hermes uses both the 2C and L2C allocation schemes (Section 2.3). We first empirically evaluate the constant c in the page-efficiency bound $c \cdot p \log\log\log(\lambda)\log\log(N/p)$ of those schemes. This is a necessary preliminary step to any concrete instantiation of Hermes, since the constant determines the bin size.

To evaluate the constant, we run experiments allocating balls with total weight $N \in [2^{12}, 2^{22}]$ using L2C and 2C. For each scheme and each parameter set, we perform 10000 trial runs. Each run, we insert balls into $m = \frac{2N}{p\log\log\log(\lambda)\log\log(N/p)}$ bins, adding new balls until a total weight of N is reached. For L2C, ball weights are sampled uniformly at random in $[1, p]$. For 2C, ball weights are set to p. Once all balls are inserted, we measure the load of the most loaded bin, and output the highest number among the 10000 trials. The results are presented in Figure 2. For $c = 2$, we see that the bound holds with a comfortable margin for all variants. This shows that 2C and L2C behave quite well in practice. The constant $c = 2$ is used for Hermes in the experiments below.

7.2 Evaluation of Hermes

We now analyze the I/O performance of Hermes. Because Hermes is the first forward-secure I/O-efficient scheme, two properties that were thought to be at odds until now, there is no direct point of comparison in the literature. Nevertheless, to provide some means of comparison, we include in the evaluation schemes that are forward-secure but not I/O-efficient, and I/O-efficient but not forward-secure. To represent forward-secure but not I/O-efficient schemes, we include the

[4] Our scheme Hermes can be interpreted as an encrypted multi-map and allows to store other items than just document identifiers. In that case, the client storage becomes $\mathcal{O}(n \cdot W)$ bits, where n is the bit-length of the items to be stored.

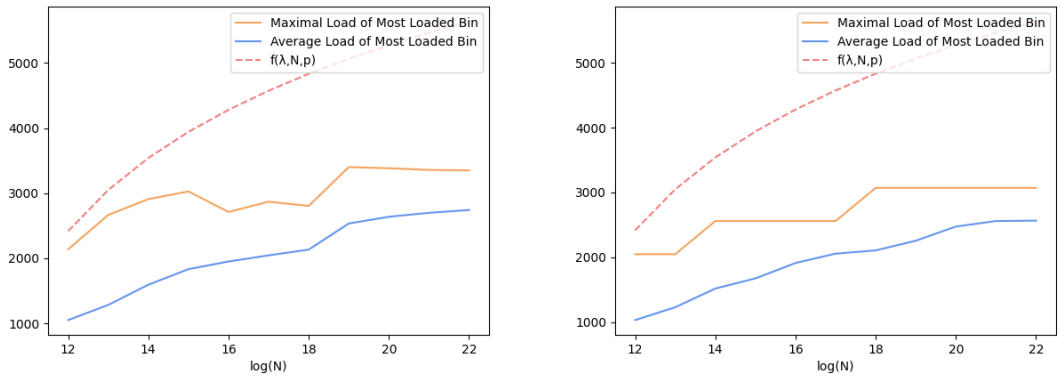

Fig. 2 – The loads of L2C (left) and 2C (right) for 10000 runs with balls of total weight N. The function $f(\lambda, N, p) = c \log\log\log(\lambda)\log\log(N/p)$ is the theoretical upper bound for the most loaded bin with constant $c = 2$ and $\lambda = 128$. For L2C, the weights $\{w_i\}$ are chosen at random in $[1, 512]$, and $w_i = 512$ for 2C.

Diana scheme, which offers state-of-the-art performance [11]. In the category of schemes that are I/O-efficient but not forward-secure, while there are many *static* constructions, the only practical *dynamic* construction we are aware of is IO-DSSE [29]. (As discussed in Section 1.2, another dynamic I/O-efficient construction can be found in [30], but it is a theoretical feasibility result, with no concrete instantiation.) To round out the comparison, we also include the *static* scheme Tethys, which attains the best page efficiency.

For each scheme in the comparison, we analyzed its memory access pattern to deduce its I/O workload for search and update queries. We then simulated that workload using the optimized Flexible I/O benchmark tool `fio` (version 3.19) [5]. The tool measures the throughput of disk accesses under that workload, on an SSD. The same technique was shown in [8] to provide accurate performance estimates. Network latency is not part of the experiment, but we note that it mainly depends on the number of round-trips. For Hermes, this can be reduced to a single roundtrip for both Search and Update (cf. Appendix E.1).

For concreteness, we run our experiments on the English Wikipedia database, containing about 140 million entries, and 4.6 million keywords. We assumed that on average, each keyword matches at least p documents. Note that the exact keyword distribution has little impact on the overheads (relative to a plaintext database) of either throughput or server storage by construction: this is a consequence of the page-efficient approach. These parameters fall under the regime of SmallHermes, and we used the variant based on weighted 2C from Appendix D.4. Due to space limitations, we omit the analysis of Hermes in the other parameter regime. We expect the practical performance of $\mathsf{BigHermes}_1$ to be almost the same as $\mathsf{SmallHermes}_1$.

Results are presented on Figure 3. The throughput of Hermes outperforms other dynamic schemes Diana [11] and IO-DSSE [29]. While Diana has slightly better storage and update efficiency, only one identifier is retrieved during each memory access in search, which heavily impacts page efficiency, and hence, throughput. This is typical of schemes that do not target I/O efficiency, and is indeed the main motivation for I/O-efficient SSE. Concretely, the I/O throughput of Hermes improves over Diana by a factor of 37 (resp. 49) for reads (resp. updates) in this setting.

IO-DSSE suffers from poor server storage efficiency when the database contains many keywords, here requiring 150 GB of encrypted data to store a plaintext index of 1.2 GB (compared to only 4.5 GB for Hermes). Hermes also improves over the I/O throughput of IO-DSSE [5] by a factor 6.

In both cases, this is because IO-DSSE relies on ORAM. The gap is expected to increase for larger databases, since Hermes scales sublogarithmically, while IO-DSSE does not.

[5] When measuring throughput, Figure 3 uses the repaired version of IO-DSSE, since the original version is insecure (cf. Appendix A).

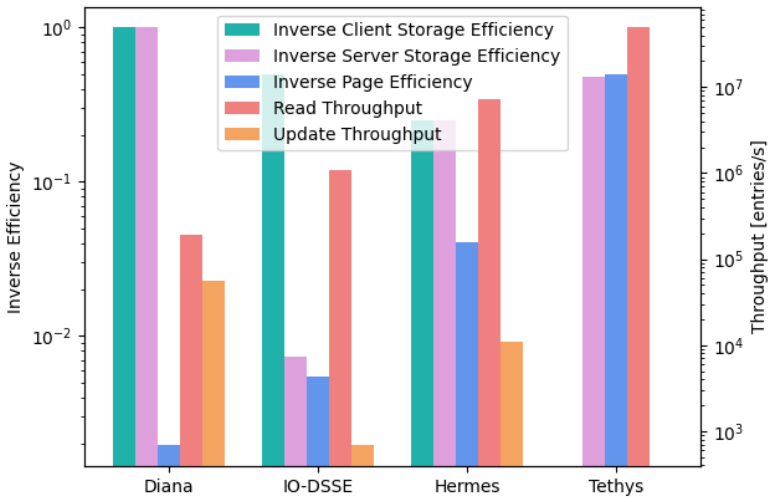

Fig. 3 – Read and update I/O throughput, inverse page efficiency, and inverse storage efficiency for various SSE schemes, in logarithmic scale. Higher is better. Note that *inverse client storage efficiency* is W divided by client storage.

Note that while Hermes requires about twice the client storage of IO-DSSE, we present tradeoffs between client storage and page efficiency in Appendix E. Given the right tradeoff, Hermes outperforms IO-DSSE in all metrics.

Tethys [8] outperforms Hermes in throughput and storage. However, Tethys does not support updates, and is only included for reference.

In conclusion, Hermes vastly outperforms non-I/O-efficient schemes in throughput, as expected. It is perhaps less expected that it also outperforms IO-DSSE, insofar as it adds forward security. But this is not so surprising when considering that the only known approach to "reasonably secure" dynamic I/O-efficient SSE prior to this work was to use ORAM.

References

1. Amjad, G.: Theoretical and Practical Advances in Structured Encryption. Ph.D. thesis, Brown University, USA (2022), https://cs.brown.edu/research/pubs/theses/phd/2022/amjad.ghous.pdf
2. Amjad, G., Patel, S., Persiano, G., Yeo, K., Yung, M.: Dynamic volume-hiding encrypted multi-maps with applications to searchable encryption. Proceedings on Privacy Enhancing Technologies **2023**(1) (2023)
3. Asharov, G., Naor, M., Segev, G., Shahaf, I.: Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In: Wichs, D., Mansour, Y. (eds.) 48th ACM STOC. pp. 1101–1114. ACM Press (Jun 2016). https://doi.org/10.1145/2897518.2897562
4. Asharov, G., Segev, G., Shahaf, I.: Tight tradeoffs in searchable symmetric encryption. Journal of Cryptology **34**(2), 9 (Apr 2021). https://doi.org/10.1007/s00145-020-09370-z
5. Axboe, J.: Flexible I/O Tester (2020), https://github.com/axboe/fio
6. Azar, Y., Broder, A.Z., Karlin, A.R., Upfal, E.: Balanced allocations. In: Proceedings of the twenty-sixth annual ACM symposium on theory of computing. pp. 593–602 (1994)
7. Blackstone, L., Kamara, S., Moataz, T.: Revisiting leakage abuse attacks. In: ISOC Network and Distributed System Security – NDSS 2020 (2020)
8. Bossuat, A., Bost, R., Fouque, P.A., Minaud, B., Reichle, M.: SSE and SSD: Page-efficient searchable symmetric encryption. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021, Part III. LNCS, vol. 12827, pp. 157–184. Springer, Heidelberg, Virtual Event (Aug 2021). https://doi.org/10.1007/978-3-030-84252-9_6
9. Bost, R.: $\Sigma o \phi o \varsigma$: Forward secure searchable encryption. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016. pp. 1143–1154. ACM Press (Oct 2016). https://doi.org/10.1145/2976749.2978303

10. Bost, R., Fouque, P.A.: Security-efficiency tradeoffs in searchable encryption. PoPETs **2019**(4), 132–151 (Oct 2019). https://doi.org/10.2478/popets-2019-0062
11. Bost, R., Minaud, B., Ohrimenko, O.: Forward and backward private searchable encryption from constrained cryptographic primitives. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 1465–1482. ACM Press (Oct / Nov 2017). https://doi.org/10.1145/3133956.3133980
12. Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS 2015. pp. 668–679. ACM Press (Oct 2015). https://doi.org/10.1145/2810103.2813700
13. Cash, D., Jaeger, J., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M.C., Steiner, M.: Dynamic searchable encryption in very-large databases: Data structures and implementation. In: NDSS 2014. The Internet Society (Feb 2014)
14. Cash, D., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M.C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for Boolean queries. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part I. LNCS, vol. 8042, pp. 353–373. Springer, Heidelberg (Aug 2013). https://doi.org/10.1007/978-3-642-40041-4_20
15. Cash, D., Tessaro, S.: The locality of searchable symmetric encryption. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 351–368. Springer, Heidelberg (May 2014). https://doi.org/10.1007/978-3-642-55220-5_20
16. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 577–594. Springer, Heidelberg (Dec 2010). https://doi.org/10.1007/978-3-642-17373-8_33
17. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Juels, A., Wright, R.N., De Capitani di Vimercati, S. (eds.) ACM CCS 2006. pp. 79–88. ACM Press (Oct / Nov 2006). https://doi.org/10.1145/1180405.1180417
18. Demertzis, I., Chamani, J.G., Papadopoulos, D., Papamanthou, C.: Dynamic searchable encryption with small client storage. In: ISOC Network and Distributed System Security – NDSS 2022 (2022)
19. Demertzis, I., Papadopoulos, D., Papamanthou, C.: Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 371–406. Springer, Heidelberg (Aug 2018). https://doi.org/10.1007/978-3-319-96884-1_13
20. Demertzis, I., Papamanthou, C.: Fast searchable encryption with tunable locality. In: Proceedings of the 2017 ACM International Conference on Management of Data. pp. 1053–1067 (2017)
21. Etemad, M., Küpçü, A., Papamanthou, C., Evans, D.: Efficient dynamic searchable encryption with forward privacy. In: Proceedings on Privacy Enhancing Technologie – PoPETS 2018 (2018)
22. Friedman, S., Krishnan, A., Leidenfrost, N.: Hash tables for embedded and real-time systems. EEE Real-Time Embedded System Workshop (2001)
23. George, M., Kamara, S., Moataz, T.: Structured encryption and dynamic leakage suppression. In: Canteaut, A., Standaert, F.X. (eds.) EUROCRYPT 2021, Part III. LNCS, vol. 12698, pp. 370–396. Springer, Heidelberg (Oct 2021). https://doi.org/10.1007/978-3-030-77883-5_13
24. Grubbs, P., Lacharité, M.S., Minaud, B., Paterson, K.G.: Learning to reconstruct: Statistical learning theory and encrypted database attacks. In: 2019 IEEE Symposium on Security and Privacy. pp. 1067–1083. IEEE Computer Society Press (May 2019). https://doi.org/10.1109/SP.2019.00030
25. Gui, Z., Paterson, K.G., Patranabis, S., Warinschi, B.: SWiSSSE: System-wide security for searchable symmetric encryption. Cryptology ePrint Archive, Report 2020/1328 (2020), https://ia.cr/2020/1328
26. Kamara, S., Moataz, T.: SQL on structurally-encrypted databases. In: Peyrin, T., Galbraith, S. (eds.) ASIACRYPT 2018, Part I. LNCS, vol. 11272, pp. 149–180. Springer, Heidelberg (Dec 2018). https://doi.org/10.1007/978-3-030-03326-2_6
27. Kamara, S., Moataz, T., Park, A., Qin, L.: A decentralized and encrypted national gun registry. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1520–1537. IEEE (2021)
28. Larsen, K.G., Nielsen, J.B.: Yes, there is an oblivious RAM lower bound! In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part II. LNCS, vol. 10992, pp. 523–542. Springer, Heidelberg (Aug 2018). https://doi.org/10.1007/978-3-319-96881-0_18
29. Miers, I., Mohassel, P.: IO-DSSE: Scaling dynamic searchable encryption to millions of indexes by improving locality. In: NDSS 2017. The Internet Society (Feb / Mar 2017)
30. Minaud, B., Reichle, M.: Dynamic local searchable symmetric encryption. In: Dodis, Y., Shrimpton, T. (eds.) Advances in Cryptology – CRYPTO 2022. Lecture Notes in Computer Science, Springer (2022)
31. MongoDB: Queryable encryption. https://www.mongodb.com/products/queryable-encryption (2022)

32. Naveed, M., Kamara, S., Wright, C.V.: Inference attacks on property-preserving encrypted databases. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS 2015. pp. 644–655. ACM Press (Oct 2015). https://doi.org/10.1145/2810103.2813651
33. Pappas, V., Krell, F., Vo, B., Kolesnikov, V., Malkin, T., Choi, S.G., George, W., Keromytis, A.D., Bellovin, S.: Blind seer: A scalable private DBMS. In: 2014 IEEE Symposium on Security and Privacy. pp. 359–374. IEEE Computer Society Press (May 2014). https://doi.org/10.1109/SP.2014.30
34. Patranabis, S., Mukhopadhyay, D.: Forward and backward private conjunctive searchable symmetric encryption. In: ISOC Network and Distributed System Security – NDSS 2021 (2021)
35. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: 2000 IEEE Symposium on Security and Privacy. pp. 44–55. IEEE Computer Society Press (May 2000). https://doi.org/10.1109/SECPRI.2000.848445
36. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: NDSS 2014. The Internet Society (Feb 2014)
37. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C.W., Ren, L., Yu, X., Devadas, S.: Path ORAM: an extremely simple oblivious RAM protocol. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM CCS 2013. pp. 299–310. ACM Press (Nov 2013). https://doi.org/10.1145/2508859.2516660
38. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: Holz, T., Savage, S. (eds.) USENIX Security 2016. pp. 707–720. USENIX Association (Aug 2016)

A A Security Flaw in IO-DSSE

In this section, we present an attack that contradicts the security claim of IO-DSSE [29]. In particular, we show that IO-DSSE is *not* forward secure. Consequently, IO-DSSE is potentially vulnerable to attacks targeting SSE schemes without forward security, for example [38]. We first introduce some background, then we present a simple attack on IO-DSSE which contradicts its security claim. Then, we present a stronger attack and highlight the flaw in the security proof of IO-DSSE. Also, we discuss how to repair the flaw which seems to inherently incur a large cost in I/O efficiency.

Overview of IO-DSSE. Let L_w be the list of document identifiers matching keyword w. To optimize I/O efficiency, IO-DSSE splits L_w into blocks of size p, for some fixed block size p (much like Hermes). For each keyword, the last block may be of size less than p. That block is called *incomplete*. Incomplete blocks are stored in a Path ORAM instance. To understand the security issue with IO-DSSE, it is necessary to explain how Path ORAM functions. We refer the reader to [37] for a full description. For the purpose of the attack, it is enough to know the following facts. Let W be the number of keywords. The ORAM needs to store (at most) W blocks. Each block is padded to the full size p before being stored in the ORAM.

To store the W blocks, Path ORAM creates a full binary tree with W leaves (for simplicity, assume W is a power of two). Each node of the tree, including the leaves, can store up to $Z = 5$ elements (sometimes Z is set to 4, but this has no impact on the attack). In addition to the tree, Path ORAM maintains a *stash*, which can receive $\widetilde{\mathcal{O}}(\log W)$ blocks. To each block w stored in the ORAM (we assimilate a block with the corresponding keyword w) is associated a uniformly random leaf l_w of the tree. At all stages of the algorithm, Path ORAM maintains the invariant that block w is stored in one of the nodes on the path from the root to l_w, or in the stash. In the standard version of Path ORAM, in order to access a block w, the client reads the entire path $P(l_w)$ from the root to l_w, plus the stash. Afterwards, l_w is updated to a fresh uniformly random leaf l'_w, and an *eviction* process is performed.

The eviction process takes all blocks currently stored in the path $P(l_w)$ and in the stash, including block w, and stores each of those blocks as low as possible along $P(l_w)$ (viewing the root as the top of the tree). Recall that a block w' must be stored above the leaf $l_{w'}$, in order to maintain the desired invariant. Ideally, block w' is stored in the lowest node along $P(l_w)$ that lies above $l_{w'}$. However, it may be the case that more than Z blocks would be stored in that node; If so, overflowing blocks are pushed to the parent node. If this happens at the root node, overflowing blocks are pushed to the stash. To sum up, each block w' is stored "as low as possible" along $P(l_w)$, under the constraint that it must remain above $l_{w'}$, and that no node can receive more than Z blocks.

The analysis of Path ORAM is quite involved, but shows that the scheme is correct (no overflow occurs), except with negligible probability. Path ORAM fully hides from the server which block is accessed. The price to pay is that each access requires downloading an entire path plus the stash, which amounts to $\widetilde{\mathcal{O}}(\log W)$ blocks, performing some computation, and reuploading.

To reduce that cost, IO-DSSE proposes the following optimization. Like most SSE schemes, IO-DSSE search operations reveal which keyword is searched, regardless of the ORAM component. Thus, hiding which block is accessed during *reads* is overkill. Following that idea, IO-DSSE performs two optimizations. First, it introduces a *deferred read* system, whereby reads are buffered, and evictions are only performed once for a batch of successive reads. That optimization poses no problem, and is unrelated to the attack. Second, for each keyword w, IO-DDSE stores in a separate position map, not only the leaf l_w associated to the keyword, but also the level h_w of the tree (or the stash) where the block is actually stored. That way, when reading block w, the client can retrieve l_w and h_w from the position map, and query a single tree node: namely the node at level h_w along the path $P(l_w)$. This saves a factor $\log W$ during reads. In their experiments, [29] consider 350 searchable keywords. Even with this very modest number of keywords, the previous optimization saves as factor more than 8, which is significant in practice (on the English Wikipedia database, the same improvement would save a factor more than 20).

Unfortunately, the latter improvement introduces a security flaw. The security claim of IO-DSSE is that, during updates, the server only learns whether the incomplete block associated to the currently updated keyword becomes full, and nothing else. (That information is leaked because the newly full block will be removed from the Path ORAM instance, and pushed to a separate SSE scheme.) During reads, IO-DSSE claims that the server only learns at which points in time the currently searched keyword was previously updated.

The point of this section is to show that this security claim is incorrect. More precisely, we will show that, when the client performs a series of updates followed by a search, it is possible for the server to infer information about whether the previous updates were on pairwise distinct keywords, or on the same keyword, even if the following search is on an unrelated keyword. The issue is subtle, but relies on the fact that the level h_w of a given keyword depends on the pattern of previous updates, even if they are on other keywords.

A.1 A simple attack

Let $w_1, \ldots, w_6$ be pairwise distinct keywords. In scenario (1), the client updates keyword w_1, w_2, …, w_6 once each, then searches keyword w_1. In scenario (2), the client updates keyword w_1, then updates w_2 five times, then searches w_1. Assume $p \geq 6$, so that no full block is created (the practical experiments of IO-DSSE set $p = 500$). Then in both scenarios, the leakage function is the same: the client performs six updates, then one search on the keyword that was updated first. Hence, according to the security claim of IO-DSSE, the server should not be able to distinguish those two scenarios.

We present an attack that allows the server to distinguish the two scenarios with non-negligible probability. The attack is simple: the server observes the level h_{w_1} of w_1 during the final search. If h_{w_1} lies in the stash, then the server guesses that the client performed scenario (1). Otherwise, the server guesses that the client performed scenario (2). We now show that this attack has a constant distinguishing advantage.

In scenario (2), only two blocks can be in the root node, namely blocks w_1 and w_2. Since the root can contain up to $Z = 5 > 2$ blocks, it is not possible that block w_1 is pushed to the stash. Hence, in scenario (2), it is never the case that h_{w_1} is at the stash level. In scenario (1), on the other hand, with probability 2^{-6}, the leaves associated to all six keywords *before they are updated* are below the *left* child of the root; and with probability 2^{-6} again, the leaves associated to all six keywords *after they have been updated* are below the *right* child of the root. If both events occur, then all six keywords need to be stored in the root, which means that one of the keyword must be pushed to the stash. With probability 1/6, keyword w_1 is pushed to the stash (the scheme does not specify how to choose which block is pushed in case of overflow; however, by linearity of expectation, it must be the case that one of the six blocks is pushed with probability at least 1/6; without loss of generality, we assume it is w_1). Hence, with probability $C \geq 1/6 \cdot 2^{-12}$, h_{w_1}

will be at the stash level. In conclusion, the server can distinguish the two scenarios with constant advantage.

A generalized attack. we sketch how to improve the simple attack on IO-DSSE from Appendix A. Further, we highlight the flaw in the security proof of IO-DSSE. Lastly, we discuss how to repair the flaw. This seems to inherently incur a large cost in I/O efficiency.

The distribution of blocks inside Path ORAM is quite difficult to analyze in general. For that reason, we have focused on a simple attack that is easy to analyze. Stronger attacks are certainly possible, at the cost of a more intricate analysis. We note in particular that there is no need to exploit the stash in the attack, and observing levels of items close to the leaves can also lead to distinguishers. As an illustration of that point, consider the following attack. Split keywords into two sets W_1 and W_2 of respective sizes $|W_1| = \min(p-1, W-2)$ and $|W_2| = W - |W_1| - 1$, plus a special keyword w not in W_1 or W_2. In scenario (1), we update $|W_1|$ times the keyword w. During this process, for each keyword $w' \in W_1 \cup W_2$, the associated leaf $l_{w'}$ remains unchanged. Hence, as we make repeated accesses to w, each block w' for $w' \in W_1 \cup W_2$ will be progressively pushed down towards its associated leaf $l_{w'}$. However, since keywords in W_1 and W_2 have to compete for space at the leaf level, it may be the case that some of them cannot fit in the leaves, no matter how many accesses to w we perform.

In scenario (2), instead of repeatedly updating w, we update all keywords in W_1 once. Each time a keyword in W_1 is accessed, its associated leaf changes, and the block must begin a new journey down to its associated leaf. As a result, intuitively, the levels associated to keywords in W_1 will be higher on average in scenario (2) than in scenario (1). This frees up more space in the leaves for keywords in W_2. Hence, the expectation of the level of a uniform keyword in W_2 should be lower in scenario (2). This provides a sketch for a distinguishing attack that only looks at levels close to the leaves.

On the security proof. IO-DSSE comes with a security proof. Unfortunately, it is incomplete. In particular, the proof does not explain how the level h_w is chosen by the simulator. Our attack implies that this is not only a gap in the proof, but it is impossible for the simulator to simulate h_w based only the leakage function.

How to repair the flaw. It appears impossible to reveal information about h_w to an adversary without breaking security. As a consequence, we do not see a way to repair the security issue without undoing the second optimization introduced by IO-DSSE. That is, the position map should only store the leaf associated to a keyword, and not its level. When accessing the keyword, the whole path is read. Note that this is how Tree ORAM schemes normally operate. Undoing the optimization incurs a $\log W$ cost in I/O efficiency, since all blocks along a path must be read.

B More Related Work and SSE Research Directions

Searchable Symmetric Encryption was introduced by Song *et al.* at Security and Privacy 2001 [35]. Since then, the area has developed in several different directions. Because of the breadth of literature on SSE, we only draw attention here to a few important branches in the current research landscape. Insofar as SSE is a trade-off between functionality, security, and efficiency, research in the area can be roughly divided into three avenues, one for each component of the trade-off.

Works that expand functionality include SSE for boolean queries [14], range queries [33], or even subsets of SQL [26]. Works that deal with security include attacks, and efforts to reduce leakage in response to those attacks. Most attacks against searchable encryption fall in the category of *leakage-abuse* attacks, a term coined in [12]. Leakage-abuse attacks do not contradict the security claims of a scheme, but show how the leakage allowed by the security model enables the server to reconstruct large parts of the database in certain settings [12, 24]. These attacks have motivated further works that reduce or suppress leakage [23], including forward-secure schemes [34, 11, 27, 21, 18].

Among works that mainly target efficiency, perhaps the most notable development in recent years is I/O efficiency. We refer the reader to the introduction for a presentation of the topic, and

the relevant literature. Beside the three research directions briefly outlined above, some works have attempted to propose complete solutions [33, 25].

Lastly, we remark that the idea of a fixed update schedule is used in other constructions. Notably, the schemes FIX in [1] and SWiSSSE in [25] use a fixed update schedule to obtain (variants of) forward security. We stress that the schedules of FIX and SWiSSSE are *not* compatible with I/O-efficiency: both schemes have worst-case page efficiency $\Omega(N/p)$, and this cost seems intrinsic to their approach. Both schemes also require client buffers of size $\Omega(N)$ in the worst case (namely, when one keyword matches a constant ratio of the database). In a different direction, multiple-choice processes have emerged as useful building blocks for SSE in recent years, and have been used in several prior works, including [3, 19, 30, 2].

C BigHermes: Additional Material

We provide an overview of the data structures used in $\mathsf{BigHermes}_0$ in Table 2.

Table 2 – Overview of the data structures used in $\mathsf{BigHermes}_1$ (see Algorithm 3).

Data Structure	Comments
Bins $B_{w_1}, \ldots, B_{w_W}$	Each bin B_w stores up to p identifiers matching keyword w
$\mathsf{CB}_{\mathsf{new}} : w \mapsto L$	Table that buffers for each keyword w fresh identifiers L matching w (up to W identifiers in total)
$\mathsf{CB}_{\mathsf{out}} : w \mapsto (L, L)$	Table that buffers for keyword w up to 2 identifier lists of size at most p
$\mathsf{CFP} : [1, W/p] \mapsto L$	Table that maps an index to either a list L of p identifiers or $\perp$
$T_{\mathsf{len}} : w \mapsto \ell$	Stores for keyword w the number ℓ of identifiers that match w.

The remainder of this section describes $\mathsf{BigHermes}_2$, the fully deamortized variant of BigHermes (Section 4).

Recall that $\mathsf{BigHermes}_1$ achieves worst-case sublogarithmic page efficiency, communication complexity, as well as server-side time and memory complexities. It also achieves sublogarithmic client-side time complexity, but only amortized over an epoch, since the last update of an epoch triggers an end-of-epoch computation that runs in time $\mathcal{O}(W)$ on the client. Although the computations are simple, this behavior may be undesirable, and one may wish for *worst-case* sublogarithmic time complexity on the client side. Since every other aspect of the scheme is already deamortized, this would result in a fully deamortized scheme. That is what we set out to do with $\mathsf{BigHermes}_2$.

Here, standard deamortization techniques suffice. The main new techniques underpinning BigHermes (SSE supporting dummy updates, and the idea of "deamortizing" a trivial ORAM) were already present in $\mathsf{BigHermes}_1$. That is why we have focused the presentation on $\mathsf{BigHermes}_1$, in both Algorithm 3 and Figure 1. Nevertheless, we now show that client-side computation can also be deamortized.

$\mathsf{BigHermes}_2$ makes use of a pipeline precomputation in two steps. For this, we require separate copies of $\mathsf{CB}_{\mathsf{new}}, \mathsf{CFP}$ and $\mathsf{CB}_{\mathsf{out}}$ for both steps. We denote by $\mathsf{CB}^{(i)}_{\mathsf{new}}, \mathsf{CFP}^{(i)}$ and $\mathsf{CB}^{(i)}_{\mathsf{out}}$ the copies of $\mathsf{CB}_{\mathsf{new}}, \mathsf{CFP}$ and $\mathsf{CB}_{\mathsf{out}}$ respectively for the step i. Additionally, we require a table T_{x} that stores for each keyword w the number of underfull sublists, *i.e.* sublists of type (1) or (2), during the current epoch; and a counter ctr that counts the total number of (full) sublists of type (2). The tables T_{len} and T_{x} are shared by both steps. Further, we assume that at the beginning of each epoch, a new permutation π is drawn, $\mathsf{CB}^{(1)}_{\mathsf{new}}, \mathsf{CFP}^{(1)}$ and $\mathsf{CB}^{(1)}_{\mathsf{out}}$ is copied to $\mathsf{CB}^{(2)}_{\mathsf{new}}, \mathsf{CFP}^{(2)}$ and $\mathsf{CB}^{(2)}_{\mathsf{out}}$ respectively, and that $\mathsf{CB}^{(1)}_{\mathsf{new}}, \mathsf{CFP}^{(1)}, \mathsf{CB}^{(1)}_{\mathsf{out}}, \mathsf{ctr}$ and T_{x} are reinitialized. The client then performs the following operations, for each pipeline step.

1. During the first step, the data structures are prepared such that in step 2, the client can directly write the content to the server. That is, each update, the new keyword-identifier pair (w, id) is added to the identifier list $\mathsf{CB}_{\mathsf{new}}[w]$, and $T_{\mathsf{len}}[w]$ is incremented. Then, the client checks whether the current identifier list $\mathsf{CB}_{\mathsf{new}}[w]$ is equal to the sublist of type (1), *i.e.* if $T_{\mathsf{x}}[w] = 0$

and $T_{\mathsf{len}}[w] = 0 \bmod p$. In that case, the sublist is complete and $T_{\mathsf{x}}[w]$ is incremented. Further, she sets $\mathsf{CB}_{\mathsf{out}}[w] \leftarrow \mathsf{CB}_{\mathsf{new}}[w]$ and empties $\mathsf{CB}_{\mathsf{new}}[w]$ thereafter.
If $T_{\mathsf{x}}[w] \neq 0$, she checks whether $|\mathsf{CB}_{\mathsf{new}}[w]| = p$, *i.e.* whether the sublist is of type (2). In that case, $T_{\mathsf{x}}[w]$ and ctr is incremented. Then, she sets $\mathsf{CFP}[\pi(\mathsf{ctr})] \leftarrow \mathsf{CB}_{\mathsf{new}}[w]$ and empties $\mathsf{CB}_{\mathsf{new}}[w]$ thereafter.
Note that at the end of an epoch, all full lists of size p are written to CFP in a random location. Further, for each keyword w, $\mathsf{CB}_{\mathsf{new}}[w]$ contains the underfull sublist of type (3) that will be written to B_w in the next step and $\mathsf{CB}_{\mathsf{out}}[w]$ contains the sublist of type (1) that completes the current list of B_w (if that is possible).
2. During the second step, the content of the data structures is written to the server as before. The only difference is that $\mathsf{CB}_{\mathsf{out}}$ only contains L_1, whereas $\mathsf{CB}_{\mathsf{new}}$ contains L_x (with the notation of Algorithm 3).

D SmallHermes

Recall that p is the page size, N is an upper bound on the total number of identifiers and W is an upper bound on the total number of distinct keywords. In this section, we assume $N \leq pW$. We detail our scheme $\mathsf{SmallHermes}$ (sketched in Section 5). Our construction $\mathsf{SmallHermes}$ builds on the page-efficient dynamic SSE scheme $\mathsf{LayeredSSE}$ [30]. While $\mathsf{LayeredSSE}$ is not forward secure, we show that with the techniques developed in this work, we can construct an oblivious update algorithm with $\mathcal{O}(W)$ client memory. Note that our technique can only be applied if $N \leq pW$. Our construction preserves the efficiency properties of $\mathsf{LayeredSSE}$, namely $\widetilde{\mathcal{O}}\left(\log\log\frac{N}{p}\right)$ page efficiency and constant storage efficiency, and uses only $\mathcal{O}(W)$ client storage.

We present the results as follows. First, we recall how $\mathsf{LayeredSSE}$ works and outline a simple amortized update procedure that it forward secure. This intermediate construction $\mathsf{SmallHermes}_0$ is presented in Appendix D.1. Next, we present $\mathsf{SmallHermes}_1$ version, which has deamortized communication. This version is also given in pseudo-code in Algorithm 4 and the used data structures are presented in Table 3. Then, we show how to deamortize both communication and computation via the construction $\mathsf{SmallHermes}_2$. Finally, we analyze the security and efficiency.

Table 3 – Overview of the data structures used in $\mathsf{SmallHermes}$ (see Algorithm 4).

Data Structure	Comments
Bins $B_1, \ldots, B_m$	Each bin stores up to $p \cdot b_{N,p}$ identifiers as in $\mathsf{LayeredSSE}$ with $b_{N,p} = \widetilde{\mathcal{O}}(\log\log(N/p))$
$T_{\mathsf{len}} : w \mapsto \ell$	Table that stores for each keyword w the number ℓ of matching identifiers
$\mathsf{CB}_{\mathsf{new}} : w \mapsto L$	Table that buffers for each keyword w fresh identifiers L matching w (up to W identifiers in total)
$\mathsf{CB}_{\mathsf{out}} : \gamma \mapsto L$	Table that buffers for the γ-th bin the identifiers (potentially matching different keywords) to be added to B_γ
$T_{\mathsf{load}} : (\gamma, \kappa) \mapsto n_{\gamma,\kappa}$	Stores for bin γ the number $n_{\gamma,\kappa}$ of sublists in layer κ

D.1 $\mathsf{SmallHermes}_0$: Amortized $\mathsf{SmallHermes}$

Recall that $\mathsf{LayeredSSE}$ stores identifiers in $m = o(N/p)$ encrypted bins according to $\mathsf{L2C}$. That is, each list L of (at most p) identifiers matching keyword w is mapped to two bins B_α, B_β with $\log\log\frac{N}{p}$ conceptual layers and capacity $\widetilde{\mathcal{O}}\left(p\log\log\frac{N}{p}\right)$. Then, L is stored in either B_α or B_β depending on the load of each bin at layer κ, where κ depends on the size of L. For a search on keyword w, the bins B_α and B_β are requested from the server. The client can read the matching identifiers from the bins after decryption. These bins are also retrieved for each update on keyword

w, and the new identifier is inserted in one of the bins according to L2C. When a list has more than p identifiers, it is split into sublists of size p which are treated independently as described above. This results in $\widetilde{\mathcal{O}}\left(\log\log\frac{N}{p}\right)$ page efficiency and $\mathcal{O}(1)$ storage efficiency. See [30] for more details.

The setup and search of $\mathsf{SmallHermes}_0$ are identical to setup and search of LayeredSSE. As updates of LayeredSSE are not forward secure, we now show how to adapt the update procedure such that it has *no* leakage. With $\mathcal{O}(W)$ client storage, the client can buffer $\mathcal{O}(W)$ fresh keyword-identifier pairs in a buffer $\mathsf{CB}_{\mathsf{new}}$. When $\mathsf{CB}_{\mathsf{new}}$ is full, she can download the entire encrypted database EDB from the server and perform the updates locally with amortized $\mathcal{O}(1)$ page efficiency, as $N \leq pW$. For this, she has to perform up to W insertions according to L2C operations. In the following, we show how to deamortize this simple approach. As the final variant $\mathsf{SmallHermes}_2$ is technically involved, we first present an intermediate variant $\mathsf{SmallHermes}_1$.

D.2 $\mathsf{SmallHermes}_1$: SmallHermes with Deamortized Communication

$\mathsf{SmallHermes}_1$ deamortizes the communication at the cost of $\mathcal{O}(W \log\log N)$ client storage. Note that the overhead can be avoided heuristically via the use weighted 2C or via more pre-computation (details follow later). Since each bin is of size $\widetilde{\mathcal{O}}\left(p\log\log\frac{N}{p}\right)$, we aim for $\widetilde{\mathcal{O}}\left(\log\log\frac{N}{p}\right)$ page efficiency and constant storage efficiency. We proceed as follows, assuming that per keyword, there are at most p matching identifiers.

The client accesses each bin in a fixed schedule and inserts new identifiers into the required bin, based on locally computed load information. The client stores an additional table T_{load} which maps a bin B_γ and layer κ to the number $n_{\gamma,\kappa} \leftarrow T_{\mathsf{load}}(\gamma,\kappa)$ of lists stored in B_γ at layer κ. [6] Whenever the client decides to store an identifier list in a bin, she also updates T_{load} accordingly. With T_{load}, the client can now directly decide locally where to insert each new identifier. Note that the client cannot download the corresponding bin directly to insert the identifier, as this would break forward security. But we can still leverage the load information of T_{load} with an additional identifier buffer $\mathsf{CB}_{\mathsf{out}}$.

Each update, the client inserts the new keyword-identifier pair into $\mathsf{CB}_{\mathsf{new}}$. After W updates $\mathsf{CB}_{\mathsf{new}}$ is filled and the client pre-computes the location of the new identifiers. That is, for all keywords w, the client pre-computes the index γ of bin B_γ in which to insert the list L_{new} of new identifiers matching w (that are buffered in $\mathsf{CB}_{\mathsf{new}}$). The list L_{new} is then moved into $\mathsf{CB}_{\mathsf{out}}[\gamma]$. Note that the index γ can be computed via T_{load}, if T_{load} is continuously updated throughout the pre-computation.

After this pre-computation, for each bin B_γ, the buffer $\mathsf{CB}_{\mathsf{out}}[\gamma]$ contains all identifiers from $\mathsf{CB}_{\mathsf{new}}$ to be inserted into bin B_γ. $\mathsf{CB}_{\mathsf{new}}$ can be emptied subsequently. During the next epoch, the client can download each bin B_γ via a fixed schedule and insert $\mathsf{CB}_{\mathsf{out}}[i]$ into B_i. For this, $\mathsf{SmallHermes}_1$ downloads bin B_i the i-th update operation of the epoch.

Note that a fresh identifier is written to the server after at most $2W$ update operations and inserted into the bin B_γ chosen according to L2C. Consequently, no bin overflows with overwhelming probability due the correctness of L2C. Also, during each update operation, at most one bin of size $p\mathcal{O}\left(\log\log\frac{N}{p}\right)$ is downloaded. As in LayeredSSE, exactly two bins are accessed during a Search. Thus, $\mathsf{SmallHermes}_1$ has $\mathcal{O}\left(\log\log\frac{N}{p}\right)$ page efficiency and $\mathcal{O}(1)$ storage efficiency.

Handling arbitrary list lengths. If there are more than p identifiers per keyword, we split the lists L of identifiers matching keyword w into full lists and (at most) one underfull list. Sublists with exactly p identifiers are called *full*, whereas sublists with less than p identifiers are referred to as *underfull*. In order to compute the correct bin for the new identifiers, the client also stores the length of L for each keyword in a table T_{len}. During an update, the client computes the size $r = T_{\mathsf{len}}[w] \bmod p$ of the underfull list on the server. L_{new} (defined as above) is split into $x = \left\lceil \frac{r+|L_{\mathsf{new}}|}{p} \right\rceil$ sublists as follows:

[6] The table T_{load} is the reason $\mathcal{O}(W \log\log N)$ client storage is required. Using some additional pre-computation, the client can compute T_{load} at only $\mathcal{O}(W)$ required positions. This retains $\mathcal{O}(W)$ client storage (see appendix D.3).

- L_1, the sublist that fills the underfull list on the server (if possible).
- $L_2, ..., L_{x-1}$, the full sublists of size p.
- L_x, the remaining underfull sublist (if any).

For each sublist L_i, we again compute the bin B_γ in which we need to insert L_i via T_{load} (according to $\mathsf{L2C}$) and insert L_i into $\mathsf{CB}_{\mathsf{out}}[\gamma]$. Note that during this process, we interpret L_1 as a list of size $|L_1| + r$, as it will complete the underfull sublist on the server, if possible. Also, T_{len} and T_{load} are updated accordingly throughout the pre-computation.

Algorithm 4 $\mathsf{SmallHermes}_1$

$\mathsf{SmallHermes}_1.\mathsf{Setup}(\mathsf{K}, N, \mathsf{DB})$

Set $B_1, ..., B_m \leftarrow \mathsf{L2C.Setup}(\{(w_i, \mathsf{DB}(w_i))\}_{i=1}^{W}, N/p)$
Fill bins $B_1, ..., B_m$ up to size $p \cdot b_{N,p}$ with zeroes
Set $B_i^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(B_i)$ for $i \in [1, m]$
$T_{\mathsf{len}}[w] \leftarrow \ell_i$ for all keywords w
Setup T_{load} according to load of $B_1, ..., B_m$
Set $\mathsf{cnt} \leftarrow 0$
return $\mathsf{EDB} = (B_1^{\mathsf{enc}}, ..., B_m^{\mathsf{enc}})$

$\mathsf{SmallHermes}_1.\mathsf{KeyGen}(1^\lambda)$

Sample $\mathsf{K}_{\mathsf{Enc}}$ for Enc with security parameter λ
return $K = \mathsf{K}_{\mathsf{Enc}}$

$\mathsf{SmallHermes}_1.\mathsf{Search}(\mathsf{K}, w; \mathsf{EDB})$

Client:

Set $\ell \leftarrow T_{\mathsf{len}}(w)$ and $x = \lceil \ell/p \rceil$
send w, x

Server:

Set $\alpha_i, \beta_i \leftarrow \mathsf{H}(w \mid\mid i)$ for $i \in [1, x]$
send $\{B_{\alpha_i}^{\mathsf{enc}}, B_{\beta_i}^{\mathsf{enc}}\}_{i=1}^{x}$

$\mathsf{SmallHermes}_1.\mathsf{Update}(\mathsf{K}, w, \mathsf{id}, \mathsf{add}; \mathsf{EDB})$

Client:

if $\mathsf{cnt} = p$ **then**
  for all keywords w **do**
    Set $L \leftarrow \mathsf{CB}_{\mathsf{new}}[w]$
    Set $r \leftarrow T_{\mathsf{len}}[w] \bmod p$
    Set $x \leftarrow \lceil (r + |L|)/p \rceil$
    Split L into x lists L_i such that L_1 has size at most $p - r$, L_x has size at most p, and the remaining lists have size p
    Precompte bin index γ_i of L_i via load information in T_{load} for all $i \in [1, x]$
    Set $\mathsf{CB}_{\mathsf{out}}[\gamma_i] \leftarrow \mathsf{CB}_{\mathsf{out}}[\gamma_i] \cup \{L_i\}$ for all $i \in [1, x]$
    Update load information in T_{load} accordingly
    Set $T_{\mathsf{len}} \leftarrow T_{\mathsf{len}}[w] + |L|$
  Empty $\mathsf{CB}_{\mathsf{new}}$ and set $\mathsf{cnt} = 0$
$\mathsf{CB}_{\mathsf{new}}[w] \leftarrow \mathsf{CB}_{\mathsf{new}}[w] \cup \{\mathsf{id}\}$
$\mathsf{cnt} \leftarrow \mathsf{cnt} + 1$
if $\mathsf{cnt} \leq m$ **then**
  send cnt

Server:

send $B_{\mathsf{cnt}}^{\mathsf{enc}}$

Client:

Set $B_{\mathsf{cnt}} \leftarrow \mathsf{Dec}_{\mathsf{K}_{\mathsf{Enc}}}(B_{\mathsf{cnt}}^{\mathsf{enc}})$
Insert $\mathsf{CB}_{\mathsf{out}}[\gamma_i]$ into B_{cnt}
send $B_{\mathsf{cnt}}^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(B_{\mathsf{cnt}})$

Server:

Update $B_{\mathsf{cnt}}^{\mathsf{enc}}$

D.3 $\mathsf{SmallHermes}_1$: Fully Deamortized SmallHermes

Here, we remove the requirement of $\mathcal{O}\left(W \log\log \frac{N}{p}\right)$ client storage and deamortize both the communication and computation of $\mathsf{SmallHermes}$. This optimization is possible, because we observe that for inserting the $\mathcal{O}(W)$ new identifiers of $\mathsf{CB}_{\mathsf{new}}$, we do not require the entire load information of $\mathsf{L2C}$. Only the load of bin layers in which a new identifier is inserted is required. Per epoch, each new identifier requires the load of some bin B_γ at some layer κ. There at most $2W$ such pairs (γ, κ). The load information of these pairs can be precomputed in a pipeline with three steps, where each step is performed during $\mathcal{O}(W)$ updates. In an additional fourth step, the client can finally fill $\mathsf{CB}_{\mathsf{out}}$ as before using the load information. During the next W updates, the content of the filled buffer $\mathsf{CB}_{\mathsf{out}}$ is pushed to the server as before. This approach naturally yields an update operation with at most $\mathcal{O}\left(p \log\log \frac{N}{p}\right)$ communication *and* computation.

Before we present each pipeline step, we fix some convention as all steps rely on the results of previous steps. We assume that there are independent copies of client data structures

$\mathsf{CB}_{\mathsf{new}}, \mathsf{CB}_{\mathsf{out}}, T_{\mathsf{load}}$ and T_{hash}, per pipeline step. We index a data structure ds from step i via $\mathsf{ds}^{(i)}$. Note that T_{len} is a table shared by all steps. The pipeline steps are executed in reverse order, *i.e.* step $i+1$ is executed before step i. We implicitly assume that the content of $\mathsf{ds}^{(i)}$ is copied to $\mathsf{ds}^{(i+1)}$ before execution of pipeline step i. The contents of all data structures of step i are emptied after the step is performed and the content was copied to step $i+1$. The 5 pipeline steps are as follows:

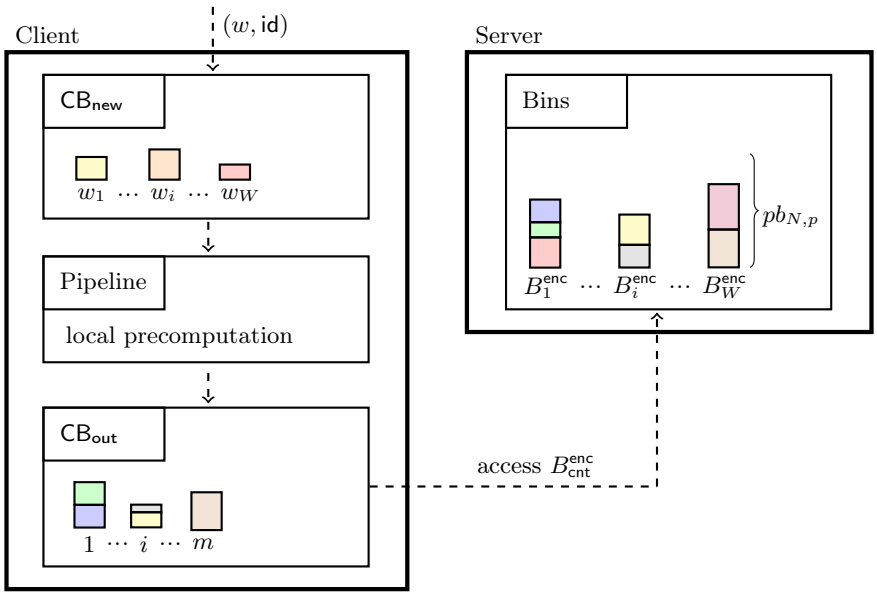

Fig. 4 – Sketch of the update schedule of $\mathsf{SmallHermes}_1$. (The data structures T_{len} and T_{load} are omitted for clarity.) Each update, the variable cnt is incremented and added keyword-identifier pairs are moved along the dotted lines. See Algorithm 4 for pseudo-code. Note that the $\mathsf{SmallHermes}_2$ shares the same structure, though the pipeline that pre-computes the final bin location of added identifiers is much more sophisticated.

1. In the first step, the client simply buffers the identifiers id in $\mathsf{CB}_{\mathsf{new}}^{(1)}$ as before, *i.e.* for each new keyword-identifier pair (w, id), the client adds id to the list $\mathsf{CB}_{\mathsf{new}}^{(1)}[w]$.
2. In the second step, the client splits for each keyword w the list of $L \leftarrow \mathsf{CB}_{\mathsf{new}}^{(2)}[w]$ of new identifiers matching w into sublists L_i (as explained above) and computes for each of these sublists the two bins it could be stored in. The result (along with auxiliary information) is stored in T_{hash}. Further, all required layers are marked in T_{load} with ∞ (and filled with the correct value in the following step).
   In more detail, during the k-th update operation, the client takes some keyword w for which $\mathsf{CB}_{\mathsf{new}}^{(2)}$ is not empty. She sets $r \leftarrow T_{\mathsf{len}}[w] \bmod p$ if w was considered for the first time this step and set $r \leftarrow 0$ otherwise [7]. Also, she sets $d \leftarrow \lfloor T_{\mathsf{len}}[w]/p \rfloor$. The client removes up to $p - r$ identifiers $L_{\mathsf{new}} = (\mathsf{id}_1, \ldots, \mathsf{id}_{\ell_{\mathsf{new}}})$ from $\mathsf{CB}_{\mathsf{new}}^{(2)}[w]$. Note that L_{new} corresponds to the i-th sublist L_i, if w was considered for the i-th time during this step. She sets $\kappa = \mathsf{layer}(\ell_{\mathsf{new}} + r)$. Then, she computes $\alpha, \beta \leftarrow \mathsf{H}(w \mid\mid (1+d))$ and stores $T_{\mathsf{hash}}^{(2)}[\beta] \leftarrow T_{\mathsf{hash}}^{(2)}[\beta] \mid\mid (L_{\mathsf{new}}, \kappa, \alpha)$ and $T_{\mathsf{hash}}^{(2)}[\alpha] \leftarrow T_{\mathsf{hash}}^{(2)}[\alpha] \mid\mid (L_{\mathsf{new}}, \kappa, \beta)$. Also, she sets $T_{\mathsf{load}}^{(2)}[\alpha, \kappa] \leftarrow 0$. Finally, she updates $T_{\mathsf{len}}[w] \leftarrow \ell_{\mathsf{new}} + T_{\mathsf{len}}[w]$.
3. In the third step, the client fetches the load for all required bin-layer pairs (γ, κ) and stores it in $T_{\mathsf{load}}^{(3)}$, *i.e.* all keys of $T_{\mathsf{load}}^{(3)}$ that are mapped to an integer n. The load $n_{\gamma,\kappa}$ will either be 0, if

[7] This can be decided using another table T_{flg} that matches a keyword w to a flag $b \in \{0,1\}$, initialized with 0 for each keyword. When the keyword w is considered, the flag T_{flg} is set to 1.

it was marked in step 2, or equal to the load of the bin at the given layer, if it was updated in step 4 (see step 4 for more details). The client proceeds as follows.
Whenever she fetches the bin B_{cnt} during an update operation, she retrieves the load $n_{\gamma,\kappa}$ of bin B_{cnt} at all layer κ. Then, *only* if $T^{(3)}_{\mathsf{load}}[\gamma,\kappa] = 0$, she stores the load in $T^{(3)}_{\mathsf{load}}[\gamma,\kappa] = n_{\gamma,\kappa}$.

4. In the fourth step, the client fills $\mathsf{CB}_{\mathsf{out}}$ with the new sublists. Also, she updates T_{load} from the previous level according to the updates in order to avoid inconsistencies. She proceeds as follows.
During the k-th update operation, she retrieves and removes some list and auxiliary information $(w, L_{\mathsf{new}}, \beta, \kappa)$ from $T^{(4)}_{\mathsf{hash}}[\alpha]$ for some bin B_α. Then, she computes the bin γ in which to store L_{new} according to L2C using the load information from $T^{(4)}_{\mathsf{load}}[\gamma,\kappa]$ and inserts the new sublist into $\mathsf{CB}^{(4)}_{\mathsf{out}}[\gamma]$. Further, she increments $T^{(4)}_{\mathsf{load}}[\gamma,\kappa]$. Note that this changes the load of the bin but we do not push L_{new} to the server in this step. Thus, the load information of the subsequent step 4 would be computed wrong for layers of bins that were updated in this step. In order to avoid these inconsistencies, the client further updates the load information of step 3 accordingly, *i.e.* sets $T^{(3)}_{\mathsf{load}}[\gamma,\kappa] = T^{(4)}_{\mathsf{load}}[\gamma,\kappa] + 1$ directly.
5. In the fifth step, the client writes the content of $\mathsf{CB}_{\mathsf{out}}$ to the server as in Algorithm 4.

Careful inspection shows that the pipeline pre-computation retains correctness. Also, the view of the server remains unchanged, thus $\mathsf{SmallHermes}_2$ remains semantically secure with the same leakage. Notably, even the client computation is de-amortized.

D.4 Heuristic Variant via Weighted 2C

In SmallHermes, we can replace L2C with weighted 2C. That is, weighted balls are inserted in the least loaded bin of two bins chosen at random, independent of layers. While we know of no non-trivial upper bound for this variant, heuristically it performs similar to L2C (see Figures 2 and 5). With this adaption, the entire load information can be kept on the client with $\mathcal{O}(W)$ storage. Thus, the $\mathsf{SmallHermes}_1$ has $\mathcal{O}(W)$ client storage using weighted 2C. Further, each update, we can directly decide in which bin to insert the added identifier, independent of layers, and update the load information accordingly. Recall that in $\mathsf{SmallHermes}_2$, we previously had to fetch *exactly* the right load information which resulted in complicated pipeline pre-computation. With weighted 2C, the client-side pipeline in $\mathsf{SmallHermes}_2$ can be heavily simplified, which reduces both the computation per update and the client storage. As this variant is only heuristically secure, we omit details.

D.5 Security

SmallHermes is forward secure with standard leakage as formalized in Theorem 5.

Theorem 5. *Let N be an upper bound on the size of the database, W be an upper bound on the number of keywords and let $p \leq N^{1-1/\log\log\lambda}$ be the page size. We model $\mathsf{H} : \{0,1\}^* \mapsto \{1,\ldots,m\}$ as a random oracle. Let $N \leq pW$. The scheme* SmallHermes *is correct and $\mathcal{L}^{\mathsf{fs}}$-adaptively semantically secure if* Enc *is* IND-CPA *secure and* PRF *is a secure pseudorandom function (for the preprocessing of w).*

Here, we sketch the proof of Theorem 5. We refer to Appendix H for a detailed proof.

The underlying L2C scheme is scaled such that up to N identifiers (organized into lists of size at most p) fit into the bins without overflowing (for any list distribution). SmallHermes inserts lists into the least loaded of two randomly chosen bins at the corresponding layer. Consequently, the bins are filled according to L2C. It follows from Lemma 2 that no bin overflows its capacity with overwhelming probability. Correctness follows immediately.

Semantic security also follows from the following facts. (1) The bins are encrypted and thus, only the upper bound N is leaked during setup. (2) During search, $2 \cdot \lceil \ell/p \rceil$ bins are accessed, where ℓ is the number of identifiers matching the searched keyword w. These bins are re-accessed if w is searched again but look random random to the server for the first search (as bin indices are the output of a hash function and w is random in the view of the server). Thus, a search leaks the search pattern and the number of sublists. (3) Updates are performed by accessing each bin via a fixed schedule (that solely depends on the number of updates).

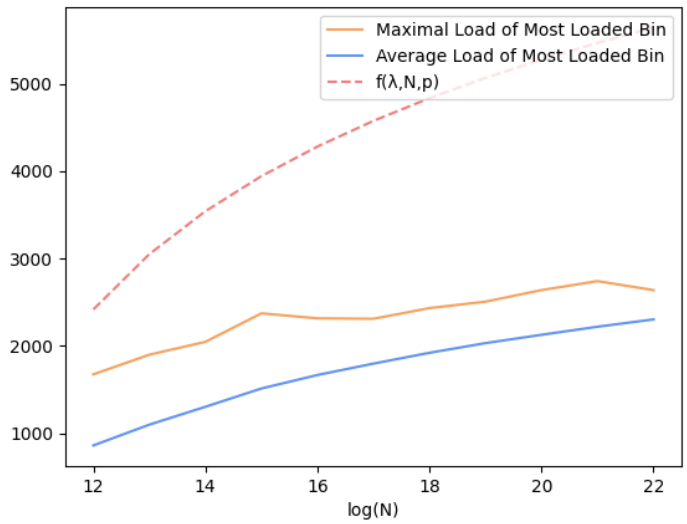

Fig. 5 – The loads of 2C for 10000 runs with balls of total weight N. The function $f(\lambda, N, p) = c \log\log\log(\lambda) \log\log(N/p)$ is the theoretical upper bound for the most loaded bin in L2C with constant $c = 2$ and $\lambda = 128$. The weights $\{w_i\}$ are chosen at random in $[1, 512]$.

D.6 Efficiency

Each data structure on the client side requires at most $\mathcal{O}(W)$ storage. As there are only a constant number of data structures and pipeline steps, the total client storage is $\mathcal{O}(W)$. At most one bin is downloaded each update and each search on keyword w, exactly $\lceil \ell/p \rceil$ bins are retrieved, where ℓ is the length of the list of identifiers matching keyword w. As each bin of size $\widetilde{\mathcal{O}}\left(p \log\log \frac{N}{p}\right)$, the page efficiency is $\widetilde{\mathcal{O}}\left(\log\log \frac{N}{p}\right)$ and the server storage is $\mathcal{O}(N)$.

E Optimizations and Trade-offs

For ease of exposition, several design choices in Hermes have been made in favor of simplicity. Depending on the deployment scenario, various tradeoffs and optimizations are possible. We sketch a few in this section. These tradeoffs apply to both sub-schemes of Hermes, BigHermes and SmallHermes.

E.1 Round Trip Time

Hermes requires 1 RTT for searches, which is optimal for response-hiding SSE. Using standard techniques (piggy backing), it is straightforward to make Hermes optimal in RTT for updates. That is, the client performs each update over the course of multiple queries by stashing the update responses, and resuming the operation on the next query (either update or search).

E.2 Dynamic Server Storage

First, as defined, the Σ_{dum} component of Hermes requires allocating all memory upfront. If the database is intended to grow up to N elements, then $\mathcal{O}(N)$ memory must be allocated at setup time. In some use cases, that behavior may be undesirable. If so, the scheme can be modified so that its memory usage scales dynamically with the size of the database. First, setup an instance of Hermes with capacity k for some small k. Once that instance reaches full capacity, create a new instance with capacity $2k$, initialized with the content of the original instance. The original instance

is then deleted, and the pattern repeats. This basic technique has been studied in depth in the context of memory allocation algorithms, and can be further refined in various way, for example to deamortize the cost of building the new instance; see *e.g.* [22]. We leave further optimizations along this line for future works.

E.3 Reducing Client Storage

Hermes requires $\mathcal{O}(W)$ client storage. This is optimal for efficient forward secure SSE schemes [10]. In most such schemes, for example [9, 11], the $\mathcal{O}(W)$ comes from the need to store a counter for each keyword, similar to T_{len} in Hermes. To improve memory efficiency, Hermes additionally buffers $\mathcal{O}(W)$ keyword-identifier pairs on the client. The client memory required for this can be reduced by a (constant) factor c in exchange for increasing update page efficiency by factor c. We sketch how to proceed for BigHermes. (The tradeoff can be applied to SmallHermes in a similar manner.) Each update, the client performs c updates of BigHermes at once. That is, she buffers the new keyword-identifier pair on the client, and instead of retrieving a single bin and performing two (dummy) updates to Σ, read c bins and perform $2c$ (dummy) updates to Σ. Now, each epoch lasts W/c updates and thus, only $\mathcal{O}(W)/c$ pairs need to be buffered on the client. While c bins are fetched each update (instead of one), note that the page efficiency of read queries is not impacted, as the worst-case load of the bins is not impacted.

E.4 On Leakage

In line with most SSE literature, Hermes does not specify how full documents are fetched on the server, once their identifier is retrieved. In SSE folklore, it is typically assumed that the documents are simply encrypted, and then queried from the server. That explains why most SSE schemes make no attempt to hide access pattern leakage (that is, which documents match a given query): this information will implicitly be revealed at a later stage, when the client queries the full documents. This leakage can sometimes be exploited by attacks, although this depends on the use case (see [7] for a detailed analysis). Some recent work has proposed to obfuscate access pattern leakage at the expense of efficiency [25]. While such techniques are out of scope, we note that Hermes naturally lends itself to such obfuscation techniques, in particular because it never requires the server to learn document identifiers in the clear (in fact, it is naturally response-hiding).

Finally, we note that Hermes realizes an encrypted multi-map. As such, beyond the direct application to keyword search, it can be used in any application that relies on encrypted multi-maps, such as the graph search algorithms from [16].

F Security Proof of Dummy(Σ)

Here, we prove Theorem 1. We show that $\mathsf{Dummy}(\Sigma)$ is secure, if Σ is suitable and forward secure. Note that correctness follows from Lemma 1 and the proof is straight-forward.

Proof. Let Sim denote the simulator and $\mathcal{A}$ an arbitrary honest-but-curious PPT adversary. Further, let Sim_Σ be a simulator for Σ with leakage $\mathcal{L}^{\mathsf{fs}}$. Initially, Sim receives $\mathcal{L}_{\mathsf{Stp}}(\mathsf{DB}, N, D, W) = (N, D, W)$ and later, an series of search and update queries with input $\mathcal{L}_{\mathsf{Srch}}(w) = (\mathsf{qp}, \ell)$ and $\mathcal{L}_{\mathsf{Updt}}(op, w, \mathsf{id}) = \perp$ respectively. First, Sim generates an encryption key $\mathsf{K}'_{\mathsf{Enc}}$ and initializes $m = N/\widetilde{\mathcal{O}}(\log\log N)$ bins $B_1, \ldots, B_m$ zeroed out up to size $\widetilde{\mathcal{O}}(\log\log N)$. Next, she sets $\mathsf{EDB}_\Sigma \leftarrow \mathsf{Sim}_\Sigma(N, W)$ and outputs $\mathsf{EDB}' = (\mathsf{Enc}_{\mathsf{K}'_{\mathsf{Enc}}}(B_1), \ldots, \mathsf{Enc}_{\mathsf{K}'_{\mathsf{Enc}}}(B_m))$. Next, Sim simulates the search and update queries.

For search queries, Sim receives the query pattern qp and the length ℓ of the identifier list matching the searched keyword. Sim outputs $\tau \leftarrow \mathsf{Sim}_\Sigma(\mathsf{qp}, \ell)$. For update queries, Sim receives no input. Sim sets $(k, v) \leftarrow \mathsf{Sim}_\Sigma(\perp)$ and outputs k. We now show that the real game is indistinguishable from the ideal game. For this, we define five hybrid games.

– Hybrid 0 is identical to the real game.

– Hybrid 1 is the same as Hybrid 0 raises a flag flag when a bin overflows its capacity during setup or update. As the client never inserts an identifier if the bin were to overflow, the probability of a flag being raised is 0. Thus, Hybrid 1 and Hybrid 0 are indistinguishable.
– Hybrid 2 is the same as Hybrid 1 except in Setup, the encrypted database EDB replaced with EDB', and in Update, the client sends back fresh encryptions of $\log\log(m)$ zeros. Since Enc is IND-CPA secure, it follows that the advantage of an adversary trying to distinguish Hybrid 2 from Hybrid 1 is negligible.
– Hybrid 3 is given in Algorithm 5 and is the same as Hybrid 2 except in Setup, the client performs the setup of Σ with DB directly instead of performing the updates locally. As Σ is suitable, Hybrid 3 and Hybrid 2 are identically distributed and thus indistinguishable.
– Hybrid 4 is given in Algorithm 6 and is the same as Hybrid 3, except the scheme Σ is simulated via Sim_Σ. Hybrid 4 and Hybrid 3 are indistinguishable, as Σ is $\mathcal{L}^{\mathsf{fs}}$-adaptively secure.
– Hybrid 5 is identical to the ideal game. Hybrid 5 and Hybrid 4 are identically distributed and thus, indistinguishable. □

Algorithm 5 Hybrid 3

$\mathsf{Dummy}(\Sigma).\mathsf{Setup}(\mathsf{K}, N, W, \mathsf{DB})$

1: Set $N_\Sigma \leftarrow N + D$ and $W_\Sigma \leftarrow W + 1$
2: Generate random $\mathsf{K}'_{\mathsf{Enc}}$
3: Generate dummy keyword w_{dum} and identifier $\mathsf{id}_{\mathsf{dum}}$
4: Let $\mathsf{EDB}_\Sigma \leftarrow \mathsf{Dummy}(\Sigma).\mathsf{Setup}(\mathsf{K}, N_\Sigma, W_\Sigma, \mathsf{DB})$
5: Initialize $m = N/\widetilde{\mathcal{O}}(\log\log N)$ bins $B_1, \ldots, B_m$ containing $\widetilde{\mathcal{O}}(\log\log N)$ zeros
6: **return** output $\mathsf{EDB} = (\mathsf{Enc}_{\mathsf{K}'_{\mathsf{Enc}}}(B_i))_{i=1}^m$

$\mathsf{Dummy}(\Sigma).\mathsf{Search}(\mathsf{K}, w; \mathsf{EDB})$

Client:
1: Set $\tau \leftarrow \Sigma.\mathsf{Search}_C(\mathsf{K}, w)$
2: **send** τ

$\mathsf{Dummy}(\Sigma).\mathsf{Update}(\mathsf{K}, (w, \mathsf{id}), \mathsf{op}; \mathsf{EDB})$

Client:
1: Set $(k, v) \leftarrow \Sigma.\mathsf{Update}_C(\mathsf{K}, (w, \mathsf{id}), \mathsf{add})$
2: **send** k

Client:
1: Receive $B^{\mathsf{enc}}_\alpha, B^{\mathsf{enc}}_\beta$
2: **send** reencrypted B_α, B_β

$\mathsf{Dummy}(\Sigma).\mathsf{DummyUpdate}(\mathsf{K}; \mathsf{EDB})$

Client:
1: Set $(k, v) \leftarrow \Sigma.\mathsf{Update}_C(\mathsf{K}, (w_{\mathsf{dum}}, \mathsf{id}_{\mathsf{dum}}), \mathsf{add})$
2: **send** k

Client:
1: Receive $B^{\mathsf{enc}}_\alpha, B^{\mathsf{enc}}_\beta$
2: **send** reencrypted B_α, B_β

Algorithm 6 Hybrid 4

$\mathsf{Setup}(\mathcal{L}_{\mathsf{Stp}}(N, W, \mathsf{DB}))$

1: Set $N_\Sigma \leftarrow N + D$ and $W_\Sigma \leftarrow W + 1$
2: Generate random $\mathsf{K}'_{\mathsf{Enc}}$
3: Let $\mathsf{EDB}_\Sigma \leftarrow \mathsf{Sim}_\Sigma(\mathcal{L}^{\mathsf{fs}}_{\mathsf{Stp}}(N_\Sigma, W_\Sigma, \emptyset))$
4: Initialize m empty bins $B_1, \ldots, B_m$
5: Fill each bin up to capacity $\widetilde{\mathcal{O}}(\log\log N)$ with zeros
6: **return** output $\mathsf{EDB} = (\mathsf{Enc}_{\mathsf{K}'_{\mathsf{Enc}}}(B_i))_{i=1}^m$

$\mathsf{Search}_C(\mathcal{L}_{\mathsf{Srch}}(w))$

Client:
1: Simulate $\tau \leftarrow \mathsf{Sim}_\Sigma(\mathcal{L}_{\mathsf{Srch}}(w))$
2: **send** τ

$\mathsf{Update}_C(\mathcal{L}_{\mathsf{Updt}}(\mathsf{op}, w, \mathsf{id}) = \perp)$

Client:
1: Simulate $(k, v) \leftarrow \mathsf{Sim}_\Sigma(\perp)$
2: **send** k

Client:
1: Receive $B^{\mathsf{enc}}_\alpha, B^{\mathsf{enc}}_\beta$
2: **send** reencrypted B_α, B_β

$\mathsf{DummyUpdate}_C()$

Client:
1: Simulate $(k, v) \leftarrow \mathsf{Sim}_\Sigma(\perp)$
2: **send** k

Client:
1: Receive $B^{\mathsf{enc}}_\alpha, B^{\mathsf{enc}}_\beta$
2: **send** reencrypted B_α, B_β

G Security Proof of BigHermes

Here, we prove Theorem 2. We show that BigHermes is secure with leakage $\mathcal{L}^{\mathsf{fs}}$ in detail. Correctness is tedious but straight-forward.

Proof. Let Sim denote the simulator and $\mathcal{A}$ an abitrary honest-but-curious PPT adversary. Further, let Sim_Σ be a simulator of Σ.

Initially, Sim receives $\mathcal{L}^{\mathsf{fs}}_{\mathsf{Stp}}(\mathsf{DB}, W, N) = (W, N)$ and later, an adaptive series of search and update queries with input $\mathcal{L}^{\mathsf{fs}}_{\mathsf{Srch}}(w_i) = (\mathsf{qp}, \ell_i)$ and $\mathcal{L}^{\mathsf{fs}}_{\mathsf{Updt}}(op_i, w_i, \mathsf{id}_i) = \perp$ respectively. First, Sim generates an encryption key $\mathsf{K}'_{\mathsf{Enc}}$ and initializes W bins $B_1, ..., B_W$ zeroed out up to size p. Then, Sim sets $\mathsf{EDB}'_\Sigma \leftarrow \mathsf{Sim}_\Sigma(W, N)$ and outputs $\mathsf{EDB}' = (\mathsf{Enc}_{\mathsf{K}'_{\mathsf{Enc}}}(B_1), ..., \mathsf{Enc}_{\mathsf{K}'_{\mathsf{Enc}}}(B_W), \mathsf{EDB}'_\Sigma)$. Further, Sim initializes a counter $\mathsf{cnt} = 0$ of the number of updates. Next, Sim simulates the search and update queries.

For search queries, Sim receives query pattern $\mathsf{qp} = (\mathsf{sp}, \mathsf{up})$ and the total number ℓ of identifiers matching the searched keyword. If the search pattern sp indicates that the keyword was already searched, Sim outputs the keyword w' from the previous query. Otherwise, Sim outputs a new uniformly random keyword w'. Also, Sim associates an index $\mu \in [1, W]$ to w' at random but distinct from indices of other keywords. Next, Sim simulates a search query of Σ. Recall that up is a bit vector that indicates for each update query whether it was an update on the searched keyword or not. As the updates on SmallHermes induce an altered update pattern on Σ, the simulator Sim needs to reconstruct the update pattern up_Σ of Σ from up. For this, she proceeds as follows for the k-th epoch:

She initializes an all-zero vector $\mathsf{up}_\Sigma = (0, ..., 0)$ of length $2\mathsf{cnt}$. If the coordinates of up_Σ were already computed for the k-th epoch during a previous search query for the current keyword, she sets up_Σ as before at these positions. If otherwise no search query was issued on the keyword after the k-th epoch, the corresponding coordinates of up_Σ were not yet set. Recall that during the last epoch, the updates on Σ depend on the updates during the previous epoch, and during the first epoch, only dummy updates are performed. She will set the coordinates as follows for $k > 1$.

Sim recomputes the number of added sublists x in the $(k-1)$-th epoch for the searched keyword. (Note based on up and ℓ, Sim can recompute the number of identifiers in the bin of the searched keyword at the beginning of the $(k-1)$-th epoch and the number of identifiers added during the $(k-1)$-th epoch. These values determine x. Then, $x-1$ sublists are then written to Σ during the k-th epoch due to preprocessing.) Then, if $x \geq 2$, she sets $\mathsf{up}_\Sigma[2(k+\mu)-1] \leftarrow 1$, where μ is the index of w'. Further, for all $i \in [1, x-2]$, she chooses some $j \in [1, W/p]$ that has not been chosen during the $(k-1)$-th epoch at random and sets $\mathsf{up}_\Sigma[2(k+j)] \leftarrow 1$. (If $2(k+\mu)-1$ or $2(k+j)$ are larger than $2\mathsf{cnt}$, then the client remembers the choice until the update happened and sets up accordingly for subsequent searches.) Note that for the first epoch, up_Σ remains zeroed out. Similarly, Sim computes the number n_{client} of identifiers matching the searched keyword that are still buffered on the client via ℓ and up, and sets sets $\ell_\Sigma \leftarrow \lfloor(\ell - n_{\mathsf{client}})/p\rfloor$. Finally, Sim invokes Sim_Σ with input qp_Σ and ℓ_Σ to simulate the search protocol of Σ.

In order to simulate an updates, Sim simply invokes Sim_Σ twice (with input $\perp$) to simulate the update protocol of Σ and increments cnt.

We now show that the real game is indistinguishable from the ideal game. For this, we define four hybrid games.

- Hybrid 0 is identical to the real game.
- Hybrid 1 is the same as Hybrid 0, except the simulated keywords w' are output during search. As we assume that w is the output of a PRF and thus indistinguishable from random, Hybrid 0 and Hybrid 1 are (computationally) indistinguishable.
- Hybrid 2 is the same as Hybrid 1, except the updates on Σ are induced by up_Σ. That is, updates for a given keyword w' on Σ are performed each u-th update operation, where $\mathsf{up}_\Sigma[u] = 1$ (but with the real identifiers). Note that by construction, for each such update on Σ, there is a full list of identifiers that was inserted during the previous epoch. Hybrid 2 is identically distributed to Hybrid 1, as the π is a random permutation.
- Hybrid 2 is the same as Hybird 1, except Σ is simulated. That is, EDB_Σ is replaced with the simulated EDB'_Σ, the search queries are simulated with input ℓ_Σ and up_Σ, and the updates

on Σ are simulated with input $\perp$. Hybrid 2 and Hybrid 1 are indistinguishable based on the security of Σ.
- Hybrid 3 is the same as Hybrid 2, except the bins are zeroed out. That is, EDB is replaced fully with EDB' and received bins are only reencrypted during updates. Since Enc is $\mathsf{IND\text{-}CPA}$ secure, Hybrid 3 and Hybrid 2 are indistinguishable.
- Hybrid 4 is identical to the ideal game. Hybrid 4 and Hybrid 3 are identically distributed and thus, indistinguishable. □

H Security Proof of SmallHermes

Here, we prove Theorem 5 (which is sketched in Theorem 3). We show that $\mathsf{SmallHermes}$ is secure with leakage $\mathcal{L}^{\mathsf{fs}}$ in detail. Correctness is tedious but straight-forward.

Proof. Let Sim denote the simulator and $\mathcal{A}$ an arbitrary honest-but-curious PPT adversary. Initially, Sim receives $\mathcal{L}^{\mathsf{fs}}_{\mathsf{Stp}}(\mathsf{DB}, W, N) = (W, N)$ and later, an series of search and update queries with input $\mathcal{L}^{\mathsf{fs}}_{\mathsf{Srch}}(w_i) = (\mathsf{qp}, \ell_i)$ and $\mathcal{L}^{\mathsf{fs}}_{\mathsf{Updt}}(op_i, w_i, \mathsf{id}_i) = \perp$ respectively. First, Sim generates an encryption key $\mathsf{K}'_{\mathsf{Enc}}$ and initializes $m = N/(p \cdot b_{N,p})$ bins $B_1, \ldots, B_m$ zeroed out up to size $p \cdot b_{N,p}$. Then, Sim outputs $\mathsf{EDB}' = (\mathsf{Enc}_{\mathsf{K}'_{\mathsf{Enc}}}(B_1), \ldots, \mathsf{Enc}_{\mathsf{K}'_{\mathsf{Enc}}}(B_m))$. Further, Sim initializes a counter $\mathsf{cnt} = 0$. Next, Sim simulates the search and update queries.

For search queries, Sim receives query pattern qp and the length ℓ of the searched identifier list. If the query pattern qp indicates that the keyword was already searched, Sim outputs the keyword w' from the previous query. Otherwise, Sim outputs a new uniformly random keyword w' that has not been output during a search query yet. In addition, Sim forwards $x = \lceil \ell/p \rceil$ to the $\mathcal{A}$.

For update queries, Sim receives no input. Sim simply increments cnt, sends cnt to the $\mathcal{A}$, re-encrypts the received bin and sends it back to to $\mathcal{A}$.

We now show that the real game is indistinguishable from the ideal game. For this, we define three hybrid games.

- Hybrid 0 is identical to the real game.
- Hybrid 1 is the same as Hybrid 0, except the simulated keywords w' are output. As we assume that w is the output of a PRF and thus indistinguishable from random, Hybrid 0 and Hybrid 1 are (computationally) indistinguishable.
- Hybrid 2 is the same as Hybrid 1, except the encrypted database EDB is replaced with EDB' and whenever the simulator receives a bin, she simply sends back re-encrypted bins. Since Enc is $\mathsf{IND\text{-}CPA}$ secure (and bins always have size $p \cdot b_{N,p}$), it follows that Hybrid 1 and Hybrid 2 are indistinguishable.
- Hybrid 3 is identical to the ideal game. Hybrid 3 and Hybrid 2 are identically distributed and thus, indistinguishable. □