Page-efficient Encrypted Multi-Maps: New Techniques for Optimal Search Bandwidth

Francesca Falzon[1⋆[0000−0001−8415−6237]], Zichen Gui[2[0009−0008−2977−8517]], and Michael Reichle[3⋆[0000−0002−3498−5472]]

[1] Lucerne University of Applied Sciences and Arts, Switzerland, francesca.falzon@hslu.ch
[2] University of Georgia, Georgia, USA, zichen.gui@uga.edu
[3] INSAIT, Sofia University "St. Kliment Ohridski", Bulgaria michael.reichle@insait.ai

Abstract. Encrypted multi-maps (EMMs) allow a client to outsource a multi-map to an untrusted server and then later retrieve the values corresponding to a queried label. They are a core building block for various applications such as encrypted cloud storage and searchable encryption. One important metric of EMMs is memory-efficiency: most schemes incur many random memory accesses per search query, leading to larger overhead compared to plaintext queries. Memory-efficient EMMs reduce random accesses but, in most known solutions, this comes at the cost of higher query bandwidth.
This work focuses on EMMs run on SSDs and we construct two page-efficient schemes—one static and one dynamic—both with optimal search bandwidth. Our static scheme achieves $\widetilde{\mathcal{O}}(\log N/p)$ page-efficiency and $\mathcal{O}(1)$ client storage, where N denotes the size of the EMM and p the SSD's page size. Our dynamic scheme achieves forward and backward privacy with $\widetilde{\mathcal{O}}(\log N/p)$ page-efficiency and $\mathcal{O}(M)$ client storage, where M denotes the number of labels. Among schemes with optimal server storage, these are the first to combine optimal bandwidth with good page-efficiency, saving up to $\mathcal{O}(p)$ and $\widetilde{\mathcal{O}}(p \log\log(N/p))$ bandwidth over the state-of-the-art static and dynamic schemes, respectively. Our implementation on real-world data shows strong practical performance.

Keywords: Dynamic searchable encryption · Encrypted multi-map · Encrypted database · Page efficiency

1 Introduction

An *Encrypted multi-map* (EMM) [14, 31] enables a client to securely outsource a multi-map MM (*i.e.*, a mapping from labels to sets of values) to an untrusted server and later privately query the multi-map for a particular label. We denote by N and M the number of label-value pairs and labels, respectively. Because EMMs are built using fast symmetric-key primitives, they provide a practical and scalable solution for secure cloud storage. For example, EMMs are a key building block of searchable encryption (SSE) (*e.g.*, [6, 9, 10, 15, 24, 28, 33, 34, 39, 42]) and often form the core of the SSE index-based structure.[4] Other privacy-preserving applications of EMMs include encrypted relational databases [11, 30], encrypted graphs [13, 20], private range search [18, 23], and encrypted gun registries [32].

Dynamic EMMs and Leakage. While much research has focused on *static* SSE and EMM schemes, for many such practical applications, it is often desirable for the EMM to be *dynamic* and support updates to the underlying multi-map. The efficiency of both static and dynamic EMMs, however, comes at a cost of some small and well-defined information about the queries or underlying dataset being leaked to the server: this information is captured in the form of a leakage function. Security of the EMM ensures that the server only learns the exact information captured by the leakage function.

An EMM scheme's leakage function is composed of *setup leakage*, *query leakage*, and—in the case of a dynamic scheme—*update leakage*. Setup leakage is the information revealed prior to query execution and comprises the size of the index (or some upper bound on the index size). When a search query is issued, EMM schemes typically leak whether the same query has been made before

⋆ Work was completed while at ETH Zurich, Switzerland.
[4] In this work, we use the notions of SSE and EMM interchangeably when referring to prior works.

(the *query pattern*) and the size of the response (the *volume*). For dynamic EMMs, additional security properties are desired due to known file-injection attacks against prior schemes *e.g.*, [47]. In particular, it is desirable that updates reveal no information about which label was modified, a property known as *forward privacy* [46]. Another important goal is that no information about the deleted values during query processing is leaked after a deletion, a property called *backward privacy* [8].

Memory-efficiency for EMMs. EMMs are often built by mapping encrypted label-value pairs to a random location in a hash table (*e.g.*, [7, 10, 15]). The locations for a given label ℓ can only be determined by the client, however, the server can recompute these locations given a search token P for ℓ, often derived by a PRF evaluation by the client. The computational overhead of a search query is very low as it relies on efficient symmetric cryptographic operations such as encryption and PRF evaluations. However, as EMMs operate over large quantities of data, the random access of hash table locations becomes the main bottleneck of EMMs [12]. This behavior is visualized in Fig. 1a. Since only the locations storing the encrypted values matching a given label are accessed, non-memory-efficient EMMs often achieve optimal search bandwidth.

Given the above, Cash and Tessaro [12] introduce the notion of *locality* (the number of disjunct memory accesses) and *read efficiency* (the amount of data read beyond the result itself) which are good predictors for the memory-efficiency on HDDs. Later, Bossuat et al. [6] introduce the notion on *page efficiency* (the number of pages read beyond the minimal requirement) which is tailored to memory-efficiency on SSDs. The challenge in constructing memory-efficient EMMs is that values associated with the same label must be stored close together. This can be achieved, for instance, by storing all values matching the same label adjacent in memory. That is, the EMM is of the form $(\tilde{V}_{\ell_1}, ..., \tilde{V}_{\ell_M})$, where $\tilde{V}_{\ell_i}$ are the encrypted values matching the i-th label ℓ_i. Unfortunately, this simple solution violates the security requirements of encrypted multi-maps: querying for a label must not leak information about the distribution of values associated with other labels. However, in the above example, the locations across labels are correlated. For example, observe that the location of values matching ℓ_2 depends on the number of values matching ℓ_1.

Despite the apparent tension between security and memory-efficiency, there are many known constructions that achieve interesting tradeoffs [2, 3, 4, 6, 9, 17, 19, 38, 39, 40, 41]. However, *all* known constructions with constant storage efficiency (*i.e.*, encrypted EMMs of size $\mathcal{O}(N)$) incur a noticeable bandwidth overhead. Let us briefly examine why.

Most memory-efficient constructions leverage weighted balls-into-bins mechanisms, following the framework introduced in [2, 6]. Since we focus on page-efficient EMMs, we recall the framework from prior page-efficient EMM works [6, 39, 40, 41]. [5] At a high level, encrypted values $\tilde{V}_\ell$ matching label ℓ are stored together in contiguous, fixed-size memory regions called *bins*. Each bin corresponds to a memory page and the objective is to access as few bins as possible. For each label ℓ, the values $\tilde{V}_\ell$ are split into pages and each page of values—some of which may be incomplete—is stored in few bin(s) chosen at random. Since the bins are chosen at random, the bin accesses required to retrieve $\tilde{V}_\ell$ are independent of those for other labels. However, the locations of values within the bins may depend on the distribution of other labels, as each bin may store values matching multiple labels to maintain a constant storage overhead. Thus, on search, the entire bin is accessed to preserve security, which leads to bandwidth overhead proportional to the size of bins. In all known memory-efficient EMMs with constant storage efficiency, this leads to non-constant bandwidth overhead.

We thus ask the following question:

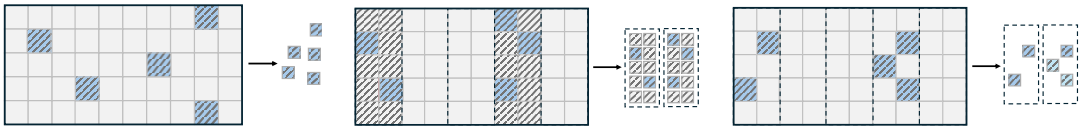
Can we build page-efficient EMMs without bandwidth overhead?

1.1 Our Contributions

We answer our research question affirmatively.

Two Bandwidth-and Memory-efficient Schemes. We propose two EMM schemes, S1C and D1C, with constant storage efficiency, $\widetilde{\mathcal{O}}(\log N/p)$ page efficiency (where p denotes the page size), and optimal search bandwidth (*i.e.*, on search, at most $\mathcal{O}(n)$ data is communicated for a plaintext result of size n). The schemes have different tradeoffs in terms of client storage and functionality:

[5] We give a more detailed overview of the framework in Section 1.1.



(a) A non-page-efficient EMM with random access, optimal bandwidth.

(b) A page-efficient EMM with bandwidth linear in the page size.

(c) Our solution for a page-efficient EMM with optimal bandwidth.

Fig. 1 – An overview of how EMM schemes differ with respect to memory access and bandwidth costs. Blue squares denote the addresses of the queried for records and shaded squares denote the addresses of records that are returned by the server to the client. Prior work falls into categories (a) and (b), and in this work, we present the first scheme that belongs to category (c).

– S1C is a static EMM that requires no client storage beyond key material. Its leakage profile matches that of standard EMMs: setup reveals the total number of values N, and search reveals the search pattern and query volume.
– D1C is a dynamic EMM with forward and backward privacy. Its setup and search leakage matches that of S1C. As with other forward-private EMMs achieving optimal search bandwidth (*e.g.*, [7, 8]), D1C requires $\mathcal{O}(M)$ client storage, where M is the number of labels.

At a high level, we organize memory accesses so that they resemble prior page-efficient EMMs (*cf.* Fig. 1b), while introducing new techniques that ensure we securely access and return only the matching values, thereby avoiding any additional bandwidth overhead (*cf.* Fig. 1c). We give a brief overview of these techniques below and compare our work with prior work in Table 1.

Table 1 – An overview of relevant EMM schemes with optimal server storage efficiency. Static schemes are given in the upper half. We consider dynamic schemes with forward and backward security in the bottom half. N denotes an upper bound on the multi-map size, M an upper bound on the number of labels, and p the page size. The "Bandwidth OH" column under "Search" reports the worst-case factor n_ℓ/n_s, where $n_\ell = |\mathsf{MM}[\ell]|$ is the plaintext response size and n_s the communication during search on label ℓ. The "Bandwidth OH" column under "Update" gives the per-update bandwidth. Note that non-page-efficient schemes require p accesses per page (*e.g.* PiBas [9], Diana [8]).

Scheme	**Persistent Storage**		**Search**		**Update**	
	Client	Server	Page Eff.	Bandwidth OH	Page Eff.	Bandwidth OH
PiBas [9]	$\mathcal{O}(1)$	$\mathcal{O}(N)$	$\mathcal{O}(p)$	$\mathcal{O}(1)$	-	-
1C [2]	$\mathcal{O}(1)$	$\mathcal{O}(N)$	$\widetilde{\mathcal{O}}(\log N)$	$\widetilde{\mathcal{O}}(\log N)$	-	-
2C [2]	$\mathcal{O}(1)$	$\mathcal{O}(N)$	$\widetilde{\mathcal{O}}(\log\log N)$	$\widetilde{\mathcal{O}}(\log\log N)$	-	-
Tethys [6]	$\mathcal{O}(p\log\lambda)$	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(p)$	-	-
S1C (Sec. 3)	$\mathcal{O}(1)$	$\mathcal{O}(N)$	$\widetilde{\mathcal{O}}\left(\log \frac{N}{p}\right)$	$\mathcal{O}(1)$	-	-
Diana [8]	$\mathcal{O}(M)$	$\mathcal{O}(N)$	$\mathcal{O}(p)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Hermes [40]	$\mathcal{O}(M)$	$\mathcal{O}(N)$	$\widetilde{\mathcal{O}}\left(\log\log \frac{N}{p}\right)$	$\widetilde{\mathcal{O}}\left(p\cdot\log\log \frac{N}{p}\right)$	$\widetilde{\mathcal{O}}\left(\log\log \frac{N}{p}\right)$	$\widetilde{\mathcal{O}}\left(p\cdot\log\log \frac{N}{p}\right)$
L-SD$_d$[1C] [41]	$\mathcal{O}(1)$	$\mathcal{O}(N)$	$\widetilde{\mathcal{O}}(\log N)$	$\widetilde{\mathcal{O}}(\log N)$	$\mathcal{O}\left(\log^2 N\right)$	$\mathcal{O}\left(\log^2 N\right)$
L-SD$_d$[2C] [41]	$\mathcal{O}(1)$	$\mathcal{O}(N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$	$\mathcal{O}\left(\log^3 N\right)$	$\mathcal{O}\left(\log^3 N\right)$
D1C (Sec. 5)	$\mathcal{O}(M)$	$\mathcal{O}(N)$	$\widetilde{\mathcal{O}}\left(\log \frac{N}{p}\right)$	$\mathcal{O}(1)$	$\widetilde{\mathcal{O}}\left(\log \frac{N}{p}\right)$	$\widetilde{\mathcal{O}}\left(p\cdot\log \frac{N}{p}\right)$

Our Techniques. Let p denote the page size, *i.e.*, the size of a contiguous memory segment on an SSD that can be accessed efficiently. In classical EMMs, values are placed in random memory locations for security [9]. Unfortunately, this leads to many random page accesses per query and

poor I/O performance (*cf.* Fig. 1a). Page-efficient EMMs address this issue by minimizing the number of distinct pages accessed per query.

Our work builds on the balls-into-bins framework for page-efficient EMMs introduced in [6], which adapts the framework of [2] to SSD-based storage. The server storage is partitioned into bins, and values matching the same label are placed into a small number of bins chosen according to a weighted hashing algorithm (*i.e.*, some balls-into-bins mechanism with weighted balls). [6] This mechanism ensures that bin locations are independent across labels, which is crucial for security. [7]

On a search for label ℓ, the bins associated with ℓ are retrieved. Since bins occupy contiguous memory, this access pattern is memory-efficient. The concrete memory efficiency is inherited from the properties of the balls-into-bins mechanism, in particular, the upper bound on the bin size and the number of bins selected per label. However, as observed in [2], the *exact* locations of values within a bin are not independent and therefore cannot be revealed to the server. As a result, prior page-efficient EMMs download entire bins and decrypt them locally, incurring significant bandwidth overhead (*cf.* Fig. 1b). This paradigm underlies essentially all existing page-efficient EMM constructions [4, 6, 17, 19, 39, 40, 41].

A central contribution of this work is showing that this bandwidth overhead is *not* inherent by introducing techniques that enable optimal bandwidth. We develop an oblivious access structure for bins that allows the server to retrieve exactly the matching values without revealing their locations, thereby achieving optimal bandwidth (*cf.* Fig. 1c).

A key tool in our work is the use of *one-choice allocation* (1C), a hashing technique which assigns each label to exactly one randomly chosen bin. Note that one-choice allocation has previously been used to achieve good page efficiency [2, 41]. However, as noted above, revealing the exact value locations is insecure, even when using 1C.

In this work, we leverage 1C's structure in a new way to provide the first security argument—to our knowledge—showing that revealing value locations within bins preserves security *if* additional measures are taken. In more detail, for other weighted hashing techniques such as two-choice allocation [2, 39, 45] or cuckoo hashing [6, 43], there is more than one choice of bins and the bin assignment depends on the multi-map's distribution. In contrast, for 1C the bin assignment is publicly known as there is only a single bin choice, so there is hope to reveal concrete locations within the bins without incurring additional leakage. However, doing this naively (*e.g.*, storing encrypted values in bins directly such as in [2, 41]) is insecure. Instead, we structure our bins such that these locations can be revealed safely (*i.e.*, each bin is a hash table with suitable properties). This insight underlies both our static construction and our EMM scheme with dummy updates, Dummy1C, which enables optimal-bandwidth dynamic EMMs. In Appendix C, we provide a detailed discussion on why our technique crucially relies on 1C allocation and fails with other allocation mechanisms. In the full version [21], we provide a detailed discussion on why our technique crucially relies on 1C allocation and fails with other allocation mechanisms.

Finally, we note that 1C allocation is (asymptotically) less efficient than some other allocation mechanisms used in literature to derive memory-efficient EMMs. Indeed, it leads to logarithmic page efficiency, whereas other works achieve doubly logarithmic or even constant page efficiency [6, 39, 40], at the cost of bandwidth overhead (*cf.* Table 1).

Instantiation and Evaluation. While our research question is motivated by theory, our construction also achieves strong practical efficiency. We implemented our schemes and our evaluation demonstrates excellent performance (see Section 6). We also compare our constructions to prior state-of-the-art EMMs for concrete data sets. Our analysis shows that we save up to 95x and 72x in bandwidth over static and dynamic page-efficient EMMs, while page efficiency remains comparable (see Appendix A for details). (see full version [21] for details).

Summary. Our contributions can be summarized as follows:

1. A new static EMM construction based on one-choice allocation that achieves good page efficiency and optimal bandwidth.

[6] In weighted hashing, balls of total weight at most n are placed into m bins. The bins are selected by a hash function. The goal is to minimize the maximal bin load.

[7] If a label matches more than p values, the values are partitioned into lists of size p and stored in separate bins. For exposition, we assume each label matches at most p values.

2. An EMM scheme with dummy updates, Dummy1C, achieving constant search bandwidth.
3. A novel security argument showing that revealing exact value locations within bins is safe under one-choice allocation.
4. A dynamic EMM construction combining the above techniques within the Hermes framework [40] to achieve forward and backward privacy.
5. An implementation and evaluation of our schemes to demonstrate their efficiency on real-world data.

2 Preliminaries

Notation. We follow the notation of [40]. We denote the security parameter as λ. For a probability distribution X, we write $x \leftarrow X$ for sampling a value x from X. For a set $\mathcal{X}$, we write $x \leftarrow \mathcal{X}$ for sampling uniformly at random from $\mathcal{X}$. A function $f(\lambda)$ is *negligible* in λ if it is $\mathcal{O}(\lambda^{-c})$ for every $c \in \mathbb{N}$, and we write $f = \mathsf{negl}(\lambda)$ for short. Similarly, f is overwhelming in λ if $f = 1 - \mathsf{negl}(\lambda)$.

For a protocol $\mathsf{prot} = (\mathsf{prot}_A, \mathsf{prot}_B)$ between two parties A and B, we denote by $\mathsf{prot_A}(\mathsf{opin}_A) \longleftrightarrow \mathsf{prot_B}(\mathsf{opin}_B)$ an execution of the protocol between A and B with input opin_A and opin_B, respectively. Sometimes, we write $\mathsf{prot}(\mathsf{opin}_A; \mathsf{opin}_B)$ if both executing parties can be inferred from context.

We primarily use hash tables and bins as data structures. For readability, we assume that tables mapping to integers are initialized with 0 entries and that bins are padded to a fixed capacity when it can be inferred from context.

2.1 Data Structures

In this work, we employ hash tables and bins frequently.

Hash Tables. A table T that is initialized for size N allows storing up to N key-value pairs (k, v). We write $v \leftarrow T[k]$ to retrieve the value v associated to k and we write $T[k] \leftarrow v$ to store v under key k. In general, when parties communicate T, we assume the party holding T sends (up to) N key-value pairs $(k_i, v_i)_{i \in [N]}$. The receiving party can structure T in such a way that retrieval and updates are efficient.

In Appendix E.2, we introduce a hash table variant that does not leak the keys k_i except on access of k_i. We employ different notation for such tables. In the full version [21], we introduce a hash table variant that does not leak the keys k_i except on access of k_i. We employ different notation for such tables.

Bins. A bin B of capacity cap is a segment of data of size cap that contains arbitrary information. We assume bins are always padded to full capacity cap.

2.2 Cryptographic Primitives

We employ pseudorandom functions (PRFs) and (symmetric) encryption schemes, and we prove security in the random oracle model. We give a brief overview of these concepts below, and refer to [35] for formal definitions.

PRF. A pseudorandom function (PRF) is a function F that maps keys $k \in \{0,1\}^\ell$ and inputs in $x \in \{0,1\}^n$ to values in $y \in \{0,1\}^m$. We write $y = F_k(x) := F(k, x)$ for short. We say that F is secure if for some $k \leftarrow \{0,1\}^\ell$, it is difficult to distinguish between F_k and a truly random function.

Encryption. An encryption scheme is a tuple of PPT algorithms $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$, where KeyGen outputs a secret key $\mathsf{K_{Enc}}$ that allows to encrypt and decrypt messages, *i.e.*, $c \leftarrow \mathsf{Enc}(\mathsf{K_{Enc}}, m)$ and $m = \mathsf{Dec}(\mathsf{K_{Enc}}, c)$. We write $c = \mathsf{Enc}_{\mathsf{K_{Enc}}}(m) := \mathsf{Enc}(\mathsf{K_{Enc}}, m)$ for short. We say that Enc is IND-CPA secure, if it is difficult to distinguish between encryptions of m_0 and m_1 of equal length.

Random Oracle Model. In the random oracle model, the adversary has oracle access to some hash function(s) $\mathcal{H}_1, ..., \mathcal{H}_n$. In the security game, these are modeled as programmable random functions.

2.3 One-choice Allocation

In one-choice allocation (1C), there are n balls of weight 1 thrown into m bins such that each ball is allocated to a single, random bin. In weighted 1C, each ball is associated with a weight $0 \leq w \leq 1$ and their total weight is at most n. For both variants, there are (either explicit or implicit) logarithmic upper bounds in the literature, *i.e.*, the most loaded bin has weight at most $\widetilde{\mathcal{O}}(\log n)$ except with negligible probability. We will employ weighted 1C allocation to place multi-map values into bins: a list L of up to p values is placed into a random bin (chosen by a hash function) and is interpreted as ball of weight L/p.

We recall the weighted one-choice (1C) allocation process. Roughly, an arbitrary number of balls $\{b_i\}$ of weight $w_i \in [0,1]_{\mathbb{R}}$ with total weight at most N are thrown into m bins at random. Let $\delta(\lambda) = \log\log\lambda$. A detailed description is given in Algorithm 1, where $\mathcal{H}$ denotes a random oracle mapping into $\mathbb{Z}_m$ and bid is a unique identifier for each ball.

Algorithm 1 One-Choice Allocation (1C)

1C.Setup(N)
1: Set $m \leftarrow \lceil N/(\delta(\lambda)\log(N))\rceil$
2: Initialize m empty bins $B_0, ..., B_{m-1}$
3: Return $B_0, ..., B_{m-1}$

1C.UpdateBall(bid, $w, B_0, ..., B_{m-1}$)
1: Receive bins $B_1, ..., B_m$, and ball bid with weight w
2: Set $\alpha \leftarrow \mathcal{H}(\mathsf{bid})$
3: **if** bid $\notin B_\alpha$ **then**
4: Insert bid into B_α
5: Update weight of bid to w in B_α

An $\widetilde{\mathcal{O}}(\log N)$ upper bound for the most loaded bin in (weighted) one-choice is implicit in [5, 39]. We recall the formal statement below. We give a standalone proof in Appendix B for completeness. We give a standalone proof in the full version [21] for completeness.

Lemma 1 (1C). *Let $N = \mathsf{poly}(\lambda)$, $\delta(\lambda) = \log\log\lambda$, and $m = \lceil N/(\delta(\lambda)\log(N))\rceil$. Assume $m = \Omega(\lambda)$. Then the most loaded bin of (weighted) 1C during the execution of a sequence of* UpdateBall *operations has load at most $\mathcal{O}(\delta(\lambda)\log N)$ except with negligible probability, if the total weight of the inserted balls does not exceed N.*

2.4 Encrypted Multi-maps

A multi-map $\mathsf{MM} = \{(\ell_i, (v_1, ..., v_{n_i}))\}_{i=1}^{L}$ is a set of label-value pairs with L total labels. We denote by $\mathsf{MM}[\ell_i] = (v_1, ..., v_{n_i})$ the list of values matching label ℓ_i. As is common in the literature, we assume that there is an upper bound M on the total number of labels $L \leq M$ and an upper bound N on the size of the multi-map $\sum_{i=1}^{L} |\mathsf{MM}[\ell_i]| \leq N$. We denote the label space as $\mathcal{L}_{\mathsf{MM}}$ and the value space as $\mathcal{V}_{\mathsf{MM}}$. We omit the subscript MM if clear by context. We write $|\mathcal{V}|_b$ and $|\mathcal{L}|_b$ for an upper bound on the bit-size of values and labels, respectively. Further, we denote by p the page size (which is treated as variable independent of the size of the database).

We recall the definition of a dynamic EMM scheme (DEMM) and related notions following [1, 14, 31, 40].

Definition 1. *A dynamic EMM scheme Σ is a tuple of two algorithms and two protocols* (KeyGen, Setup, Search, Update)*:*

- Σ.KeyGen(1^λ)*: a PPT algorithm that takes as input the security parameter λ and outputs client secret key* K*.*
- Σ.Setup(K, $N, M,$ MM)*: a PPT algorithm that takes as input client secret key* K*, upper bounds N on the multi-map size and M on the number of values, and a multi-map* MM*; it outputs an encrypted multi-map* EMM *and the client state* st*.*

– $\Sigma.\mathsf{Search}(\mathsf{K}, \ell, \mathsf{st}; \mathsf{EMM})$*: a protocol executed by the client and server. The client receives as input the secret key* K*, label* ℓ *and state* st*. The server receives as input the encrypted multi-map* EMM*. It outputs some data* d *and updated state* st' *for the client; it outputs updated encrypted multi-map* EMM' *for the server.*
– $\Sigma.\mathsf{Update}(\mathsf{K}, \ell, v, \mathsf{op}, \mathsf{st}; \mathsf{EMM})$*: a protocol executed by the client and server. The client receives as input the secret key* K*, a label-value pair* (ℓ, v)*, an operation* $\mathsf{op} \in \{\mathsf{del}, \mathsf{add}\}$ *and state* st*. The server receives as input the encrypted multi-map* EMM*. Outputs updated state* st' *for the client. Outputs updated encrypted multi-map* EMM' *for the server.*

Conventions. We use the following conventions.

– If the client state st is clear from context, we omit st and assume that it is stored and updated by the client.
– We sometimes omit the parameters MM and M in Setup if not they are not required (*e.g.*, if the scheme is forward-private and setup can be done interactively via Update with no additional leakage). If MM is omitted in Setup, we assume that $|\mathcal{V}|_b$ and $|\mathcal{L}|_b$ are still passed as input implicitly.
– All EMM schemes that support an "add" operation can be extended to support deletes with the generic framework of [7, 8] while maintaining efficiency (up to a factor 2). For readability, we present our constructions without deletions and assume that $\mathsf{op} = \mathsf{add}$ in Update.

Correctness and Security. We now define the properties of a correct and secure EMM scheme.

Correctness. Correctness ensures that when a label ℓ is queried for, the client obtains $\mathsf{MM}[\ell]$.

Definition 2 (Correctness). *An* EMM *scheme* Σ *is correct if for all databases* DB *and parameters* $M, N \in \mathbb{N}$*, keys* $\mathsf{K} \leftarrow \Sigma.\mathsf{KeyGen}(1^\lambda)$*, encrypted multi-maps* $\mathsf{EMM} \leftarrow \Sigma.\mathsf{Setup}(\mathsf{K}, N, M, \mathsf{MM})$ *and sequences of search, add, or delete queries* S*, the search protocol returns the correct result for all queries of the sequence if the size of the database remains at most* N *and at most* M *keywords are used.*

Security. Semantic security ensures that the server learns information quantified by a leakage function. Let $\mathcal{L} = (\mathcal{L}_{\mathsf{Stp}}, \mathcal{L}_{\mathsf{Srch}}, \mathcal{L}_{\mathsf{Updt}})$ be a leakage function, where $\mathcal{L}_{\mathsf{Stp}}$ denotes setup leakage, $\mathcal{L}_{\mathsf{Srch}}$ the search leakage, and $\mathcal{L}_{\mathsf{Updt}}$ the update leakage. The adversary and challenger interact in the games EMMReal or EMMIdeal as defined below. First, the honest-but-curious adversary chooses a multi-map MM. In EMMReal, the encrypted multi-map EMM is generated by $\mathsf{Setup}(1^\lambda, N, M, \mathsf{MM}; 1^\lambda)$, whereas in EMMIdeal the encrypted multi-map is simulated by a (stateful) simulator Sim on input $\mathcal{L}_{\mathsf{Stp}}(\mathsf{MM}, N, M)$. After receiving EMM, the adversary adaptively issues search and update queries. All queries are answered via the corresponding client protocol in EMMReal. In EMMIdeal, search queries for label ℓ are simulated by Sim on input of $\mathcal{L}_{\mathsf{Srch}}(\ell)$, while update queries for operation op, keyword ℓ, and value v are simulated by Sim on input of $\mathcal{L}_{\mathsf{Updt}}(\mathsf{op}, \ell, v)$. The adversary outputs a bit b.

Definition 3 (Semantic Security). *Let* Σ *be an* EMM *scheme and* $\mathcal{L} = (\mathcal{L}_{\mathsf{Stp}}, \mathcal{L}_{\mathsf{Srch}}, \mathcal{L}_{\mathsf{Updt}})$ *be a leakage function.* Σ *is* $\mathcal{L}$*-adaptively secure if for all honest-but-curious PPT adversaries* $\mathcal{A}$*, there exists a PPT simulator* Sim *such that*

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

Semantic security guarantees that each query reveals no information to the server beyond the leakage of the query.

Leakage Patterns. We now recall the standard leakage patterns; we refer to [1] for formal definitions. The search pattern sp, update pattern up, and query pattern qp are defined as follows:

(1) The search pattern $\mathsf{sp}(\ell)$ for a label ℓ comprises of the time slots corresponding to previous search queries on ℓ.

(2) The update pattern $\mathsf{up}(\ell)$ for a label ℓ comprises of the time slots corresponding to previous update queries on ℓ.
(3) The query pattern $\mathsf{qp}(\ell) = (\mathsf{sp}, \mathsf{up})$ for a label ℓ is the time slots of previous search or update queries for a label ℓ. In multi-map literature, the query pattern is also referred to as *label equality pattern*.

We also define the volume pattern cnt: the volume pattern $\mathsf{cnt}(\ell)$ for a label ℓ is equal to the number of values that match label ℓ.

Forward and Backward Privacy. Forward privacy was introduced in [46]. Here, we target the stronger variant introduced in [29], *i.e.*, we ask that updates leak no information.

Definition 4. *A DEMM scheme is* forward-private *if it is secure with respect to a leakage function* $\mathcal{L} = (\mathcal{L}_{\mathsf{Stp}}, \mathcal{L}_{\mathsf{Srch}}, \mathcal{L}_{\mathsf{Updt}})$ *such that* $\mathcal{L}_{\mathsf{Updt}} = \bot$.

Backward privacy [8] restricts leakage incurred by deletions. We consider type-II backward privacy, which requires search queries to leak at most the values currently matching label ℓ, when they were inserted, and when all updates on ℓ happened (but not their content). We refer to [8] for formal definitions.

By convention, we do not supply an explicit deletion operation (*cf.* Section 2.4) and employ the framework in [7] to augment our schemes to support deletes, combined with the techniques in [8, Section 4.3] to obtain type-II backward privacy. Note that while generic, this transformation is efficient and forms the basis of most modern and efficient type-II backward-private schemes. Intuitively, deletes are stored in an additional encrypted multi-map $\mathsf{EMM}_{\mathsf{del}}$. That is, to delete label-value pair (ℓ, v), the pair (ℓ, v) is added to $\mathsf{EMM}_{\mathsf{del}}$. On search, the client searches label ℓ on both EMMs and obtains two sets of values: $\mathcal{V}$ and $\mathcal{V}_{\mathsf{del}}$. For backward-privacy, the client sets $\mathcal{R} = \mathcal{V} \setminus \mathcal{V}_{\mathsf{del}}$ as the result.

Note that our schemes are response-hiding, *i.e.*, on search the client retrieves the values matching ℓ locally without revealing the values to the server.

EMM Supporting Dummy Updates. The notion of EMMs supporting dummy updates was introduced in [40]. Intuitively, such an SSE scheme allows to make *dummy* updates that are indistinguishable from real updates, but do not change the content of the database. This procedure is denoted by $\mathsf{DummyUpdate}$, which takes the client secret key K and client state st as input. Also, Setup receives an additional parameter D, an upper bound on the number of possible dummy updates. We refer to Appendix E.1 for a formal definition. We refer to the full version [21] for a formal definition.

Efficiency Measures. We now recall the notions of storage efficiency [12] and page efficiency [6]. We set $\mathsf{K} \leftarrow \mathsf{KeyGen}(1^\lambda)$ and $\mathsf{EMM} \leftarrow \mathsf{Setup}(\mathsf{K}, N, M, \mathsf{MM})$, and let $S = (\mathsf{op}_i, \mathsf{opin}_i)_{i=1}^s$ be a sequence of search and update queries, where $\mathsf{op}_i \in \mathsf{add}, \mathsf{del}, \mathsf{srch}$ denotes the operation and $\mathsf{opin}_i = (\mathsf{op}_i, \ell_i, v_i, \mathsf{st}_i; \mathsf{EMM}_i)$ its input. After executing all previous operations op_j for $j \leq i$, we denote the resulting client state as st_i, the encrypted multi-map as EMM_i, and the underlying plaintext multi-map as MM_i. We assume that the total number of identifiers (resp. values) never exceeds N (resp. M). Also, if $\mathsf{op}_i = \mathsf{srch}$, the query is a search query and a value is not provided.

Definition 5 (Storage Efficiency). *An EMM scheme has storage efficiency E if for any $\lambda, N \in \mathbb{N}$, multi-map MM, sequence S, and any i, $|\mathsf{EMM}_i| \leq E \cdot |\mathsf{MM}_i|$.*

Definition 6 (Page Pattern). *Regard server-side storage as an array of* pages*, containing encrypted multi-map EMM. When processing a search query $\mathsf{Search}(\mathsf{K}, \ell_i, \mathsf{st}_i; \mathsf{EMM}_i)$ or an update query $\mathsf{Update}(\mathsf{K}, \ell_i, v_i, \mathsf{op}_i, \mathsf{st}_i; \mathsf{EMM}_i)$, the server accesses pages $p_1, \ldots, p_{h'}$. We call this sequence of pages the* page pattern *and denote it as $\mathsf{PgPat}(\mathsf{op}_i, \mathsf{opin}_i)$.*

Definition 7 (Page Efficiency). *An EMM scheme has page efficiency P if for any $\lambda, N \in \mathbb{N}$, multi-map MM, sequence S, and any i, $|\mathsf{PgPat}(\mathsf{op}_i, \mathsf{opin}_i)| \leq P \cdot X$, where X is the number of pages accessed to answer the query in the plaintext MM.*

Definition 8 (Communication Efficiency). *An EMM scheme has communication efficiency C if for any $\lambda, N \in \mathbb{N}$, multi-map MM, sequence S, and any i, we have $R \leq C \cdot X$, where R is the total client-server communication when operation op_i with input opin_i is performed on EMM, where X is the size of the answer to the query in plaintext multi-map MM.*

3 Our Static EMM

As introduced in Section 1.1, a key challenge in page-efficient EMMs is to retrieve only the relevant values for a queried label without incurring bandwidth overhead. To motivate our approach, we first present a simpler static scheme that captures the core idea that we later extend to the dynamic setting (Section 4). Similar to prior work, we structure data using weighted hashing to achieve good page efficiency. Our main insight is that, if the bins are organized for efficient retrieval, we can avoid downloading an entire bin at query time and instead retrieve only the encrypted values that match the queried label (see Fig. 1c for a visualization). Specifically, the values associated to some label ℓ are stored close-together within a few bins B_i and the server only sends the matching values within B_i. While the idea may seem natural in hindsight, naively revealing the exact memory locations of values introduces leakage: as observed in [2], these locations are not independent and should thus not be revealed to the server. Securely instantiating this idea therefore requires careful design and analysis.

In our static EMM $\mathsf{S1C}$, the bins B_i are structured as hash tables that allow the retrieval of individual values directly. We therefore denote $T_i := B_i$ to stress that T_i is a table. Observe that accessing specific locations in T_i reveals the exact positions of values in the bins. Following the paradigm of [2], the set of values V_ℓ associated with a label ℓ is mapped to a set of bins $(T_{\ell,1}, \ldots, T_{\ell,c})$, where c depends on the weighted hashing variant. If the choice of the bin to which a given value is assigned depends on the multi-map's distribution, then the value's locations *across* bins are not independent, which could compromise security. For example, in weighted cuckoo hashing or weighted two-choice hashing we have $c = 2$, *i.e.*, a value can be mapped to either of two bins $T_{\ell,1}$ or $T_{\ell,2}$ depending on the multi-map's distribution; as a result, instantiating the scheme with one of these hashing mechanisms may lead to non-trivial access pattern leakage. (We provide more intuition for these issues in Appendix C.)

We observe that this issue does not occur for one-choice allocation ($\mathsf{1C}$), where exactly one bin T_ℓ is selected for each label (*i.e.*, $c = 1$) using a hash function $\mathcal{H}$. In our security analysis, we show that this allows the exact locations of values in the bin to be revealed without compromising security. Our construction is based on Pluto [6], which stores values in full pages using a static EMM. However, we make a subtle but important modification: incomplete pages are stored using one-choice allocation across multiple bins, with each bin structured as a hash table. Full pages are stored in standard hash tables. This design avoids the bandwidth overhead while preserving a page efficiency of $\widetilde{\mathcal{O}}(\log N/p)$.

3.1 Scheme Overview

The client initializes two tables, T_{full} and T_{len}, with $\lceil N/p \rceil$ and N entries, respectively. Let MM be the plaintext multi-map that we wish to securely outsource. For each label in MM, the client partitions the corresponding values into pages, encrypts each of the full pages, and stores the encrypted pages in T_{full}. The table T_{len} is used to store the (blinded) number of values associated with each label.

Since not all value sets have size divisible by p, we must also handle the overflow values, *i.e.*, the values in an incomplete page. For a label with cnt_i associated values, the first $\lfloor \mathsf{cnt}_i/p \rfloor$ pages are full, while a $(\lfloor \mathsf{cnt}_i/p \rfloor + 1)$-st page contains the remaining $\mathsf{cnt}_i \bmod p$ values, if any. To store these overflow values, the client initializes another $m = \lceil N/(p \cdot \delta(\lambda) \cdot \log N) \rceil$ empty tables, $T_{\mathsf{val},0}, \ldots, T_{\mathsf{val},\mathsf{m}-1}$, encrypts the values, and inserts each encrypted value into these tables using one-choice.

To query a label, the client generates a query-specific token that allows the server to retrieve and unblind the number of associated values from T_{len}. The token can further be used to retrieve the corresponding full pages from T_{full} as well as any additional values stored in the tables $T_{\mathsf{val},\gamma}$

for $\gamma \in [0, m-1]$. The pseudocode is given in Algorithm 2, and a detailed description is given in Appendix D.1.

The pseudocode is given in Algorithm 2. Note that $\mathcal{H}_P$ is a hash function that maps to $\{0,1\}^{2\lambda}$.

Remark. For clarity, we instantiate our static scheme concretely. We note, however, that T_{full} can instead be computed by defining a plaintext multimap $\mathsf{MM}_{\mathsf{full}}$ that maps each label $\ell_i \in \mathsf{MM}$ to the set of corresponding complete pages and then encrypting $\mathsf{MM}_{\mathsf{full}}$ using any static EMM scheme. Since $\mathsf{MM}_{\mathsf{full}}$ stores full pages, this black-box EMM scheme does not itself need to be page efficient. This abstraction will later be useful for understanding our dynamic scheme in Section 4, as we make similar black-box use of a specific type of EMM.

Algorithm 2 S1C

$\mathsf{S1C.KeyGen}(1^\lambda)$

Sample $\mathsf{K}_{\mathsf{PRF}} \leftarrow \{0,1\}^\lambda$ and encryption key $\mathsf{K}_{\mathsf{Enc}}$
return $\mathsf{K} = (\mathsf{K}_{\mathsf{PRF}}, \mathsf{K}_{\mathsf{Enc}})$

$\mathsf{S1C.Search}(\mathsf{K}, \ell_i; \mathsf{EMM})$

Client:

Set $(K_i, \mu_i, P_i') \leftarrow F_{\mathsf{K}_{\mathsf{PRF}}}(\ell_i)$
send (K_i, μ_i, P_i')

Server:

Initialize empty set R
Set $\mathsf{cnt}_i \leftarrow T_{\mathsf{len}}[K_i] \oplus \mu_i$
Set $\mathsf{pgcnt}_i \leftarrow \lfloor \mathsf{cnt}_i / p \rfloor$
for all $1 \le j \le \mathsf{pgcnt}_i$ **do**
    Set $P_{i,j} \leftarrow \mathcal{H}_P(P_i', j)$
    Add $T_{\mathsf{full}}[P_{i,j}]$ to R
Set $P_{i,\mathsf{pgcnt}_i+1} \leftarrow \mathcal{H}_P(P_i', \mathsf{pgcnt}_i + 1)$
Set $\gamma \leftarrow \mathcal{H}_{\mathsf{1C}}(P_{i,\mathsf{pgcnt}_i+1})$
for all $j \in [\mathsf{cnt}_i \bmod p]$ **do**
    Set $P_{i,\mathsf{pgcnt}_i+1,j} \leftarrow \mathcal{H}_P(P_i', \mathsf{pgcnt}_i + 1, j)$
    Add $T_{\mathsf{val},\gamma}[P_{i,\mathsf{pgcnt}_i+1,j}]$ to R
send R

Client:

Decrypt R to retrieve $\mathsf{MM}(w)$

$\mathsf{S1C.Setup}(1^\lambda, \mathsf{K}, N, M, \mathsf{MM})$

Initialize empty table T_{len} with N entries, each of size $\lceil \log N \rceil$
Initialize empty table T_{full} with $\lceil N/p \rceil$ entries, each of size $p \cdot |\mathcal{V}|_b$
Set $\mathsf{cap} \leftarrow \mathcal{O}(p\delta(\lambda)\log(N/p))$ and $m \leftarrow \left\lceil \frac{N}{p \cdot \delta(\lambda) \cdot \log(N/p)} \right\rceil$
for all $\gamma \in \{0, \ldots, m-1\}$ **do**
    Initialize empty table $T_{\mathsf{val},\gamma}$ with cap entires, each of size $|\mathcal{V}|_b$
for all labels ℓ_i **do**
    Set $V_i \leftarrow \mathsf{MM}[\ell_i]$ and $\mathsf{cnt}_i \leftarrow |V_i|$
    Set $\mathsf{pgcnt}_i \leftarrow \lfloor \mathsf{cnt}_i / p \rfloor$
    Set $(K_i, \mu_i, P_i') \leftarrow F_{\mathsf{K}_{\mathsf{PRF}}}(\ell_i)$
    Set $T_{\mathsf{len}}[K_i] \leftarrow \mu_i \oplus \mathsf{cnt}_i$
    Partition V_i into $\mathsf{pgcnt}_i + 1$ lists $V_{i,j}$ such that $|V_{i,j}| = p$ for $j \le \mathsf{pgcnt}_i$, and $0 \le |V_{i,\mathsf{pgcnt}_i+1}| < p$
    for all $1 \le j \le \lfloor \mathsf{cnt}_i / p \rfloor$ **do**
        Set $P_{i,j} \leftarrow \mathcal{H}_P(P_i', j)$
        Set $T_{\mathsf{full}}[P_{i,j}] \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(V_{i,j})$
    if $\mathsf{cnt}_i \bmod p \neq 0$ **then**
        Set $P_{i,\mathsf{pgcnt}_i+1} \leftarrow \mathcal{H}_P(P_i', \mathsf{pgcnt}_i + 1)$
        Set $\gamma \leftarrow \mathcal{H}_{\mathsf{1C}}(P_{i,\mathsf{pgcnt}_i+1})$
        Set $j \leftarrow 1$
        for all $\mathsf{v} \in V_{i,\mathsf{pgcnt}_i+1}$ **do**
            $P_{i,\mathsf{pgcnt}_i+1,j} \leftarrow \mathcal{H}_P(P_i', \mathsf{pgcnt}_i + 1, j)$
            $T_{\mathsf{val},\gamma}[P_{i,\mathsf{pgcnt}_i+1,j}] \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(\mathsf{v})$
            $j \leftarrow j + 1$
Fill T_{len} and T_{full} with random entries
send $\mathsf{EMM} = (T_{\mathsf{len}}, T_{\mathsf{full}}, (T_{\mathsf{val},0}, \ldots, T_{\mathsf{val},m-1}))$

Efficiency. It is easy to see that storage efficiency and communication efficiency are $O(1)$. Page efficiency is $\widetilde{\mathcal{O}}(\log N/p) + \mathcal{O}(1)$: each search query for a label ℓ with cnt corresponding values, at most $p-1$ distinct positions in $T_{\mathsf{val},\gamma}$ of size $\mathsf{cap} = \widetilde{\mathcal{O}}(p \log N/p)$, and at most $\mathsf{pgcnt} = \lfloor \mathsf{cnt}/p \rfloor$ full pages are accessed in T_{full}.

Correctness. Correctness is straightforward. By construction, the full pages are stored in T_{full} at location $P_{i,j}$ and the incomplete pages are stored in the table $T_{\mathsf{val},\gamma}$, where $\gamma = \mathcal{H}_{\mathsf{1C}}(P_{i,\mathsf{pgcnt}})$. The properties of $\mathsf{1C}$ (*cf.* Lemma 1) ensure that no table $T_{\mathsf{val},\gamma}$ overflows.

Security. The setup leakage is the multi-map size N and search leakage is the search pattern sp and volume pattern cnt. We give a formal security analysis in Appendix F.1. We give a formal security analysis in the full version [21]. Note that our analysis crucially relies on the properties of $\mathsf{1C}$. Let us briefly explain the core reason this enables us to simulate the access to the concrete locations.

As full pages are stored together in T_{full} at a single random location, simulation is straightforward following standard techniques from EMM literature. The crux is in how we simulate the access to incomplete pages. Here, we rely on the observation that all values within an incomplete page for

label ℓ are mapped to the same bin T_ℓ. (Note that for the first search query on label ℓ the bin is chosen randomly in the view of the adversary.) Then, since each bin is structured as a table and the values are mapped to random locations within the table, the simulator simply chooses $\mathsf{cnt}(\ell)$ random locations in T_ℓ to simulate the search query. Notably, the search pattern leakage allows the simulator to simulate queries consistently.

4 Our EMM with Dummy Updates

Our dynamic, page-efficient EMM scheme (Section 5), makes black-box use of a dynamic (not necessarily page-efficient) EMM scheme that supports *dummy updates*. Dummy updates were introduced in [40] in order to simultaneously realize forward privacy and I/O efficiency. A dummy update allows the client to issue a query that is indistinguishable from a real update from the server's point of view, without requiring additional storage overhead.

In this section, we present our EMM scheme with dummy updates, Dummy1C, which we use to build our dynamic EMM in Section 5. Dummy1C supports a $\mathsf{poly}(\lambda)$ number of dummy updates with $\widetilde{\mathcal{O}}(\log N)$ communication for updates and $\mathcal{O}(1)$ search communication. Notably, Dummy1C gives a different tradeoff to the $\mathsf{Dummy}(\Sigma)$ scheme of Minaud and Reichle [40] which has $\widetilde{\mathcal{O}}(\log\log N)$ communication for both updates and searches. We give a brief overview of $\mathsf{Dummy}(\Sigma)$ [40] below.

At a high level, $\mathsf{Dummy}(\Sigma)$ combines a forward-private EMM scheme Σ with two-choice (2C) allocation.[8] Σ is required to be such that each value is stored in a hash table T at a concrete location determined by the scheme. For a real update, instead of inserting v into T at location P (as in Σ), v is placed in the least loaded of two bins B_α, B_β, where $\alpha, \beta \leftarrow \mathcal{H}(P)$. For a dummy update, two random bins are accessed. In both cases, the bins are reencrypted so the server cannot distinguish whether an insertion occurred. During search, the client computes positions P_i via $\Sigma.\mathsf{Search}$ and retrieves the corresponding bins $B_{\alpha_i}, B_{\beta_i}$ for $\alpha_i, \beta_i \leftarrow \mathcal{H}(P_i)$.

We further observe that values can be retrieved directly from bins to reduce communication. To achieve this in Dummy1C, each bin is implemented as a hash table supporting $\mathcal{O}(1)$ read access. With 2C allocation, each value could lie in one of two bins, forcing the client to access both. Since revealing the correct bin leaks information (see Appendix E.3), we replace 2C with 1C so that each value maps to a single bin. Since revealing the correct bin leaks information (see full version [21] for a detailed explaination), we replace 2C with 1C so that each value maps to a single bin. We note that our security proof requires special care in handling hash table keys.

The forward security of Σ ensures forward security of $\mathsf{Dummy}(\Sigma)$. Furthermore, because the values are stored in encrypted bins, the server cannot distinguish between real and dummy updates since—in both cases—two (seemingly random) bins are retrieved and reencrypted. The bins only need to be scaled for N values, since dummy updates do not change the content of the bins. This approach has a multiplicative communication blowup of $\widetilde{\mathcal{O}}(\log\log N)$ for searches and updates due to the use of 2C allocation.

Because Dummy1C relies on hash tables, care is needed to hide the distinction between real and dummy updates. In standard hash tables, keys are stored in the clear; while these are only hash values, real updates necessarily modify a key whereas dummy updates cannot, which leaks information.

We resolve this by introducing a new data structure, a *keyless cuckoo hash table*, which is based on cuckoo-hashing, but modifies the table so that the hash keys are not stored with the items. During lookup, both candidate items are returned, and the client determines which is correct. We describe standard cuckoo hashing and our modified construction in Appendix E.2. We describe standard cuckoo hashing and our modified construction in the full version [21].

Dummy1C has $\mathcal{O}(1)$ and $\widetilde{\mathcal{O}}(\log N)$ search and update communication efficiency, respectively. A detailed description of the construction and its security can be found in Appendix E and the pseudocode in Algorithm 5. A detailed description of the construction and its security, as well as the pseudocode are given in the full version [21].

[8] In 2C, a ball is placed in the least loaded of two bins chosen uniformly at random. After n balls are allocated to $m = N/\widetilde{\mathcal{O}}(\log\log N)$ bins, the heaviest bin contains at most $\widetilde{\mathcal{O}}(\log\log n)$ balls.

5 Our Dynamic and Forward-private EMM

We now turn to the dynamic setting, where achieving good page efficiency is complicated by the need for forward and backward privacy. We build on our static construction (Section 3) and our EMM with dummy updates (Section 4), to develop a dynamic EMM scheme that achieves constant search communication together with good page efficiency.

The main challenge is that forward privacy requires that no information is leaked about which label is updated, whereas page efficiency requires that values associated with the same label be stored close together in memory (*i.e.*, label identities and storage locations must remain correlated). There are two known approaches that preserve forward and backward privacy while still achieving page-efficiency.

(1) The first [40] employs an EMM with *dummy updates* to achieve $\widetilde{\mathcal{O}}(\log\log N/p)$ page efficiency for search and updates, at the cost of $\mathcal{O}(M)$ client storage.
(2) The second [41] is based on hierarchical SSE [16], where updates rely on oblivious sorting and compaction. While this approach requires (almost) no client storage, it incurs $\mathcal{O}(\log N)$ page efficiency and bandwidth due to the use of oblivious primitives.

Our dynamic EMM builds on (1) the Hermes framework [40]. Note that while asymptotically the client storage is $O(M)$, in practice the size is small (*e.g.*, at most 1MB for 400K emails in our experiments, see Section 6). In contrast, while [41] has almost no client storage, achieving optimal bandwidth appears challenging due to its hierarchical structure and use of oblivious primitives. We leave optimizing the search bandwidth of this approach for future work.

5.1 Scheme Overview

For convenience, we partition updates into epochs such that the k-th epoch comprises all updates and searches that are performed between the $((k-1)\cdot M+1)$-th and $(k\cdot M)$-th update. Updates are buffered at the client and later pushed to the server according to an oblivious schedule (see Fig. 2). During a search, values that have not yet been pushed are retrieved from the client buffer. As in [40], client storage scales linearly with the number of labels, which is typical for efficient forward-private SSE.

Similar to Hermes, we employ a bin structure, $(T_i)_{i\in[m]}$, to store incomplete pages and an EMM, Σ, which supports dummy updates to store full pages. We instantiate Σ using $\mathsf{Dummy1C}$ (our construction from Section 4) and structure each bin as a hash table for efficient access.

We recap the Hermes framework and describe our modifications to obtain optimal search bandwidth. We first discuss an amortized variant, where each M updates may read up to $O(M)$ pages; the construction can be deamortized, which we discuss below. Let p denote the page size, N an upper bound on the total number of values, and M the number of labels. Depending on whether $N < pM$ or $N \geq pM$, the multi-map is in the *sparse* or *dense* regime, respectively. As in Hermes, our construction proceeds differently in these two cases.

Sparse Regime. In the sparse regime, Hermes builds on a static EMM. Minaud and Reichle [40] observe that for every M updates, it is possible to read the entire encrypted multi-map with amortized $\mathcal{O}(1)$ update page efficiency, since $N < pM$ implies that the entire multi-map can be stored in $\mathcal{O}(M)$ pages. This allows buffering M updates at the client and reencrypting the multi-map locally. Note that this rebuild process is *forward private*, *i.e.*, leaks no additional information, and can be deamortized.

Hermes instantiates this approach using $\mathsf{LayeredSSE}$ [39] and thus inherits its $\widetilde{\mathcal{O}}(\log\log N/p)$ page efficiency and search bandwidth. In contrast, we base our dynamic construction, $\mathsf{D1C}$, on our static EMM $\mathsf{S1C}$ (Section 3), specifically leveraging one-choice allocation. As a result, $\mathsf{D1C}$ inherits the performance guarantees of $\mathsf{S1C}$, thus achieving good page efficiency and $\mathcal{O}(1)$ search bandwidth.

Concretely, in our scheme, the server maintains $m = \mathcal{O}(N/\mathsf{cap})$ bins, each of capacity $\mathsf{cap} = p\cdot\widetilde{\mathcal{O}}(\log N/p)$. Each value associated with a label ℓ is stored in a bin T_i, where $i = \mathcal{H}_{\mathsf{1C}}(P_\ell)$ and P_ℓ is a search token derived by applying a PRF to ℓ. Once p or more values associated with ℓ accumulate in T_i, a full page of values for ℓ is flushed from T_i and inserted into $\mathsf{Dummy1C}$.

Dense Regime. In the dense regime, the Hermes framework maintains two structures: (i) a collection of bins $\mathbf{B} = (B_1, ..., B_M)$, each storing an incomplete page of up to $p-1$ values per label, and (ii) an EMM scheme Σ that stores full pages and supports dummy updates. All M bins can be read with $O(1)$ page efficiency per epoch. When a bin becomes full, its contents are flushed to Σ. To preserve forward privacy, updates to Σ are performed independently of whether a bin was full. Since at most $\lceil M/p \rceil$ full pages and M bins can be filled over M updates, $O(M)$ real or dummy updates suffice per epoch, and the construction can be deamortized. Overall efficiency is inherited from Σ and the bins.

To instantiate this framework, we set $\Sigma = \mathsf{Dummy1C}$ (Section 4), our EMM scheme with dummy updates based on one-choice hashing. $\mathsf{Dummy1C}$ achieves optimal search bandwidth, reading $O(1)$ memory locations per search. As in our static scheme $\mathsf{S1C}$, we structure the bins $\mathbf{B}$ as tables $\mathbf{T} = (T_1, ..., T_M)$, which allows searches to avoid bandwidth overhead.

In this regime, each bin has capacity $\mathsf{cap} = p$, and there are $m = M$ bins in total. A value v associated with label ℓ is stored in the bin $T_{f(\ell)}$, where f is an injective choice function. When a bin T_i becomes full, the corresponding full page is flushed to $\mathsf{Dummy1C}$.

Updates are pushed to the server using a client buffer mechanism. We depict the update schedule for the dense regime in Fig. 2. We stress that, while the schedule looks rather complex, all operations are efficient and comprise of page-efficient memory accesses, local restructuring of data on the client, or rebuilding small bins T_i.

In both regimes, we obtain optimal search bandwidth with $\widetilde{\mathcal{O}}(\log N/p)$ page efficiency, optimal storage efficiency (the encrypted multi-map has size $\mathcal{O}(N)$), and $\widetilde{\mathcal{O}}(\log N/p)$ update page efficiency and communication in the worst case.

Towards Constant Communication. Since $\mathsf{Dummy1C}$ (Section 4) has $\mathcal{O}(1)$ search communication, invoking $\mathsf{Dummy1C.Search}$ has constant overhead. Naively, if we were to download each bin T_i as in [40], then communication would be $\Theta(\mathsf{cap}) = \Omega(p)$ in the worst case. We thus structure each bin as a hash table, as we did in our static construction (Section 3), to achieve constant communication.

A subtle but important detail is that we need to *rekey* the hash table whenever a bin T_i is updated: the hash table keys leak whether a full page is written to Σ and removed from T_i. It is tempting to employ our keyless hash table from Appendix E.2, but the keys would be revealed on search. It is tempting to employ our keyless hash table (see full version [21] for details), but the keys would be revealed on search. Since values are removed from the table, the distribution of the seeds for $\mathcal{H}_{\mathsf{CT}}$ may leak whether a previously searched value was removed from T_i: removed keys might not constitute a valid cuckoo assignment. Instead, we use fresh keys each epoch. When a table T_i is rebuilt in the j-th epoch, then j is employed to derive the keys via a PRF. With this modification, the keys are freshly distributed after the table T_i is rebuilt each epoch, independent of whether values were written to or removed from T_i. Since the update schedule of bin T_i is deterministic, the client can derive the correct key according to the current epoch. This resolves the issue.

We present a unified construction in Algorithm 3.

Efficiency We assume that Σ is instantiated using our construction from Section 4. Roughly, the optimal search bandwidth of $\mathsf{S1C}$ (resp. Σ) translates to optimal search bandwidth for incomplete pages (resp. full pages) in $\mathsf{D1C}$. Furthermore, the page efficiency of $\mathsf{S1C}$ (resp. number of distinct memory locations accessed in Σ per value) translates to the page efficiency of D1C for search and updates on incomplete (resp. full pages). Note that Σ does not have to be page efficient itself as it exclusively stores full pages in $\mathsf{D1C}$.

In both regimes, storage and search communication is $\mathcal{O}(1)$. The other efficiency metrics depend on the regime; we elaborate below.

In the sparse regime, searches have $\widetilde{\mathcal{O}}(\log N/p) + \mathcal{O}(1)$ page efficiency: each search query on a label ℓ with cnt matching values, at most $p-1$ distinct positions in T_γ of size $\mathsf{cap} = \widetilde{\mathcal{O}}(p \log N/p)$, and at most $\mathsf{pgcnt} = \lfloor \mathsf{cnt}/p \rfloor$ full pages are accessed in Σ, where each Σ access induces $\mathcal{O}(1)$ page accesses. Update page efficiency is $\widetilde{\mathcal{O}}(\log N/p)$ as T_γ has size cap and Σ induces $\widetilde{\mathcal{O}}(\log N/p)$ page accesses. In this dense regime, tables T_γ are of size $\mathsf{cap} = p$. Search page efficiency is $\mathcal{O}(1)$ and update page efficiency remains $\widetilde{\mathcal{O}}(\log N/p)$.

Note that in each epoch, there are at most M full pages that are completed and buffered in $\mathsf{CFP}_{\mathsf{new}}$. Therefore, client storage is $\mathcal{O}(pM)$. This is simply a consequence of presenting the construction for both regimes in a unified manner to improve readability. With a simple modification, we obtain an upper bound $\mathcal{O}(M) = M \cdot \mathcal{O}(|\mathcal{V}|_b + |\mathcal{L}|_b + |\log N|)$ on client storage. We describe the modifications below.

In the dense regime, there are at most M/p full pages amongst the values added in any epoch. Further, each bin T_γ contains at most $p-1$ values that match label ℓ such that $f(\ell) = \gamma$. Therefore, an incomplete page added in any epoch can lead to at most 1 full page when merged with the content in T_γ. Instead of adding this completed page to $\mathsf{CFP}_{\mathsf{new}}$, the full page is written to Σ immediately (or a dummy update is executed if no such page is completed), therefore there are at most $\mathcal{O}(M/p)$ full pages buffered on the client.

In the sparse regime, there are also at most N/p full pages amongst the fresh values per epoch. However, each bin T_γ can induce more than one completed page when merged with the incomplete pages in $\mathsf{CB}_{\mathsf{out}}[\gamma]$, and we cannot simply push all completed pages to Σ immediately without incurring an overhead in communication.

As a first step, let us argue that at most $\widetilde{\mathcal{O}}(\log N/p)$ incomplete pages become full when a bin T_γ is merged with $\mathsf{CB}_{\mathsf{out}}[\gamma]$. If there were more than $\widetilde{\mathcal{O}}(\log N/p)$ full pages in bin T_γ at any point during a merge, then this bin overflows over capacity $\mathsf{cap} = \widetilde{\mathcal{O}}(\log N/p)$. This occurs only with negligible probability by the $\mathsf{1C}$ upper bound (*cf.* Lemma 1).

We can reduce client storage as follows. Between each T_γ access, we postpone reading $T_{\gamma+1}$ for the next $\widetilde{\mathcal{O}}(\log N/p)$ updates[9]. For each update, we perform 2 updates on Σ: one access to push a full page and another to (potentially) push one of the (up to) $\widetilde{\mathcal{O}}(\log N/p)$ completed pages to Σ. After $\widetilde{\mathcal{O}}(\log N/p)$-many updates after T_γ was read, all complete pages will be pushed to Σ before $T_{\gamma+1}$ is read. With this modification, the client storage is at most $\mathcal{O}(M) + p \cdot \widetilde{\mathcal{O}}(\log N/p) = \mathcal{O}(M)$, since $N/p < M$ in this regime.

Correctness It is straightforward to confirm correctness. Observe that all full pages are moved to Σ via the update mechanism and incomplete pages that match label ℓ are stored in bins $T_\gamma[f(\ell)]$. At most M pages are filled per epoch, therefore all full pages from $\mathsf{CFP}_{\mathsf{out}}$ are flushed to Σ each epoch.

Security Setup leaks the upper bound N on the multi-map size and search leaks the query pattern qp and volume pattern cnt. As a consequence, the scheme is forward and backward private (*cf.* Section 2.4). We give a formal analysis in Appendix F.2. Let us provide some intuition below. We give a formal analysis in the full version [21]. Let us provide some intuition below.

Simulation of setup and update queries is straightforward. [10] To simulate search queries, we can deal with full pages (stored in Σ) as in Hermes [40], leveraging the dummy security of Σ. Indeed, our use of the EMM with dummy security Σ mimics the use in Hermes, except that we rely on a different instantiation for efficiency. Roughly, given the search and volume pattern for $\mathsf{D1C}$, we can recompute the query pattern and volume pattern for Σ. This is indeed possible as the number of full pages stored in $\mathsf{D1C}$ and the update pattern of $\mathsf{D1C}$ depends on when incomplete pages in the tables T_i become full and are moved to $\mathsf{D1C}$—this can be learned from $\mathsf{D1C}$'s leakage pattern alone. Given these patterns, the simulator for Σ takes care of the simulation.

Again, we must take special care for incomplete pages. These are stored in the bins $(T_0, \ldots, T_{m-1})$. Though the simulation is more technical, the idea is the same as in the proof for $\mathsf{S1C}$. That is, each incomplete page V_ℓ matching label ℓ is mapped to a single bin T_ℓ (according to choice function f, depending on the regime). The table T_ℓ itself is structured such that each value $v \in V_\ell$ is mapped to a random location k in T_ℓ. While values are encrypted, it is also important that the keys k—the

[9] This is possible, since $\frac{M}{m} \geq \widetilde{\mathcal{O}}(\log N/p)$ in this regime. Then, each epoch all tables T_γ will still be accessed and updated as before—instead of accessing all tables T_γ at the beginning of the epoch, the accesses of T_γ are spaced evenly amongst the M updates per epoch.

[10] We leverage that update leakage for Σ is empty, therefore we may simply invoke the simulator of Σ with empty input. The corresponding update and volume patterns are recomputed during search in a consistent manner.

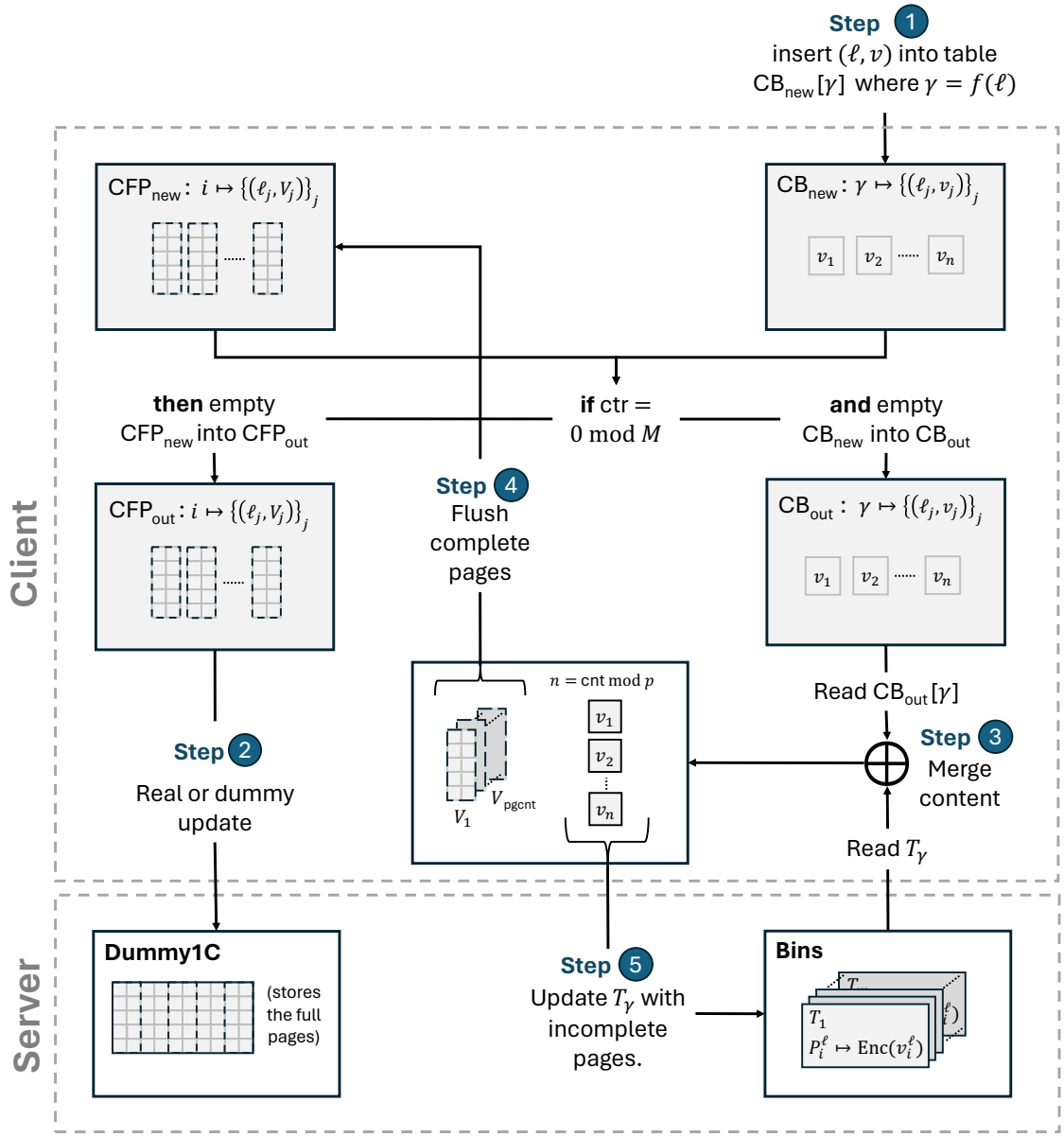


Fig. 2 – An informal overview of our dynamic EMM scheme for the dense regime ($N \geq pM$). For each (ℓ, v) added to MM, the counter ctr, and the number of values associated with ℓ, $T_{\mathsf{len}}[\ell]$, are incremented. In this regime, there is one bin per label, each bin has capacity p, and each label ℓ maps to one table index γ via f. We refer to Algorithm 3 for a formal description.

locations where the values v are stored and which are revealed in clear—can be simulated. Here, it is crucial that the tables T_i are fully rekeyed each epoch, *i.e.*, the location k_j of the j-th value v_j from an incomplete page matching ℓ is derived from (epoch, ℓ) and its index j within the page. Thus, the key-value pairs (k_j, v_j) are independent across epochs; this enables simulation of search queries, even when complete pages are moved to Σ in prior update queries (which at the time of the update query, is not yet known to the server).

Algorithm 3 D1C: pseudocode marked with ⌜ ⌝ and ▭ is executed in the sparse ($N < pM$) and dense regime ($N \geq pM$), respectively.

D1C.KeyGen(1^λ)

Sample $\mathsf{K_{Enc}}$ and $\mathsf{K_{PRF}}$ for Enc and F with λ, respectively
Sample $\mathsf{K_\Sigma} \leftarrow \Sigma.\mathsf{KeyGen}(1^\lambda)$
return $\mathsf{K} = (\mathsf{K_{Enc}}, \mathsf{K_{PRF}}, \mathsf{K_\Sigma})$

D1C.Setup(K, N, M)

Set ⌜ $\mathsf{cap} \leftarrow \mathcal{O}(p\delta(\lambda)\log(N/p))$ ⌝ ▭ $\mathsf{cap} \leftarrow p$
Set ⌜ $m \leftarrow \left\lceil \frac{N}{p \cdot \delta(\lambda) \cdot \log(N/p)} \right\rceil$ ⌝ ▭ $m \leftarrow M$
⌜ Let $f : \mathcal{L} \to \{0, \ldots, m-1\}$ such that $f(\ell) \mapsto \mathcal{H}_{1\mathsf{C}}(F_{\mathsf{K_{PRF}}}(\ell))$ ⌝
▭ Let $f : \mathcal{L} \to \{0, \ldots, m-1\}$ be a random injective partial function ▷ implicitly built iteratively when a new label is added in Update
Initialize empty tables $T_{\mathsf{len}}, \mathsf{CB_{new}}, \mathsf{CB_{out}}, \mathsf{CFP_{new}}, \mathsf{CFP_{out}}$
for all $\gamma \in \{0, \ldots, m-1\}$ **do**
    Init empty table T_γ with cap entries, each of size $|\mathcal{V}|_b + |\mathcal{L}|_b$
    Set $T_\gamma[k_i] \leftarrow \mathsf{Enc}_{\mathsf{K_{Enc}}}(0\|0)$ for $k_i \leftarrow \{0,1\}^{2\lambda}, i \in [\mathsf{cap}]$
$\mathsf{ctr} \leftarrow 0$ ▷ Counts the total number of updates
$\mathsf{num} \leftarrow 0$ ▷ Counts the number of full pages added this epoch
$\pi \xleftarrow{\$} S_M$ ▷ Sample random permutation
$\mathsf{EMM_{full}} \leftarrow \Sigma.\mathsf{Setup}(\mathsf{K_\Sigma}, \lceil N/p \rceil)$
$\mathsf{EMM} \leftarrow (\mathsf{EMM_{full}}, (T_0, \ldots, T_{m-1}))$
return EMM ▷ Client state is $(f, m, \mathsf{cap}, \mathsf{ctr}, \mathsf{num}, \pi, T_{\mathsf{len}}, \mathsf{CB_{new}}, \mathsf{CB_{out}}, \mathsf{CFP_{new}}, \mathsf{CFP_{out}})$

D1C.Search(K, ℓ; EMM)

Client:

Run $R_{\mathsf{full}} \leftarrow \Sigma.\mathsf{Search}(\mathsf{K_\Sigma}, \ell; \mathsf{EMM_{full}})$ ▷ retrieve full pages
Set $\gamma \leftarrow f(\ell)$ and $\mathsf{cnt} \leftarrow T_{\mathsf{len}}[\ell] \bmod p$
if $\mathsf{cnt} = \perp \vee \mathsf{cnt} = 0$ **then return** R_{full}
Set $\mathsf{epoch} \leftarrow \lceil \mathsf{ctr}/M \rceil$
if $b \leftarrow (\gamma > \mathsf{ctr} \bmod M)$ **then** ▷ T_γ was not updated this epoch
    Set $\mathsf{epoch} \leftarrow \mathsf{epoch} - 1$
Set $P_\ell \leftarrow F_{\mathsf{K_{PRF}}}(\mathsf{epoch}, \ell)$
send $(P_\ell, \mathsf{cnt}, \gamma)$

Server:

Initialize empty set R
for all $i \in \{1, \ldots, \mathsf{cnt}\}$ **do**
    Set $k_i \leftarrow \mathcal{H}_P(P_\ell, i)$ and add $T_\gamma[k_i]$ to R
send R

Client:

Decrypt R to retrieve incomplete list and combine with R_{full}
Retrieve values associated to ℓ from client state

D1C.Update(K, (ℓ, v), add; EMM)

Client:

$\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$
Add $(v\|\ell)$ to $\mathsf{CB_{new}}[\gamma]$ for $\gamma = f(\ell)$
Increment $T_{\mathsf{len}}[\ell] \leftarrow T_{\mathsf{len}}[\ell] + 1$
if $\mathsf{ctr} = 0 \bmod M$ **then** ▷ every fresh epoch
    Set $\mathsf{CB_{out}} \leftarrow \mathsf{CB_{new}}$ and empty $\mathsf{CB_{new}}$
    Set $\mathsf{CFP_{out}} \leftarrow \mathsf{CFP_{new}}$ and empty $\mathsf{CFP_{new}}$
    Set $\mathsf{num} \leftarrow 0$ and $\pi \xleftarrow{\$} S_M$ ▷ sample random permutation
Set $\gamma \leftarrow \mathsf{ctr} - 1 \bmod M$
Set $(\ell_V, V) \leftarrow \mathsf{CFP_{out}}[\gamma + 1]$
if $(\ell_V, V) \neq \perp$ **then**
    Run $\Sigma.\mathsf{Update}(\mathsf{K}, (\ell_V, V), \mathsf{add}; \mathsf{EMM_{full}})$
else
    Run $\Sigma.\mathsf{DummyUpdate}(\mathsf{K}; \mathsf{EMM_{full}})$
if $\gamma \geq m$ **then return**
send γ

Server:

send T_γ

Client:

Set $V'_\gamma := (v'_i\|\ell'_i)_{i=1}^{\mathsf{cap}} \leftarrow \mathsf{Dec}_{\mathsf{K_{Enc}}}(T_\gamma)$
Remove $(0\|0)$ values from V'_γ
Add $\mathsf{CB_{out}}[\gamma]$ to V'_γ and empty $\mathsf{CB_{out}}[\gamma]$
Parse V'_γ as multi-map $\mathsf{MM}_\gamma = \{(\ell'_i, (v_1, \ldots, v_{n_{\ell'_i}}))\}_i$
Reinitialize T_γ as empty table with cap entries
Set $\mathsf{epoch} \leftarrow \lceil \mathsf{ctr}/M \rceil$
for all ℓ_i in MM_γ **do**
    Set $V_i \leftarrow \mathsf{MM}_\gamma[\ell_i]$ and $\mathsf{cnt}_i \leftarrow |V_i|$
    Set $\mathsf{pgcnt}_i \leftarrow \lfloor \mathsf{cnt}_i/p \rfloor$
    Partition V_i into $\mathsf{pgcnt}_i + 1$ lists $V_{i,j}$ such that $|V_{i,j}| = p$ for $j \leq \mathsf{pgcnt}_i$, and $0 \leq |V_{i,\mathsf{pgcnt}_i+1}| < p$
    for all $1 \leq j \leq \lfloor \mathsf{cnt}_i/p \rfloor$ **do**
        Set $\mathsf{num} \leftarrow \mathsf{num} + 1$
        Set $\mathsf{CFP_{new}}[\pi(\mathsf{num})] \leftarrow (\ell_i, V_{i,j})$
    Set $j \leftarrow 1$ and $P_i \leftarrow F_{\mathsf{K_{PRF}}}(\mathsf{epoch}, \ell_i)$
    for all $\mathsf{v}_j \in V_{i,\mathsf{pgcnt}_i+1}$ **do**
        Set $k_j \leftarrow \mathcal{H}_P(P_i, j)$ and $T_\gamma[k_j] \leftarrow \mathsf{Enc}_{\mathsf{K_{Enc}}}(\mathsf{v}_j\|\ell_i)$
        Set $j \leftarrow j + 1$
Fill T_γ up to capacity cap with fresh $\mathsf{Enc}_{\mathsf{K_{Enc}}}(0\|0)$ entries
Send T_γ

Server:

Update T_γ

6 Evaluation

6.1 Experimental Setup

Choice of Primitives. We use HMAC-SHA256 with 32-byte keys as the PRF functions. We use HMAC-SHA256 without a key as the 1C hash functions. We use randomized AES-128-CBC as the encryption function in our implementation.

Implementation. We implement S1C and D1C in C++ with OpenSSL as the underlying cryptography library. The client and the server are implemented in the same program so there is no network layer in our experiments. We use standard C++ library functions (std::ifstream and std::ofstream) to read from and write to SSD. This means that, in the worst case, each encrypted page is stored in two physical pages by the operating system, leading to a factor of at most two in page efficiency. The implementation of S1C and D1C is publicly available at `https://github.com/Merge-SSE/Merge-SSE`.

Building Multi-maps. We pick $10K, 50K, 100K, 200K$, and $400K$ emails from the Enron email corpus to form raw databases. For each database, we use the Natural Language Toolkit[11] to extract the keywords from the emails. We build an inverted index mapping each keyword to the document identifiers of the documents containing the keyword and treat the inverted index as a multi-map where labels are the keywords and values are the matching document identifiers.

The resulting multi-maps are used directly in the experiments on S1C. For D1C, we further process the multi-maps so that they are either in the sparse regime ($N < pM$) or the dense regime ($N \geq pM$) (we determine p before running this; see Section 6.2). To obtain a multi-map in the sparse regime, we order the labels (ℓ_i) of the multi-map used for the experiment on S1C by the number of associated values ($|\mathsf{MM}[\ell_i]|$) in increasing order and add the labels to a new multi-map up until $N < pM$ no longer holds. Conversely, to obtain a multi-map in the dense regime, we order the labels of the multi-map used for the experiment on S1C by the number of associated values in decreasing order and continue adding the labels to a new multi-map until $N \geq pM$ no longer holds. The size of the multi-maps used in our experiments can be found in Table 2.

For the experiments used to determine the optimal page size (Section 6.2), we fixed δ to 0 and 4 for S1C and D1C, respectively, to allow for fair comparison of the different page sizes. For all other experiments, we selected the smallest δ such that the hash tables have reasonable loads; we report the δ values used for each dataset in Table 2.

Concretely, we set the size of a label to 16 bytes ($|\mathcal{L}|_b = 16$ bytes) and the size of a value to 16 bytes ($|\mathcal{V}|_b = 16$ bytes) with appropriate padding.

Hash Table Instantiation. The hash tables used in S1C and D1C are both implemented as cuckoo tables. For S1C, the cuckoo tables are implemented with fingerprints. That is, for $T_{\mathsf{full}}[P_{i,j}] \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(V_{i,j})$, the payload stored in the cuckoo table is $\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(V_{i,j}) \,\|\, P_{i,j}$. As a consequence, when retrieving the entry with hash key $P_{i,j}$, the server can look up the entry in the clear (even though there are two possible locations for the payload). Thus, retrieving entries has optimal bandwidth.

In contrast, for D1C, the payload of $T_{\mathsf{full}}[P_{i,j}] \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(V_{i,j})$ is implemented as $\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(V_{i,j} \,\|\, P_{i,j})$, thus, not exposing the key as described in Appendix E.2. thus, not exposing the key. The consequence is that the server must send two entries of a cuckoo table for each search query, resulting in a bandwidth overhead (and the client's computational overhead) of two.

The cuckoo tables must be instantiated with concrete capacities. For capacity cap in the pseudocodes (where $\mathcal{O}(p\delta(\lambda)\,\log(N/p))$ is set to $3p\log(N/p)$ and $\mathcal{O}(\delta(\lambda)\log(N))$ is set to $3\log(N)$), we set the size of the cuckoo tables to be $2(1+\delta)\mathsf{cap}$ (2 comes from the fact that a cuckoo table contains two hash tables and $(1+\delta)\mathsf{cap}$ is the number of entries in each hash table). The choice of δ mainly affects the setup time and update time (since we must encrypt more entries during setup and updates) and it does not have an effect on the query response time. We use different δ on different multi-maps and our choices of δ are reported in Table 2.

[11] `https://www.nltk.org/`

Table 2 – General statistics of the multi-maps and δ values we used. N is the size of the multi-map. M is the number of labels. δ is the expansion factor of the cuckoo table. The two numbers in the rows for D1C are for the sparse regime ($N < pM$) and the dense regime ($N \geq pM$) respectively.

#emails	N	M	Scheme	δ
10K	332,312	12,026	S1C	0
	332,312 / 312,493	12,026 / 5,040	D1C	0 / 9
50K	1,726,354	19,391	S1C	0
	1,181,972 / 1,726,354	19,065 / 19,391	D1C	1 / 1.5
100K	4,632,953	25,664	S1C	0
	1,494,440 / 4,632,953	24,114 / 25,664	D1C	1 / 2
200K	7,366,746	27,959	S1C	0
	1,590,517 / 7,366,746	25,661 / 27,959	D1C	1 / 3.5
400K	14,265,325	32,201	S1C	0
	1,771,192 / 14,265,325	28,568 / 32,201	D1C	1 / 4

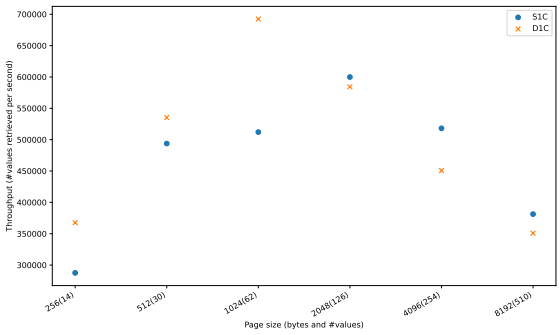


Fig. 3 – Throughput of S1C and D1C using different page sizes. The optimal page size is 512 bytes (30 values per page) for both schemes on our machine.

Experimental Environment. All benchmarks are executed on a single core of a machine with an AMD Ryzen 9 5900X @3.7 GHz with 12 cores, 64 GB of DDR4 memory, and a SATA SSD with 1.8 TB storage.

6.2 Determining the Page Size

Before running the experiments, we need to determine the optimum page size p. We do this by measuring the search throughput of S1C and D1C on the full pages using the multi-map derived from 400K emails, with page sizes of 256, 512, 1024, 2048, and 4096 bytes. (here, δ is fixed to 0 for S1C and 4 for D1C). These correspond to $p = 14, 30, 126, 254$, and 510 in our pseudocode. We query all keywords in the emails and report the throughputs in Fig. 3. We obtain the highest throughput with a page size equal to 1024 bytes ($p = 126$). We use 1024 bytes as the page size ($p = 126$) for all subsequent experiments.

6.3 Benchmarking S1C

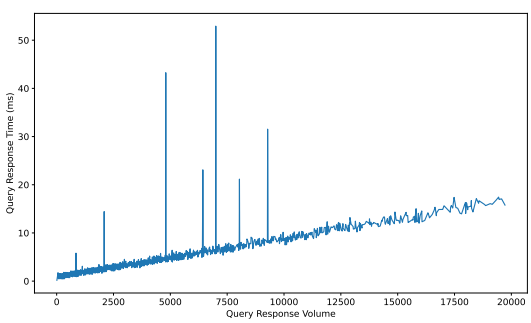
Note that in the *static* setting and *dense* regime (*i.e.*, no updates, $pM \leq N$) there is a trivial solution: split lists into sublists of size p, pad the incomplete lists to size p and store all items in a non-page-efficient EMM (*e.g.*, [9]). In the dense *regime*, this solution has optimal performance and is implicit in our *dynamic* construction in the dense regime. We stress that *dynamic* and dense EMMs are non-trivial (*cf.* Section 1). Therefore, we only evaluate S1C in the sparse regime.

Setup Time. For $N = 332312, 1726354, 4632953, 7366746, 14265325$, S1C takes $0.7, 1.7, 3.6, 5.4, 10.6$ seconds to complete the setup, respectively. The setup time does not scale linearly, as building the cuckoo tables takes random amounts of time due to reseeding when we fail to insert all elements into the table with a particular seed.

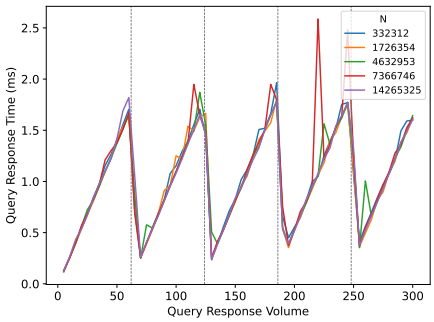
Storage. Due to space limitations, we only report client and server storage for the experiment on the largest dataset (400K emails). The client storage consists of 32 bytes of cryptographic keys. The server storage is 2.35 GB.

Query Time. We query all the keywords in the 400K documents ($N = 14265325$, $M = 32201$) and depict the query performance of S1C in Fig. 4a. It only takes 7.87 milliseconds to retrieve $21,290$ values. We also conduct experiments on smaller multi-maps and depict the results in Fig. 4b. The search time is not affected by the size of the multi-map and, as such, S1C should be considered practically efficient.

An interesting observation is that the query response time follows a zig-zag pattern (see Fig. 4b): between query response volumes kp and $(k+1)p$ for some integer k and page size p, the query response time grows linearly. However, the query response time for a query with a query response volume slightly larger than $(k+1)p$ is generally smaller than the query response time of a query with a query response volume slightly smaller than $(k+1)p$. In other words, it is more efficient to retrieve a full page of values from T_{full} than to retrieve slightly less than p values from some $T_{\mathsf{val},\cdot}$. This serves as supporting evidence that S1C is page-efficient.



(a) Search performance of S1C on a multi-map with $N = 13950793, M = 32112$.



(b) Search performance of S1C on various multi-maps up to a query response volume of 300.

Fig. 4 – Search performance of S1C.

As an independent result, we run S1C in memory and observe the same zig-zag pattern on the query response time (see Fig. 8 in Appendix G). As an independent result, we run S1C in memory and observe the same zig-zag pattern on the query response time (see full version [21] for details). This is surprising as one would expect the use of pages to not have any effect on the query response time (as RAM does not have a page-like behavior). We believe that this result is due to the concrete efficiency gain when pages are used. For simplicity, assume that the length of each value is equal to the block size of the encryption scheme used (16 bytes in the case of AES-128-CBC). When encrypting p values as a single entry in T_{full}, the payload consists of $p + 1$ ciphertext blocks (p blocks from the values and one block from the initialization vector) and one fingerprint block, so retrieving a full page in T_{full} takes one fingerprint check on the server, $p + 2$ blocks of bandwidth, and p AES decryption operations on the client. On the other hand, if p values are encrypted individually, as in $T_{\mathsf{val},\cdot}$, we obtain p payloads, each containing two ciphertext blocks (one block from the value and one from the initialization vector) and one fingerprint block. Thus, retrieving p entries stored in $T_{\mathsf{val},\cdot}$ takes p fingerprint checks on the server, $3p$ blocks of bandwidth and p AES decryption operations on the client. Even without the use of fingerprints, retrieving query responses in pages is more bandwidth-efficient. This observation could help design more efficient in-memory SSE schemes.

6.4 Benchmarking D1C

Setup Time. We use a modified setup procedure for D1C. Instead of running setup with an empty multi-map, we run it with a non-empty multi-map and parameters p, N, and M. This modification allows us to benchmark the query responses without waiting for the label-value pairs to be inserted into the EMM one by one. The setup time with a non-empty multi-map is longer than that with an empty multi-map.

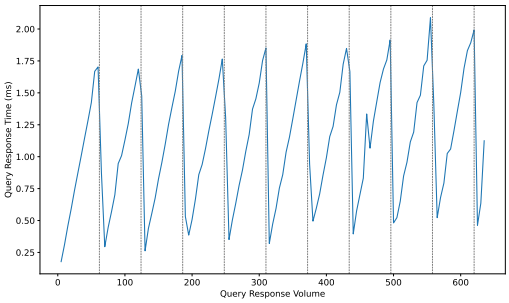
The setup time of D1C in the sparse regime ($N < pM$) for multi-maps of size between $N = 332312, M = 12026$ and $N = 1771192, M = 28568$ ranges from 45.8 to 402.6 seconds. The setup time of D1C in the dense regime ($N \geq pM$) for multi-maps of size between $N = 312493, M = 5040$ and $N = 14265325, M = 32201$ ranges from 351.2 to 1182.4 seconds. The setup time is longer in the dense regime as we used larger δ's in our experiments.

Storage. For 400K emails in the sparse regime, the client storage consists of 64 bytes of cryptographic keys, 0.8 MB of constants and counters, and 0.8 MB of content in the stashes. The server storage is 0.53 GB. In the dense regime, the client storage consists of 64 bytes of cryptographic keys, 1 MB of constants and counters, and 1 MB of content in the stashes. The server storage is 3.38 GB.

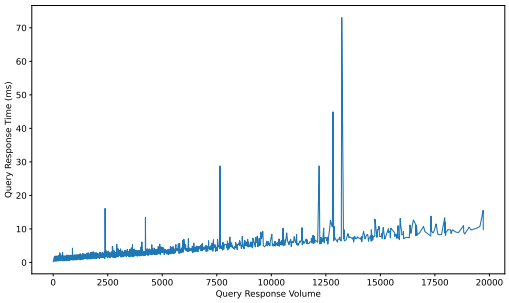
Update Time. We ran 1000 update queries on the EMMs and reported the average update times. The update time ranges from 189.4 ms to 442.5 ms for the multi-maps in the sparse regime and

ranges from 28.0 ms to 102.0 ms for the multi-maps in the dense regime. Updates required more time in the sparse regime since the bins are larger.

Query Time. We show the query response time of S1C on the multi-map derived from 400K documents ($N = 1771192, M = 28568$ in the sparse regime; $N = 14265325, M = 32201$ in the dense regime) in Figures 5a and 5b, respectively. D1C appears to be more efficient in the sparse regime as the EMMs are smaller. So D1C benefits more from file system caching and caching in the SSD. Similar to S1C, D1C displays excellent search efficiency, taking only 0.21 milliseconds to retrieve 248 values in the sparse regime, and 10.27 milliseconds to retrieve 21414 values in the dense regime.



(a) Search performance on a multi-map with $N = 1771192$, $M = 28568a$. (sparse)



(b) Search performance on a multi-map with $N = 14265325$, $M = 32201$ (dense).

Fig. 5 – Search performance of D1C.

6.5 Concrete Comparison to [41]

To the best of our knowledge, all prior works on dynamic and memory-efficient EMMs have not been implemented, except for [41]. We are unable to run the artefact in [41] out of the box but we managed to fix it ourselves. We provide a comparison of our schemes with $\mathrm{SD_a}[1C], \mathrm{SD_a}[2C]$, and L-$\mathrm{SD}_d[1C]$ below. All experiments are run with 400K emails ($N = 14265325, M = 32201$).

Setup Time. Setup of $\mathrm{SD_a}[1C], \mathrm{SD_a}[2C]$, and L-$\mathrm{SD}_d[1C]$ takes 123.9, 154.1, and 256.5 seconds, respectively.

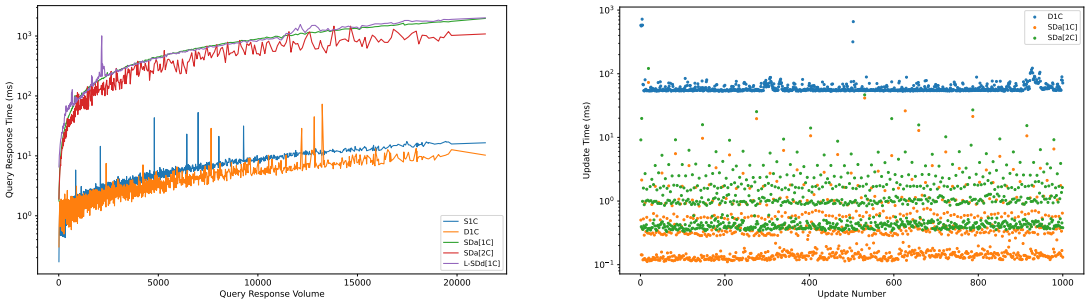
Storage. The storage cost for $\mathrm{SD_a}[1C], \mathrm{SD_a}[2C]$, and L-$\mathrm{SD}_d[1C]$ are 3.63, 4.72, and 14.50 GB, respectively.

Query Time. Figure 6a compares the query efficiencies of our schemes S1C and D1C with those of [41], namely $\mathrm{SD_a}[1C], \mathrm{SD_a}[2C]$, and L-$\mathrm{SD}_d[1C]$. In terms of query throughput, S1C and D1C are over 30× faster than the other schemes.

Update Time. We compare the update efficiency of D1C, $\mathrm{SD_a}[1C]$, and $\mathrm{SD_a}[2C]$ up to the first 1000 update queries in Figure 6b. $\mathrm{SD_a}[1C]$ and $\mathrm{SD_a}[2C]$ run a slow rebuild process when the number of updates hit a power of 2 and the maximum update time gets longer as the number of updates increases. On the other hand, D1C has a consistent update time. We were not able to obtain the update time for L-$\mathrm{SD}_d[1C]$ as it is too slow at this scale.

7 Conclusion

This work resolves an open question in the design of memory-efficient encrypted multi-maps (EMMs) by presenting two page-efficient EMM constructions with optimal search bandwidth. We introduce one static and one dynamic EMM scheme that simultaneously achieve optimal bandwidth, optimal storage efficiency, and polylogarithmic page complexity. Our dynamic construction



(a) Search performance of S1C, D1C, $\mathrm{SD}_a[1C]$, $\mathrm{SD}_a[2C]$, and L-$\mathrm{SD}_d[1C]$ on a multi-map with $N = 14265325, M = 32201$.

(b) Update performance of D1C, $\mathrm{SD}_\mathrm{a}[1C]$, and $\mathrm{SD}_\mathrm{a}[2C]$ on a multi-map with $N = 14265325, M = 32201$.

Fig. 6 – A comparison of our schemes to those from [41].

additionally satisfies forward and backward privacy. To obtain the dynamic scheme, we introduce a new data structure—the keyless cuckoo hash table—which enables client updates without leaking to the server whether an update is real or dummy; this data structure may be of independent interest. We support our constructions with formal security proofs and an extensive experimental evaluation on real-world datasets.

Our schemes exhibit a standard leakage profile. While prior work has investigated mitigating either access-pattern leakage or volume leakage, fully hiding access patterns generally requires heavyweight cryptographic primitives such as ORAM, and hiding volume is inherently at odds with achieving optimal bandwidth. One interesting direction for future work would be to explore the trade-off between partial volume hiding and moderate bandwidth overhead.

Acknowledgments. Francesca Falzon was supported by Zurich Information Security and Privacy Center (ZISC). This paper was edited for grammar using ChatGPT.

References

1. Amjad, G., Patel, S., Persiano, G., Yeo, K., Yung, M.: Dynamic volume-hiding encrypted multi-maps with applications to searchable encryption. PoPETs **2023**(1), 417–436 (Jan 2023). https://doi.org/10.56553/popets-2023-0025
2. 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
3. Asharov, G., Segev, G., Shahaf, I.: Tight tradeoffs in searchable symmetric encryption. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 407–436. Springer, Cham (Aug 2018). https://doi.org/10.1007/978-3-319-96884-1_14
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. Berenbrink, P., Friedetzky, T., Hu, Z., Martin, R.: On weighted balls-into-bins games. Theoretical Computer Science **409**(3), 511–520 (2008)
6. 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, Cham, Virtual Event (Aug 2021). https://doi.org/10.1007/978-3-030-84252-9_6
7. 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
8. 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

9. 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). https://doi.org/10.14722/ndss.2014.23264
10. 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, Berlin, Heidelberg (Aug 2013). https://doi.org/10.1007/978-3-642-40041-4_20
11. Cash, D., Ng, R., Rivkin, A.: Improved structured encryption for SQL databases via hybrid indexing. In: Sako, K., Tippenhauer, N.O. (eds.) ACNS 2021, Part II. LNCS, vol. 12727, pp. 480–510. Springer, Cham (Jun 2021). https://doi.org/10.1007/978-3-030-78375-4_19
12. 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, Berlin, Heidelberg (May 2014). https://doi.org/10.1007/978-3-642-55220-5_20
13. Chamani, J.G., Demertzis, I., Papadopoulos, D., Papamanthou, C., Jalili, R.: Graphos: Towards oblivious graph processing. Proc. VLDB Endow. **16**(13), 4324–4338 (2023). https://doi.org/10.14778/3625054.3625067, https://www.vldb.org/pvldb/vol16/p4324-chamani.pdf
14. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 577–594. Springer, Berlin, Heidelberg (Dec 2010). https://doi.org/10.1007/978-3-642-17373-8_33
15. 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
16. Demertzis, I., Chamani, J.G., Papadopoulos, D., Papamanthou, C.: Dynamic searchable encryption with small client storage. In: NDSS 2020. The Internet Society (Feb 2020). https://doi.org/10.14722/ndss.2020.24423
17. 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, Cham (Aug 2018). https://doi.org/10.1007/978-3-319-96884-1_13
18. Demertzis, I., Papadopoulos, S., Papapetrou, O., Deligiannakis, A., Garofalakis, M.N.: Practical private range search revisited. In: Özcan, F., Koutrika, G., Madden, S. (eds.) Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016. pp. 185–198. ACM (2016). https://doi.org/10.1145/2882903.2882911, https://doi.org/10.1145/2882903.2882911
19. 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)
20. Falzon, F., Ghosh, E., Paterson, K.G., Tamassia, R.: PathGES: An efficient and secure graph encryption scheme for shortest path queries. In: Luo, B., Liao, X., Xu, J., Kirda, E., Lie, D. (eds.) ACM CCS 2024. pp. 4047–4061. ACM Press (Oct 2024). https://doi.org/10.1145/3658644.3670305
21. Falzon, F., Gui, Z., Reichle, M.: Page-efficient encrypted multi-maps: New techniques for optimal search bandwidth. Cryptology ePrint Archive, Report 2025/1621 (2025), https://eprint.iacr.org/2025/1621
22. Falzon, F., Markatou, E.A., Akshima, Cash, D., Rivkin, A., Stern, J., Tamassia, R.: Full database reconstruction in two dimensions. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 443–460. ACM Press (Nov 2020). https://doi.org/10.1145/3372297.3417275
23. Falzon, F., Markatou, E.A., Espiritu, Z., Tamassia, R.: Range search over encrypted multi-attribute data. Proc. VLDB Endow. **16**(4), 587–600 (2022). https://doi.org/10.14778/3574245.3574247, https://www.vldb.org/pvldb/vol16/p587-falzon.pdf
24. Fuller, B., Varia, M., Yerukhimovich, A., Shen, E., Hamlin, A., Gadepally, V., Shay, R., Mitchell, J.D., Cunningham, R.K.: SoK: Cryptographically protected database search. In: 2017 IEEE Symposium on Security and Privacy. pp. 172–191. IEEE Computer Society Press (May 2017). https://doi.org/10.1109/SP.2017.10
25. Grubbs, P., Lacharité, M.S., Minaud, B., Paterson, K.G.: Pump up the volume: Practical database reconstruction from volume leakage on range queries. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018. pp. 315–331. ACM Press (Oct 2018). https://doi.org/10.1145/3243734.3243864
26. 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
27. Gui, Z., Johnson, O., Warinschi, B.: Encrypted databases: New volume attacks against range queries. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 361–378. ACM Press (Nov 2019). https://doi.org/10.1145/3319535.3363210

28. Gui, Z., Paterson, K.G., Patranabis, S., Warinschi, B.: SWiSSSE: System-wide security for searchable symmetric encryption. PoPETs **2024**(1), 549–581 (Jan 2024). https://doi.org/10.56553/popets-2024-0032
29. Kamara, S., Moataz, T.: Boolean searchable symmetric encryption with worst-case sub-linear complexity. In: Coron, J.S., Nielsen, J.B. (eds.) EUROCRYPT 2017, Part III. LNCS, vol. 10212, pp. 94–124. Springer, Cham (Apr / May 2017). https://doi.org/10.1007/978-3-319-56617-7_4
30. 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, Cham (Dec 2018). https://doi.org/10.1007/978-3-030-03326-2_6
31. Kamara, S., Moataz, T.: Computationally volume-hiding structured encryption. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part II. LNCS, vol. 11477, pp. 183–213. Springer, Cham (May 2019). https://doi.org/10.1007/978-3-030-17656-3_7
32. Kamara, S., Moataz, T., Park, A., Qin, L.: A decentralized and encrypted national gun registry. In: 2021 IEEE Symposium on Security and Privacy. pp. 1520–1537. IEEE Computer Society Press (May 2021). https://doi.org/10.1109/SP40001.2021.00072
33. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Sadeghi, A.R. (ed.) FC 2013. LNCS, vol. 7859, pp. 258–274. Springer, Berlin, Heidelberg (Apr 2013). https://doi.org/10.1007/978-3-642-39884-1_22
34. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) ACM CCS 2012. pp. 965–976. ACM Press (Oct 2012). https://doi.org/10.1145/2382196.2382298
35. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman and Hall, CRC Press, third edn. (2014)
36. Kellaris, G., Kollios, G., Nissim, K., O'Neill, A.: Generic attacks on secure outsourced databases. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016. pp. 1329–1340. ACM Press (Oct 2016). https://doi.org/10.1145/2976749.2978386
37. Kirsch, A., Mitzenmacher, M., Wieder, U.: More robust hashing: Cuckoo hashing with a stash. SIAM Journal on Computing **39**(4), 1543–1561 (2010)
38. 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). https://doi.org/10.14722/ndss.2017.23394
39. Minaud, B., Reichle, M.: Dynamic local searchable symmetric encryption. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO 2022, Part IV. LNCS, vol. 13510, pp. 91–120. Springer, Cham (Aug 2022). https://doi.org/10.1007/978-3-031-15985-5_4
40. Minaud, B., Reichle, M.: Hermes: I/O-efficient forward-secure searchable symmetric encryption. In: Guo, J., Steinfeld, R. (eds.) ASIACRYPT 2023, Part VI. LNCS, vol. 14443, pp. 263–294. Springer, Singapore (Dec 2023). https://doi.org/10.1007/978-981-99-8736-8_9
41. Mondal, P., Chamani, J.G., Demertzis, I., Papadopoulos, D.: I/O-efficient dynamic searchable encryption meets forward & backward privacy. In: Balzarotti, D., Xu, W. (eds.) USENIX Security 2024. USENIX Association (Aug 2024), https://www.usenix.org/conference/usenixsecurity24/presentation/mondal
42. Naveed, M., Prabhakaran, M., Gunter, C.A.: Dynamic searchable encryption via blind storage. In: 2014 IEEE Symposium on Security and Privacy. pp. 639–654. IEEE Computer Society Press (May 2014). https://doi.org/10.1109/SP.2014.47
43. Pagh, R., Rodler, F.F.: Cuckoo hashing. Journal of Algorithms **51**(2), 122–144 (2004)
44. Patel, S., Persiano, G., Yeo, K., Yung, M.: Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 79–93. ACM Press (Nov 2019). https://doi.org/10.1145/3319535.3354213
45. Richa, A.W., Mitzenmacher, M., Sitaraman, R.: The power of two random choices: A survey of techniques and results. Combinatorial Optimization **9**, 255–304 (2001)
46. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: NDSS 2014. The Internet Society (Feb 2014). https://doi.org/10.14722/ndss.2014.23298
47. 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), https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/zhang

A Concrete Comparison

In this section, we provide concrete page efficiency and communication efficiency comparisons between S1C and prior works on static memory-efficient EMMs, and between D1C and prior works on dynamic memory-efficient EMMs, respectively.

In these comparisons, we fix the size of the multimap to $N = 13,950,793$ and the page size to $p = 32$. And we compare the concrete overhead of the schemes with respect to the number of values retrieved in the query. The results will favour S1C and D1C even more if the page size is set to something larger (that is more common in modern SSDs).

We measure communication efficiency as the number of records retrieved by the given scheme for a query that is supposed to retrieve x records, divided by x. The comparison between selected schemes is shown in Figure 7a. The schemes that are omitted in the plots have worse communication efficiency than the ones that have been shown (e.g., 1C is worse than 2C). An optimal EMM should have communication efficiency 1 regardless of the number of records retrieved. The comparison shows that S1C and D1C are significantly better than the other schemes in terms of the communication efficiency.

We measure page efficiency as the number of pages accessed by the EMM for a query that is supposed to retrieve x records, divided by $\lceil x/p \rceil$, where p is the page size. The comparison between selected schemes is shown in Figure 7b. The schemes that are omitted in the plots have worse page efficiency than the ones that have been shown (e.g., 1C is worse than 2C). An optimal EMM should have page efficiency 1 regardless of the number of records retrieved. The comparison shows that while S1C and D1C are not optimal, they are practically acceptable.

B Proof for Weighted One-Choice

We provide the proof of Lemma 1. We first recall some useful lemmas. The first is one of the formulations of the Chernoff bound. The second lemma is a first-moment argument borrowed from [39]. The third lemma gives upper bounds on the most-loaded bin in weighted one-choice process in expectation.

Lemma 2 (Chernoff's Bound). *Suppose that $X_1, \ldots, X_n$ are independent random variables taking values in $\{0,1\}$. Let X denote their sum and let $\mu = \mathbb{E}(X)$ denote the expectancy of X. Then for any $\delta > 0$, it holds that*

$$\Pr[X \geq (1+\delta)\mu] \leq \exp\left(-\frac{\delta^2 \mu}{2+\delta}\right)$$

Lemma 3 ([39]). *Let $N \in \mathbb{N}$ with $N = \mathsf{poly}(\lambda)$. For any integer-valued random variable $X \in [0, N]$ and any $R_N > 0$:*

$$\Pr[X > R_N] = \mathsf{negl}(\lambda) \quad \textit{iff} \quad \mathbb{E}(\max(X - R_N, 0)) = \mathsf{negl}(\lambda).$$

Lemma 4 ([5], Corollary 3.5). *Let $N \in \mathbb{N}$ and $c \in [0,1]_{\mathbb{R}}$. Let $\tilde{w} = (c)_{i \in [\tilde{n}]}$ and $w = (w_i)_{i \in [n]}$ be non-negative vectors. Let $\sum_{i=1}^{n} w_i \leq c \cdot \tilde{n}$ and $w_i \leq c$ for all $i \in [n]$.*

Denote by X_{mlb} (resp. $\tilde{X}_{\mathsf{mlb}}$) the random variable indicating the load of the most loaded bin after throwing n balls with weights w (resp. N balls with weights $\tilde{x}$) uniformly and independently at random into m bins. For any positive $R \in \mathbb{R}$, it holds that

$$\mathbb{E}[\max(X_{\mathsf{mlb}} - R, 0)] \leq \mathbb{E}[\max(\tilde{X}_{\mathsf{mlb}} - R, 0)].$$

Note that we simplified the formulation of [5] and adapted it to our needs. Roughly, [5] requires that $\tilde{w}$ *majorizes* w, which holds if both are chosen as in Lemma 4. Then, [5] shows that $\mathbb{E}[\tilde{X}_{\mathsf{mlb}}] \geq \mathbb{E}[X_{\mathsf{mlb}}]$ using a convexity argument. As $f(X) = \max(X - R, 0)$ is increasing, it is straightforward to adapt their proof to our formulation. For the upper bound on the most loaded bin in weighted 1C, our formulation is sufficient. We are now ready to prove the upper bound.

Proof (Proof of Lemma 1). Let X_{mlb} be the random variable denoting the load of the most loaded bin at some point during the sequence. Let $R = 3\lceil N \rceil / m$. We show that $\Pr[X_{\mathsf{mlb}} > R+1] = \mathsf{negl}(\lambda)$. Note that this is implied by $\Pr[\lceil X_{\mathsf{mlb}} \rceil > R + 1] = \mathsf{negl}(\lambda)$.

As $\lceil X_{\mathsf{mlb}} \rceil < \lceil N \rceil$ is integer-valued, we can apply Lemma 3. Now, we need to show that $\mathbb{E}(\max(\lceil X_{\mathsf{mlb}} \rceil - (R+1), 0)) = \mathsf{negl}(\lambda)$. As $\lceil X_{\mathsf{mlb}} \rceil < X_{\mathsf{mlb}} + 1$, it suffices to show that the expression $\mathbb{E}(\max(X_{\mathsf{mlb}} - R, 0))$ is negligible in λ.

By Lemma 4, this quantity is upper bounded by $\mathbb{E}[\max(\tilde{X}_{\mathsf{mlb}} - R, 0)]$, where $\tilde{X}_{\mathsf{mlb}}$ is the load of the most loaded bin for $\lceil N \rceil$ balls of weight 1. Note that $\tilde{X}_{\mathsf{mlb}} \leq \lceil N \rceil$ is a non-negative integer variable. After a final application of Lemma 3, it remains to show that $\Pr[\tilde{X}_{\mathsf{mlb}} \geq R] = \mathsf{negl}(\lambda)$. For the last quantity, it is sufficient to analyze classical 1C without weights.

Let X_i be the random variable that denotes the number of balls in bin B_i. Every ball has a chance of $1/m$ to be assigned to B_i and there are $\lceil N \rceil$ balls. Thus, we have $\mathbb{E}[X_i] = \lceil N \rceil / m$. Note that X_i can be expressed as sum of independent random variables taking values in $\{0, 1\}$. Applying Lemma 2 with constant 2, we obtain:

$$\begin{aligned} &\Pr[X_i \geq 3\,\mathbb{E}[X_i]] \leq \exp\left(-\frac{4 \cdot \mathbb{E}[X_i]}{4}\right) \\ \implies &\Pr[X_i \geq R] \leq \exp(-\lceil N \rceil / m) = \mathsf{negl}(\lambda), \end{aligned}$$

by assumption. Finally, a union bound over all bins yields that $\Pr[\tilde{X}_{\mathsf{mlb}} \geq R] = \mathsf{negl}(\lambda)$. The statement follows because

$$\exp(-\delta(\lambda) \log N) = \mathsf{negl}(\lambda)$$

if $N \geq \lambda$ and $\delta(\lambda) = \omega(1)$.

C On using other Allocation Mechanisms

Instead of using 1C as the underlying allocation algorithm in our constructions, one might consider applying our approach with other common variants from the literature, such as two-choice allocation [45] or cuckoo hashing [43]. We discuss below why this fails even in the static setting.

Two-choice allocation. In two-choice allocation (2C), there are m bins, denoted $B_1, \ldots, B_m$, each with capacity $\widetilde{\mathcal{O}}(\log \log N)$, where $m = N/\widetilde{\mathcal{O}}(\log \log N)$. When allocating a ball of weight w_i, two random bins $B_{i,1}$ and $B_{i,2}$ are chosen uniformly at random. The ball is then placed into the bin with the smaller total weight among $B_{i,1}$ and $B_{i,2}$ at the time of insertion. Variants of two-choice allocation are known to have provable correctness guarantees with these parameters [2, 39]; nevertheless, even for the simple variant sketched above, our approach runs into issues.
Recall that in the context of memory efficiency, balls represent values that match the same label ℓ, that is, values in V_ℓ. In particular, to achieve optimal bandwidth, we must access the exact locations of the values $v \in V_\ell$ within either $B_{i,1}$ or $B_{i,2}$. While we can structure these bins as hash tables to allow direct access, as in our work, doing so reveals whether the values in V_ℓ were placed in bin $B_{i,1}$ or $B_{i,2}$. This information is leaked to the server and depends on the multi-map's distribution.
For example, if the database consists of many balls of weight 1, the balls will be distributed roughly evenly among the bins. In contrast, if the database contains many balls of weight 1 and a single large ball of weight 10, then the smaller balls will be biased away from the bin containing the large ball, as that bin already has a large total weight if the large ball is inserted early.From the adversary's perspective, however, these two distributions should be indistinguishable when querying only balls of weight 1, since whether the database contains one large ball or many small balls should remain hidden.
Cuckoo hashing. In cuckoo hashing, there are $m = (1 + \varepsilon)N$ bins in total, each with capacity 1. As in 2C, two bins are chosen at random per ball, except that allocation is optimized globally via an eviction strategy (cf. Appendix E.2). It was shown in [6] that there exists an optimal allocation strategy for weighted balls. However, as in 2C, the exact bin in which a ball is placed (among its two choices) depends on the multi-map's distribution, and our proof strategy therefore fails to apply.

While the above intuition does not constitute a concrete attack, it highlights a form of non-standard leakage arising from distribution-dependent allocation. As demonstrated by prior work (*e.g.*, [36, 26, 25, 22, 27]), non-standard or seemingly benign leakage of this kind can often be leveraged to carry out leakage-abuse attacks, especially when combined with auxiliary knowledge.

D Details for our Memory-Efficient EMMs

In this section, we provide detailed overviews of our constructions in Sections 3 and 5.

D.1 Detailed Description of our Static Scheme

Let us give a detailed overview of the scheme in Section 3. A description in pseudocode is given in Algorithm 2.

$\mathsf{S1C}$ makes use of two hash functions: a token hash function $\mathcal{H}_P : \{0,1\}^* \to [2^{2\lambda}]$ that maps to the key space of T_{full} and a one-choice hash function $\mathcal{H}_{\mathsf{1C}} : \{0,1\}^* \to [0, m-1]$ that maps to the space of bin identifiers. The scheme also makes use of a PRF, denoted PRF, that maps to the space $\{0,1\}^{2\lambda} \times \{0,1\}^{\lceil \log N \rceil} \times \{0,1\}^{2\lambda}$.

To encrypt a multimap MM, the client starts by executing the KeyGen algorithm, which takes as input security parameter λ and which samples a PRF key, $\mathsf{K}_{\mathsf{PRF}}$, and an encryption key, $\mathsf{K}_{\mathsf{Enc}}$.

Next, the client executes Setup, which takes as input the security parameter λ, secret key K, upper bounds N and M, and the multimap MM. The client initializes two tables: T_{len} with N entries each of size $\lceil \log N \rceil$, and T_{full} with $\lceil N/p \rceil$ entries each of size $p \cdot |\mathcal{V}|_b$. It computes $m \leftarrow \lceil N/(p \cdot \delta(\lambda) \cdot \log(N/p)) \rceil$ and $\mathsf{cap} \leftarrow \mathcal{O}(p \cdot \delta(\lambda) \log(N/p))$. Then, it initializes m tables, $T_{\mathsf{val},\gamma}$ for $\gamma \in [0, m-1]$, each with cap random entries.

For each label $\ell_i \in \mathsf{MM}$, the client retrieves the associated values $V_i = \mathsf{MM}[\ell_i]$ and computes the number of associated values, $\mathsf{cnt}_i = |V_i|$, and the number of pages needed to store those values, $\mathsf{pgcnt}_i = \lceil \mathsf{cnt}_i/p \rceil$. . The search tokens K_i and P_i' (for T_{len} and T_{full}, respectively), and the mask μ_i are then derived by evaluating the PRF on the label, i.e., $(K_i, \mu_i, P_i') \leftarrow F_{\mathsf{K}_{\mathsf{PRF}}}(\ell_i)$. The number of values is blinded and stored as $T_{\mathsf{len}}[K_i] \leftarrow \mathsf{cnt}_i \oplus \mu_i$. This will later be used at query time to help the server determine how many values must be retrieved. The values associated with label ℓ_i are partitioned into pgcnt_i many lists, which we denote as $V_{i,j}$; each list is of length p (the page size). Then, for all $j = 1, \ldots, \lfloor \mathsf{cnt}_i/p \rfloor$, we compute the hash output $P_{i,j} \leftarrow \mathcal{H}_P(P_i', j)$ and store the encrypted full pages as $T_{\mathsf{full}}[P_{i,j}] \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(V_{i,j})$.

It remains to compute the tables $T_{\mathsf{val},0}, \ldots, T_{\mathsf{val},\mathsf{m}-1}$. For each label ℓ_i, the client computes the hash output $P_{i,\mathsf{pgcnt}_i} \leftarrow \mathcal{H}_P(P_i', \mathsf{pgcnt}_i)$ and sets $\gamma \leftarrow \mathcal{H}_{\mathsf{1C}}(P_{i,\mathsf{pgcnt}_i})$. The value γ specifies which bin to insert the corresponding values into. The client initializes $j \leftarrow 1$. Then, for each value v in the final partition V_{i,pgcnt_i}, the client: (1) computes the location $P_{i,\mathsf{pgcnt}_i,j} \leftarrow \mathcal{H}_P(P_i', \mathsf{pgcnt}_i, j)$, (2) encrypts the value and stores it in the table as $T_{\mathsf{val},\gamma}[P_{i,\mathsf{pgcnt}_i,j}] \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(\mathsf{v})$, and (3) increments j.

Setup concludes with the client filling the remaining empty entries in T_{full} and T_{len} with random values and outputting $\mathsf{EMM} = (T_{\mathsf{len}}, T_{\mathsf{full}}, (T_{\mathsf{val},0}, \ldots, T_{\mathsf{val},\mathsf{m}-1}))$ to the server.

To make a query for a label ℓ_i, the client and server jointly execute Search. The search protocol takes the private key K and the label ℓ_i as input from the client, and EMM as input from the server. The client starts by computing the search token as $(K_i, \mu_i, P_i') \leftarrow \mathsf{PRF}_{\mathsf{K}_{\mathsf{PRF}}}(\ell_i)$ and returning it to the server.

On input of the search token, the server retrieves the (encrypted) values as follows. First it initializes an empty set R that will store the response. The number of values is retrieved as $\mathsf{cnt}_i \leftarrow T_{\mathsf{len}}[K_i] \oplus \mu_i$ and cnt_i is then used to compute pgcnt_i. For each $j = 1, \ldots, \lfloor \mathsf{cnt}_i/p \rfloor$, the hash $P_{i,j} \leftarrow \mathcal{H}_P(P_i', j)$ is computed and the full (encrypted) page $T_{\mathsf{full}}[P_{i,j}]$ is added to R. The server then uses P_i' and pgcnt_i to derive P_{i,pgcnt_i} and γ. Then, for each $j \in [\mathsf{cnt}_i \bmod p]$, it computes the value's location $P_{i,\mathsf{pgcnt}_i,j} \leftarrow \mathcal{H}_P(P_i', \mathsf{pgcnt}_i, j)$ and adds $T_{\mathsf{val},\gamma}[P_{i,\mathsf{pgcnt}_i,j}]$ to R. The server returns R to the client.

Finally, the client decrypts the values in R using $\mathsf{K}_{\mathsf{Enc}}$.

D.2 Detailed Description of our Dynamic EMM

Our dynamic, page-efficient EMM scheme operates in two regimes, however we describe them as one unified scheme.

Similar to our static scheme, $\mathsf{D1C}$ makes use of two hash functions: a token hash function $\mathcal{H}_P : \{0,1\}^* \to [2^{2\lambda}]$ that maps to the key space of the bins and a one-choice hash function $\mathcal{H}_{\mathsf{1C}} : \{0,1\}^* \to [0, m-1]$ that maps to the space of bin identifiers. The scheme also uses a

PRF,PRF, that maps to the space $\{0,1\}^{2\lambda} \times \{0,1\}^{\lceil \log N \rceil} \times \{0,1\}^{2\lambda}$. It also makes black-box use of a EMM scheme with dummy updates, which we denote by Σ.

Key generation. To encrypt a multimap MM, the client must first generate the secret key using KeyGen. This algorithm takes as input the security parameter λ and samples a PRF key, $\mathsf{K}_{\mathsf{PRF}}$ and an encryption key, $\mathsf{K}_{\mathsf{Enc}}$. It also executes $\Sigma.\mathsf{KeyGen}(1^\lambda)$ to generate the key K_Σ for the underlying EMM. It returns $(\mathsf{K}_{\mathsf{Enc}}, \mathsf{K}_{\mathsf{PRF}}, \mathsf{K}_\Sigma)$ to the client.

Setup. To setup the EMM, the client executes Setup which takes as input the secret key K and upper bounds on the EMM size, N, and number of labels, M. It is at this point that the scheme description for the two regimes diverges:

Regime $N < pM$: In this regime we define the function f as a one-choice allocation ($\mathsf{1C}$) function; f is used to select which bin to store the values of a given label. Each bin is of size $\mathsf{cap} = p \cdot \widetilde{\mathcal{O}}(\log N/p)$ and there are $m = \mathcal{O}(N/\mathsf{cap})$ bins in total.

Regime $N \geq pM$: In contrast, this regime selects the bin index using an injective choice function f. Each of size $\mathsf{cap} = p$ and there are $m = M$ bins in total.

From here the two different regimes proceed similarly. The client initializes five tables which are all stored client-side as part of the state: T_{len} which stores the number of values associated with each label, two client buffers, $\mathsf{CB}_{\mathsf{new}}$ and $\mathsf{CB}_{\mathsf{out}}$, and two buffers to store full pages, $\mathsf{CFP}_{\mathsf{new}}$ and $\mathsf{CFP}_{\mathsf{out}}$. Next, the client initializes m bins, $\{T_i\}_{m\in[0,m-1]}$, and fills each of them with cap many encryptions of zeros, i.e., $\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(0\|0)$. The client then initializes a counter that maintains the total number of life-time updates, $\mathsf{ctr} = 0$, a counter that maintains the number of full pages added in the current epoch, $\mathsf{num} = 0$, and samples a random permutation π on M. Finally, if runs $\Sigma.\mathsf{Setup}$ to obtain an (empty) EMM, $\mathsf{EMM}_{\mathsf{full}}$, which will store the full pages on the server side and sends $\mathsf{EMM} = (\mathsf{EMM}_{\mathsf{full}}, (T_0, ..., T_{m-1}))$ to the server.

Update. To add a label-value pair (ℓ, v), to the EMM the client and server execute the Update protocol as follows. On the client-side, the counter ctr is incremented and the pair is added to the client buffer as $\mathsf{CB}_{\mathsf{new}}[\gamma] \leftarrow (\ell, v)$ for $\gamma = f(\ell)$ (recall, that f is defined as the one choice function for regime $N < pM$ and as an injective function otherwise). The number of values $T_{\mathsf{len}}[\ell]$ is also incremented.

If $\mathsf{ctr} \equiv 0 \bmod M$, then a new epoch starts; the client empties buffers $\mathsf{CB}_{\mathsf{new}}$ and $\mathsf{CFP}_{\mathsf{new}}$ into $\mathsf{CB}_{\mathsf{out}}$ and $\mathsf{CFP}_{\mathsf{out}}$, respectively, re-initializes $\mathsf{num} = 0$, and samples a fresh permutation π.

It sets $\gamma \equiv \mathsf{ctr} - 1 \bmod M$ and retrieves $(\ell, V) \leftarrow \mathsf{CFP}_{\mathsf{out}}[\gamma + 1]$. If this pair is non-empty, (i.e., there is a full page to be flushed to Σ) then we run $\Sigma.\mathsf{Update}$ to add (ℓ, V) to $\mathsf{EMM}_{\mathsf{full}}$. Otherwise, the pair is empty and we execute a dummy query via $\Sigma.\mathsf{DummyUpdate}$. If $f \geq m$ then return. Finally, the client sends the index γ to the server.

The server responds by sending the bin T_γ back to the client who first decrypts the bin as $V'_\gamma := (v'_i \| \ell'_i)_{i=1}^{\mathsf{cap}} \leftarrow \mathsf{Dec}_{\mathsf{K}_{\mathsf{Enc}}}(T_\gamma)$ and removes any padding of the form $0\|0$. The contents of $\mathsf{CB}_{\mathsf{out}}$ are then emptied into V'_γ and V'_γ is parsed as a multimap $\mathsf{MM}_\gamma = \{(\ell'_i, (v_1, ..., v_{n_{\ell'_i}}))\}_i$. Bin T_γ is re-initialzed as an empty table with cap entries and the epoch number is set to $\mathsf{epoch} \leftarrow \lceil \mathsf{ctr}/M \rceil$.

For each label $\ell_i \in \mathsf{MM}_\gamma$ the client does the following. It retrieves the corresponding values V_i, sets the total number of values as $\mathsf{cnt}_i = |V_i|$, and sets the total page count as $\mathsf{pgcnt}_i = \lfloor \mathsf{cnt}_i/p \rfloor$. The set of values V_i is partitioned into full pages $V_{i,j}$ each of size p, for $j \leq \mathsf{pgcnt}_i$ and (potentially) a final, incomplete page $V_{i,\mathsf{pgcnt}_{i+1}}$. For each full page $V_{i,j}$, we increment num and insert the full page and its corresponding label as $\mathsf{CFP}_{\mathsf{new}}[\pi(\mathsf{num})] \leftarrow (\ell_i, V_{i,j})$. For each value v_j in the incomplete page, if it exists, we derive the value $k_i \leftarrow \mathcal{H}_P(P_i, j)$ and insert the encrypted value-label pair into the bin as $T_\gamma[k_j] \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(v_j \| \ell_i)$.

The client concludes by padding up T_γ with fresh $\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(0\|0)$ encryptions and sending it back to the server who then updates EMM accordingly.

Search. To query the EMM for a label ℓ, the client and server execute the following protocol. The client starts by retrieving all of the full pages from the EMM with dummy updates, Σ, i.e., $R_{\mathsf{full}} \leftarrow \Sigma.\mathsf{Search}(\mathsf{K}_\Sigma, \ell; \mathsf{EMM}_{\mathsf{full}})$. It then computes the bin index $\gamma \leftarrow f(\ell)$ and computes the number of values stored in bin T_γ as $\mathsf{cnt} \leftarrow T_{\mathsf{len}}[\ell] \bmod p$.

If $\mathsf{cnt} = \perp$ or $\mathsf{cnt} = 0$, then there are no additional values to retrieve and the client simply returns R_{full}. Otherwise, it computes $\mathsf{epoch} \leftarrow \lceil \mathsf{ctr}/M \rceil$. If $(\gamma > \mathsf{ctr} \bmod M)$, then T_γ was not updated this

epoch and we decrement epoch. We then derive the search token, $P_\ell \leftarrow F_{\mathsf{K}_{\mathsf{PRF}}}(\mathsf{epoch}, \ell)$, and send $(P_\ell, \mathsf{cnt}, \gamma)$ to the server.

The server initializes an empty set R and for all $i \in [\mathsf{cnt}]$, computes $k_i \leftarrow \mathcal{H}_P(P_\ell, j)$, retrieves the encrypted pair from $T_\gamma[k_i]$, and adds it to R. When all encrypted pairs have been retrieved, the server sends R to the client. To conclude, the client decrypts R, combines it with R_{full}, and retrieves andy remaining values associated with ℓ from the client state.

E Dummy1C

In this section, we formalize our EMM scheme with dummy updates, $\mathsf{Dummy1C}$, which supports $\mathsf{poly}(\lambda)$ dummy updates with $\widetilde{\mathcal{O}}(\log N)$ communication for updates and *constant* search communication. This gives a different tradeoff to $\mathsf{Dummy}(\Sigma)$ [40] which has $\widetilde{\mathcal{O}}(\log \log N)$ communication for both updates and searches.

We first give a formal definition of EMM with dummy updates. Then, we recap the framework in [40] and describe our construction based on our keyless cuckoo hash tables (Appendix E.2).

E.1 Security of EMMs with Dummy Updates

The notion of searchable encryption supporting dummy updates was introduced in [40]. We adapt their definition to EMMs below.

An EMM schemes supports dummy updates if its interface is equipped with an additional protocol, $\mathsf{DummyUpdate}$, which takes as input only the client's master key K. The Setup algorithm additionally receives a parameter D, which specifies an upper bound on the total number of dummy updates. We omit unecessary parameters for simplicity.

Intuitively, an EMM scheme with dummy updates is secure if it is secure in the same sense as a normal EMM scheme with the added requirement that dummy updates should be indistinguishable from real updates from the server's perspective.

Definition 9 (Adaptive Semantic Security with Dummy Updates). *Let Σ be an EMM scheme supporting dummy updates. Let $q \in \mathbb{N}$ be such that $q = \mathsf{poly}(\lambda)$ and $\mathcal{L} = (\mathcal{L}_{\mathsf{Stp}}, \mathcal{L}_{\mathsf{Srch}}, \mathcal{L}_{\mathsf{Updt}})$ be a leakage function. We say that Σ is $\mathcal{L}$-adaptively secure with support for dummy updates if for all honest-but-curious PPT adversaries $\mathcal{A}$, there exists a (stateful) PPT simulator Sim such that*

$$\begin{aligned}&|\Pr[\text{EMMReal}^{\mathsf{d,adp}}_{\Sigma,\mathcal{A}}(\lambda) = 1] \\ &\quad - \Pr[\text{EMMIdeal}^{\mathsf{d,adp}}_{\Sigma,\mathsf{Sim},\mathcal{L}}(\lambda) = 1]| = \mathsf{negl}(\lambda),\end{aligned}$$

where the games $\text{EMMReal}^{\mathsf{d,adp}}_{\Sigma,\mathcal{A}}$ and $\text{EMMIdeal}^{\mathsf{d,adp}}_{\Sigma,\mathcal{A},\mathcal{L}}$ are defined in Algorithm 4.

E.2 Keyless Cuckoo Hash Table

To address this search communication blowup, we propose a new EMM scheme, $\mathsf{Dummy1C}$ (described in Appendix E.3), which leverages a new data-structure: keyless cuckoo hash tables. We first recap cuckoo hashing which serves as the basis for our hash table.

Cuckoo hashing [43]. In cuckoo hashing, n items are inserted into two cuckoo tables T_1, T_2 each of capacity $(1+\varepsilon)n$, for some arbitrary constant $\varepsilon > 0$.

Each table cell can contain at most one item. For each item to be inserted, two positions $\alpha, \beta \leftarrow \{0, \ldots, (1+\varepsilon)n - 1\}$ are chosen independently and uniformly at random (e.g., by hashing a value of the item). Each value, bid, is then inserted into either $T_1[\alpha]$ or $T_2[\beta]$ according to the following process: if one of the cells $T_1[\alpha]$ or $T_2[\beta]$ is free, bid is placed in the free cell, else bid replaces the exisiting item bid' in $T_1[\alpha]$ and bid' is moved to its other possible cell, potentially creating a chain reaction. The probability that all n items could be inserted is bounded by $\mathcal{O}(1/n)$. In case of failure, new hash functions are chosen and the allocation process is restarted; this event is called a *rehash*. Pagh and Rodler show that cuckoo hashing allocates n items in expected time $\mathcal{O}(n)$, including the cost of rehashes.

Algorithm 4 Semantic security of dummy updates (Definition 9).

$\text{EMMREAL}^{\mathsf{d,adp}}_{\Sigma,\mathcal{A}}$

$(N, M, D, \mathsf{MM}, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}(1^\lambda)$
$\mathsf{K} \leftarrow \Sigma.\mathsf{KeyGen}(1^\lambda)$
$\mathsf{EMM} \leftarrow \Sigma.\mathsf{Setup}(\mathsf{K}, N, M, D, \mathsf{MM})$
Send EMM to $\mathcal{A}$
for $1 \le i \le q$ **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 $\mathsf{in}_i = \ell_i$
    $\Sigma.\mathsf{Search}_C(\mathsf{K}, \ell_i) \leftrightarrow \mathcal{A}(\mathsf{st}_{\mathcal{A}})$
  else if $\mathsf{op}_i \in \{\mathsf{add}, \mathsf{del}\}$ **then**
    Parse $\mathsf{in}_i = (\ell_i, v_i)$
    $\Sigma.\mathsf{Update}_C(\mathsf{K}, \ell_i, v_i, \mathsf{op}_i) \leftrightarrow \mathcal{A}(\mathsf{st}_{\mathcal{A}})$
  else
    $\Sigma.\mathsf{DummyUpdate}_C(\mathsf{K}) \leftrightarrow \mathcal{A}(\mathsf{st}_{\mathcal{A}})$
Output bit $b \leftarrow \mathcal{A}(\mathsf{st}_{\mathcal{A}})$

$\text{EMMIDEAL}^{\mathsf{d,adp}}_{\Sigma,\mathsf{Sim},\mathcal{L},\mathcal{A}}$

$(N, M, D, \mathsf{MM}, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}(1^\lambda)$
$\mathsf{EMM} \leftarrow \mathsf{Sim}(\mathcal{L}_{\mathsf{Stp}}(\mathsf{MM}, N, M, D))$
Send EMM to $\mathcal{A}$
for $1 \le i \le q$ **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 $b \leftarrow \mathcal{A}(\mathsf{st}_{\mathcal{A}})$

Lemma 5 (Cuckoo-Hashing [43]). *For $n \in \mathbb{N}$ and $\varepsilon > 0$, the runtime of allocating n items in cuckoo tables T_1, T_2 of capacity $(1+\varepsilon)n$ via the cuckoo process is $\mathcal{O}(n)$ in expectation, and the probability that allocation fails is at most c for some constant $c < 1$, where the probability is over the random positions choices of each ball.*

For a more detailed description of cuckoo hashing, see [43].

Keyless hash table construction. For our EMM construction, we need a hash table with constant lookups (in the worst case) and a setup that leaks no information about the keys of the input items. Cuckoo-hashing fulfills these requirements, so long as the keys are not stored together with the items within the hash table. However, if the keys are not stored with the items, then the key of the searched item cannot be compared with the key of the two possible items during lookup. We thus simply return both items and the client can later decide which was the right value.

If we were to use cuckoo hashing with a stash [37], then setting up the cuckoo table would have constant overhead at the cost of a logarithmic stash. Unfortunately, since we employ multiple hash tables within our EMM constructions, the resulting client storage due to stashes would become prohibitive. To overcome this, we rehash keys by choosing new seeds for the hash function until allocation is possible. We note that this seemingly minor change introduces some technical challenges in the security proof our constructions later on.[12]

We now formally describe our hash table variant, CT. Let $\varepsilon > 0$ and let $\mathcal{H}_{\mathsf{CT}} : \{0,1\}^* \to [1, m]$ be a hash function.

- $\mathsf{CT.HTSetup}(S)$: On input of n key-value pairs $S = (k_i, x_i)_{i=1}^n$, do the following. Sample $\mathsf{seed} \leftarrow \{0,1\}^{2\lambda}$ for $\mathcal{H}_{\mathsf{CT}}$. Generate tables T_1, T_2 of size $n(1+\varepsilon)$ each, initialized with zero entries. Compute $\alpha_i, \beta_i \leftarrow \mathcal{H}_{\mathsf{CT}}(\mathsf{seed}, k_i)$. Insert x_i at either $T_1[\alpha_i]$ or $T_2[\beta_i]$ according to the cuckoo process (see above). If allocation is not possible, choose new seed $\mathsf{seed} \leftarrow \{0,1\}^{2\lambda}$ and retries. If allocation was not successful after λ tries, output $\perp$. Otherwise, set $T = (T_1, T_2)$ and return (seed, T). Optionally, also output all seeds S_{seeds} that were tried in the above procedure.[13]
- $\mathsf{CT.HTLookup}(k_i, T, \mathsf{seed})$: Given key k_i, tables $T = (T_1, T_2)$ and seed, return $T_1[\alpha_i], T_2[\beta_i]$ for $\alpha_i, \beta_i \leftarrow \mathcal{H}_{\mathsf{CT}}(\mathsf{seed}, k_i)$.

Lemma 6. *We model $\mathcal{H}_{\mathsf{CT}}$ as a random oracle. The probability that setup fails,* i.e., $\perp \leftarrow \mathsf{CT.HTSetup}(S)$, *is negligible. The expected runtime of $\mathsf{CT.HTSetup}(S)$ is $\mathcal{O}(|S|)$ and the runtime of $\mathsf{CT.HTLookup}(k_i, T, \mathsf{seed})$ is $\mathcal{O}(1)$ for all inputs (k_i, T, seed).*

[12] Some applications that use cuckoo hashing cannot afford to rehash, e.g., Tethys is insecure when rehashing is performed [6]. Through careful analysis we are able to show that–in our case–the rehashing procedure does not leak additional information.

[13] This will be convenient for the security proof of Dummy1C.

Proof. This is an immediate consequence of Lemma 5. The probability that allocation fails for some seed is at most $c < 1$ and, thus, the probability that allocation fails for λ seeds is at most $\mathcal{O}(c^\lambda) = \mathsf{negl}(\lambda)$, as the hash functions values are independently random due to different seeds.

In the following, we omit CT in $\mathsf{CT.HTSetup}$ and $\mathsf{CT.HTLookup}$ if the use of CT is clear by context.

Remark. The static volume-hiding EMM scheme of [44] also uses cuckoo hashing with a stash and similarly returns both items at the two hash locations for a given searched item. However, they do so with different motivations. In their case, updates are of no concern; since the items in the tables are encrypted using the client's secret key, the client simply requests both items from the server. Unlike our keyless hash table, they do not hash using a seed during setup.

E.3 Detailed Description

We refer to Section 4 for a brief overview of $\mathsf{Dummy}(\Sigma)$ [40], which our construction is based on, and a brief high-level description of our construction $\mathsf{Dummy1C}$. Below, we describe $\mathsf{Dummy1C}$ in detail.

Let $\delta(\lambda) \leftarrow \log\log(\lambda)$, $m \leftarrow \mathcal{O}(N/(\delta(\lambda)\log(N)))$ be the number of tables and $\mathsf{cap} \leftarrow \mathcal{O}(\delta(\lambda)\log(N))$ be the table capacity. Let $\mathcal{H}_{\mathsf{1C}}$ be a hash function mapping to $\{0,\dots,m-1\}$. This hash function will be used for the underlying $\mathsf{1C}$ mechanism. Similarly, let $\mathcal{H}_{\mathsf{CT}}$ be a hash function mapping to $\{0,\dots,(1+\varepsilon)\mathsf{cap}\}^2$ for some $\varepsilon > 0$. We use $\mathcal{H}_{\mathsf{CT}}$ for keyless cuckoo hash tables CT of capacity U (cf. Appendix E.2). Roughly, the server stores the multi-map values in m CT hash tables $T_0,\dots,T_{m-1}$. The table where some (encrypted) value is stored is derived via $\mathsf{1C}$, which ensures that the tables T_γ need to accommodate at most $\mathsf{cap} = \widetilde{\mathcal{O}}(\log n)$ items and allows for dummy updates. We give a description of $\mathsf{Dummy1C}$ below; a formal description of $\mathsf{Dummy1C}$ is given in Algorithm 5. Note that we assume that Setup does not obtain a multi-map MM as input for simplicity: As the scheme is forward-private, the database can be setup via Update operations online.

During setup, the client first sets up the hash tables $(T_\gamma, \mathsf{seed}_\gamma) \leftarrow \mathsf{HTSetup}(S_\gamma)$ for all $\gamma \in \{0,\dots,m-1\}$, initially with random input $S_\gamma \leftarrow (\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(0), k_i)_{i=1}^{\mathsf{cap}}$ for $k_i \leftarrow \{0,1\}^{2\lambda}$. She then encrypts $S_\gamma^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(S_\gamma)$ and sends the (empty) encrypted multi-map $\mathsf{EMM} = (S_\gamma^{\mathsf{enc}}, T_\gamma, \mathsf{seed}_\gamma)_{\gamma=0}^{m-1}$ to the server. Later, the key-value pairs in S_γ will be replaced with concrete (encrypted) values during updates. Also, the client locally keeps track of the number cnt_i of values associated to label cnt_i in a table $T_{\mathsf{len}}[\mathsf{cnt}_i] = \mathsf{cnt}_i$.

For each update that inserts the i-th value v_i matching label cnt, the client computes its search token $P_i^{\mathsf{cnt}} \leftarrow F_{\mathsf{K}_{\mathsf{PRF}}}(\mathsf{cnt}, i)$, and sets $\gamma \leftarrow \mathcal{H}_{\mathsf{1C}}(P_i^{\mathsf{cnt}})$ to determine the table T_γ where v_i is stored. Then, she downloads T_γ and the input S_γ^{enc} with which T_γ was setup from the server. After decrypting S_γ^{enc}, she inserts v_i with key P_i^ℓ into S_γ, and rebuilds $(T_\gamma, \mathsf{seed}_\gamma) \leftarrow \mathsf{HTSetup}(S_\gamma)$ with the updated input. Then, she sends the updated encrypted input set S_γ^{enc}, table T_γ and seed seed_γ to the server. Note that without rebuilding the table from scratch, we have to change the seed_γ if $\mathsf{HTSetup}$ fails during updates, but never during dummy updates, which would allow to distinguish between both operations. As T_i is small, the rebuild process is efficient. Also, we designed CT such that T_γ and seed_γ leak no information about P_i^ℓ, though showing this in the context of the security game requires care.

To search label cnt, the client reveals the search tokens P_i^ℓ to the server for all $i \in \{1,\dots,T_{\mathsf{len}}[\ell]\}$. These tokens allow the server to recompute the index $\gamma = \mathcal{H}_{\mathsf{1C}}(P_i^\ell)$ of the hash table T_γ that contains the i-th (encrypted) value associated to cnt. The server then retrieves all encrypted values via $\mathsf{HTLookup}(P_i^\ell, T_\gamma, \mathsf{seed}_\gamma)$ and sends the results to the client, who finally decrypts them and filters out false positives.

Remark. When T_γ is accessed during an update, the table T_γ must be rebuilt to ensure forward privacy. For this, we store the content S_γ—with which T_γ was built—on the server (in encrypted format). The client then simply rebuilds T_γ via S_γ (after inserting a fresh value if necessary).

Instead, we can store encryptions of value-key pairs $(\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(v\|k))$ in T_γ instead of encryptions of the values v itself. Then, it is possible to retrieve the input S_γ to $\mathsf{HTSetup}$ from T_γ itself.

Algorithm 5 Dummy1C

$\underline{\mathsf{Dummy1C.KeyGen}(1^\lambda)}$

Sample $\mathsf{K_{PRF}} \leftarrow \{0,1\}^\lambda$ and encryption key $\mathsf{K_{Enc}}$
return $\mathsf{K} = (\mathsf{K_{PRF}}, \mathsf{K_{Enc}})$

$\underline{\mathsf{Dummy1C.Setup}(\mathsf{K}, N)}$

Set $m \leftarrow \lceil N/(\delta(\lambda)\log(N))\rceil$
Set $\mathsf{cap} \leftarrow \mathcal{O}(\delta(\lambda)\log(N))$
for all $\gamma \in \{0,\ldots,m-1\}$ **do**
    Set $S_\gamma \leftarrow (\mathsf{Enc}_{\mathsf{K_{Enc}}}(0), k_i)_{i=1}^{\mathsf{cap}}$ for $k_i \leftarrow \{0,1\}^{2\lambda}$
    Set $(\mathsf{seed}_\gamma, T_\gamma) \leftarrow \mathsf{HTSetup}(S_\gamma)$
    Set $S_\gamma^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K_{Enc}}}(S_\gamma)$
Initialize table $T_{\mathsf{len}}[\ell] = 0$ for all keywords ℓ
return $\mathsf{EMM} = (S_\gamma^{\mathsf{enc}}, T_\gamma, \mathsf{seed}_\gamma)_{\gamma=0}^{m-1}$

$\underline{\mathsf{Dummy1C.Update}(\mathsf{K}, (\ell, v), \mathsf{op}; \mathsf{EMM})}$

Client:

Set $\mathsf{cnt} \leftarrow T_{\mathsf{len}}[\ell]$
if $\mathsf{cnt} = \perp$ **then** set $\mathsf{cnt} \leftarrow -1$
Set $P_{\mathsf{cnt}+1}^\ell \leftarrow F_{\mathsf{K_{PRF}}}(\ell, \mathsf{cnt}+1)$
Set $\gamma \leftarrow \mathcal{H}_{\mathsf{1C}}(P_{\mathsf{cnt}+1}^\ell)$
send γ

Server:

Send $(T_\gamma, S_\gamma^{\mathsf{enc}})$

Client:

Decrypt $S_\gamma \leftarrow \mathsf{Dec}_{\mathsf{K_{Enc}}}(S_\gamma^{\mathsf{enc}})$
Reencrypt all values in S_γ
Replace an encrypted zero in S_γ with $(\mathsf{Enc}_{\mathsf{K_{Enc}}}(v), P_{\mathsf{cnt}+1}^\ell)$
Run $(T_\gamma, \mathsf{seed}_\gamma) \leftarrow \mathsf{HTSetup}(S_\gamma)$
Reencrypt $S_\gamma^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K_{Enc}}}(S_\gamma)$
Increment $T_{\mathsf{len}}[\ell]$
send $(T_\gamma, S_\gamma^{\mathsf{enc}}, \mathsf{seed}_\gamma)$

Server:

Store updated $T_\gamma, S_\gamma^{\mathsf{enc}}$ and seed_γ

$\underline{\mathsf{Dummy1C.Search}(\mathsf{K}, \ell; \mathsf{EMM})}$

Client:

Set $\mathsf{cnt} \leftarrow T_{\mathsf{len}}[\ell]$
if $\mathsf{cnt} = \perp$ **then return** $\emptyset$
Set $(P_i^\ell) \leftarrow F_{\mathsf{K_{PRF}}}(\ell, i)$ for $i \in \{1,\ldots,\mathsf{cnt}\}$
send $(P_i^\ell)_{i=1}^{\mathsf{cnt}}$

Server:

Initialize empty set R
for all $i \in \{1,\ldots,\mathsf{cnt}\}$ **do**
    Set $\gamma \leftarrow \mathcal{H}_{\mathsf{1C}}(P_i^\ell)$
    Add $\mathsf{HTLookup}(P_i^\ell, T_\gamma, \mathsf{seed}_\gamma)$ to R
send R

Client:

Decrypt R to retrieve $\mathsf{MM}(\ell)$

$\underline{\mathsf{Dummy1C.DummyUpdate}(\mathsf{K}; \mathsf{EMM})}$

Client:

Set $\gamma \leftarrow \{0,\ldots,m-1\}$
send γ

Server:

Send $(T_\gamma, S_\gamma^{\mathsf{enc}})$

Client:

Decrypt $S_\gamma \leftarrow \mathsf{Dec}_{\mathsf{K_{Enc}}}(S_\gamma^{\mathsf{enc}})$
Reencrypt all values in S_γ
Run $(T_\gamma, \mathsf{seed}_\gamma) \leftarrow \mathsf{HTSetup}(S_\gamma)$
Reencrypt $S_\gamma^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K_{Enc}}}(S_\gamma)$
send $(T_\gamma, S_\gamma^{\mathsf{enc}}, \mathsf{seed}_\gamma)$

Server:

Store updated $T_\gamma, S_\gamma^{\mathsf{enc}}$ and seed_γ

Security. Theorem 1 shows that Dummy1C is secure with respect to definition 9 with standard search and setup leakage and optimal update leakage.

Theorem 1. *Let $N \geq \lambda$, F be a secure* PRF *mapping to $\{0,1\}^{2\lambda}$ and* Enc *be a secure* IND-CPA *encryption. We model the hash functions $\mathcal{H}_{1C}$ and $\mathcal{H}_{\mathsf{CT}}$ as random oracles. The scheme* Dummy1C *is correct and supports $\mathsf{poly}(\lambda)(\lambda)$-many dummy updates with leakage $\mathcal{L}^{\mathsf{dummy}} = (\mathcal{L}^{\mathsf{dummy}}_{\mathsf{Stp}}, \mathcal{L}^{\mathsf{dummy}}_{\mathsf{Srch}}, \mathcal{L}^{\mathsf{dummy}}_{\mathsf{Updt}})$, where $\mathcal{L}^{\mathsf{dummy}}_{\mathsf{Stp}}(N, D, M) = N, \mathcal{L}^{\mathsf{dummy}}_{\mathsf{Srch}}(\mathsf{cnt}) = (\mathsf{qp}, \mathsf{cnt})$ and $\mathcal{L}^{\mathsf{dummy}}_{\mathsf{Updt}}(\mathsf{op}, \mathsf{cnt}, v) = \perp$.*

While verifying correctness is straight-forward, the proof for semantical dummy security is rather technical. We first sketch security and give a detailed proof below.

First, note that Setup only relies on its input N, and thus trivially leaks no other information than N (or in other words, can be simulated only given N). But to later show that updates and dummy updates are indistinguishable, we also need to make sure that the output of HTSetup, namely T_γ and seed_γ, reveal no information about the keys k_i in the input S_γ.

We first give some intuition for why this is important. Assume that on the contrary, the adversary could learn some k_i in S_γ based on the output of HTSetup. Then, when querying $\mathcal{H}_{\mathsf{CT}}$ on these k_i, *i.e.*, $(\alpha_i, \beta_i) \leftarrow \mathcal{H}_{\mathsf{CT}}(\mathsf{seed}_\gamma, k_i)$, the positions $\{\alpha_i, \beta_i\}_i$ always allow for a valid cuckoo hashing assignment. During updates, assume that the keys k_i were replaced with other keys. Then, when querying $\mathcal{H}_{\mathsf{CT}}$ on keys k_i with the a new seed seed'_γ, the new positions $\{(\alpha'_i, \beta'_i) \mid (\alpha'_i, \beta'_i) \leftarrow \mathcal{H}_{\mathsf{CT}}(\mathsf{seed}'_\gamma, k_i)\}_i$ are random, as the k_i are not considered anymore during HTSetup (for deciding whether a seed is sufficient). Thus, they might not allow for a cuckoo assignment anymore with noticeable probability [14]. The above yields a concrete distinguishing attack between a scenario with only dummy updates, and one with only real updates. In summary, when only dummy updates are performed, the outputs of $\mathcal{H}_{\mathsf{CT}}$ always remain valid with regards to cuckoo hashing on input k_i, but is invalid with noticeable probability otherwise.

Thus, we need to show that the outputs $(T_\gamma, \mathsf{seed}_\gamma)$ reveal no information about the keys in S_γ. Intuitively, this follows as HTSetup only uses these keys to query $\mathcal{H}_{\mathsf{CT}}$, and the output of $\mathcal{H}_{\mathsf{CT}}$ is uniformly random in our model. On a technical level, we show this by simulating the setup with different keys, and reprogramming $\mathcal{H}_{\mathsf{CT}}$ later, such that the distribution of the update and $\mathcal{H}_{\mathsf{CT}}$ output is equally distributed to the real game. This technique also applies after some keys k_i were replaced with some tokens P_i^ℓ during updates. The important consequence is that the simulator can "lazily" reprogram $\mathcal{H}_{\mathsf{CT}}$ for inputs P_i^ℓ until the token was revealed during search, as the adversary is unlikely to query P_i^ℓ beforehand. Also, the simulator can program $\mathcal{H}_{\mathsf{CT}}$ for non-search token inputs k_i arbitrarily, as these are never sent to the adversary, and only queried with negligible probability.

The above allows us to argue that updates leak no information as follows. Observe that the first client output γ looks uniformly random to the server, as $\mathcal{H}_{1\mathsf{C}}$ is modeled as a random oracle. Further, the output of HTSetup is independent on the used keys (as explained above), and the items stored in the tables T_γ are encrypted. As also S^{enc}_γ is encrypted, the outputs of Update can be simulated with zero leakage (as long as both oracles $\mathcal{H}_{\mathsf{CT}}$ and $\mathcal{H}_{1\mathsf{C}}$ are reprogrammed according to the query leakage during searches). We can proceed exactly the same to simulate dummy updates, and thus both are indistinguishable.

For searches, note that the output of F is pseudorandom. Thus, as in most EMM security proofs, the simulator can simply generate random tokens for not yet queried label-value pairs, and otherwise reuse the tokens from previous searches. This simulation requires the query pattern qp and the number cnt of associated values. Also, for all new tokens, the random oracles $\mathcal{H}_{\mathsf{CT}}$ and $\mathcal{H}_{1\mathsf{C}}$ are reprogrammed to match the expected distribution (which is possible provided qp).

Proof (Proof of Theorem 1.). Recall the setup leakage $\mathcal{L}^{\mathsf{dummy}}_{\mathsf{Stp}}(N, D, M) = N$, search leakage $\mathcal{L}^{\mathsf{dummy}}_{\mathsf{Srch}}(\ell) = (\mathsf{qp}, \mathsf{cnt})$ and update leakage $\mathcal{L}^{\mathsf{dummy}}_{\mathsf{Updt}}(\mathsf{op}, \ell, v) = \perp$ of the leakage function $\mathcal{L}^{\mathsf{dummy}}$. Let $Q_{\mathcal{H}_{\mathsf{CT}}}$ be the number of $\mathcal{H}_{\mathsf{CT}}$-queries, $Q_{\mathcal{H}_{1\mathsf{C}}}$ be the number of $\mathcal{H}_{1\mathsf{C}}$-queries, Q_{Updt} be the number of update queries (including dummy updates) and Q_{Srch} be the number of search queries, performed by the adversary $\mathcal{A}$. The simulator is given in Algorithm 11. We show the claim by providing a sequence of indistinguishable hybrids from the real game to the ideal game.

[14] This probability is noticeable by the pigeonhole principle. For example if three keys k_1, k_2, k_3 are mapped to the same two bin choices, there is no valid cuckoo assignment. This happens with probability $(\mathsf{cap} + \varepsilon)^{-6}$.

Hybrid 0. Hybrid 0 is identical to the real game. Note that $\mathcal{H}_{1\mathsf{C}}$ and $\mathcal{H}_{\mathsf{CT}}$ are modeled as programmable random oracles.

Hybrid 1. Hybrid 1 is identical to Hybrid 0, except each search token P_i^ℓ is drawn uniformly at random from $\{0,1\}^{2\lambda}$ instead of being the output of F. Under the PRF security of F, hybrid 1 and hybrid 0 are indistinguishable.

Hybrid 2. Hybrid 2 is identical to Hybrid 1, except instead of generating a random $\gamma \in \{0,\ldots,m-1\}$ in a dummy update, first a $P \leftarrow \{0,1\}^{2\lambda}$ is drawn at random, and then $\gamma \leftarrow \mathcal{H}_{1\mathsf{C}}(P)$ is output instead. If P is never queried in random oracle $\mathcal{H}_{1\mathsf{C}}$, then the outputs of Hybrid 2 and Hybrid 1 are identically distributed. Because there are $2^{2\lambda}$ possible values of P, but at most $Q_{\mathcal{H}_{1\mathsf{C}}} + Q_{\mathsf{Updt}}$ evaluations of $\mathcal{H}_{1\mathsf{C}}$, the probability that P was already queried is at most $\frac{Q_{\mathcal{H}_{1\mathsf{C}}}+Q_{\mathsf{Updt}}}{2^{2\lambda}}$. Thus, an adversary can distinguish between Hybrid 1 and Hybrid 2 with advantage at most $\frac{Q_{\mathcal{H}_{1\mathsf{C}}}+Q_{\mathsf{Updt}}}{2^{2\lambda}}$.

Hybrid 3. Hybrid 3 is identical to Hybrid 2, except the tuples S_γ are stored locally for all $\gamma \in \{0,\ldots,m-1\}$. During updates, the retrieved encrypted S_γ^{enc} is ignored, and the pair $(\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(v), P^\ell_{\mathsf{cnt}+1})$ is inserted into the local S_γ instead which is used as input for $\mathsf{HTSetup}$. The output S_γ^{enc} is an encryption of the local S_γ. Hybrid 2 and Hybrid 3 are distributed identically, as the adversary is honest but curious.

Hybrid 4. Hybrid 4 is identical to Hybrid 3, except all encryptions are replaced with fresh encryptions of 0 (of the appropriate length). Note that $\mathsf{HTSetup}$ behaves independently of the concrete items in S_γ, it just stores them in tables at positions that depend only on the keys. Under $\mathsf{IND\text{-}CPA}$ security of Enc, Hybrid 4 and Hybrid 3 are indistinguishable.

Hybrid 5. Hybrid 5 is given in Algorithm 6 and is identical to Hybrid 4, except during the i-th (dummy or real) update, we store the randomly chosen P_i for later. During search, we identify each "real" label with some arbitrary, fresh label ℓ whenever it was first searched for readability, which is the case if sp is 0 for all indices except for the current search. Then, we recompute which cnt tokens $(P_j^\ell)_{j=1}^{\mathsf{cnt}}$ of the stored $\{P_i\}_i$ match the searched label via the update pattern up, where cnt is the number of matching values, and output these tokens. Hybrid 4 and Hybrid 5 are identically distributed.

Next, we show that the output during updates leak no information about the used keys during $\mathsf{HTSetup}$. This allows us to later argue that we can delay the programming of $\mathcal{H}_{\mathsf{CT}}$ until search.

Hybrid 6. Hybrid 6 is given in Algorithm 7 and is identical to Hybrid 5, except that a flag $\mathsf{flg}_{\mathsf{bad}} \leftarrow 0$ is initialized during setup. If during the experiment, the adversary succeeds in setting $\mathsf{flg}_{\mathsf{bad}} = 1$, then output FAIL. Further, during the i-th (real or dummy) update, if some seed seed that was used during $\mathsf{HTSetup}$ was already queried in $\mathcal{H}_{\mathsf{CT}}$, set $\mathsf{flg}_{\mathsf{bad}} \leftarrow 1$. The seeds are chosen uniformly at random during $\mathsf{HTSetup}$, and per update query, at most λ such seeds are chosen. Thus, $\mathsf{flg}_{\mathsf{bad}}$ will be set with probability at most $\frac{(Q_{\mathcal{H}_{\mathsf{CT}}}+\lambda\cdot Q_{\mathsf{Updt}})^2}{2^{2\lambda}}$. As Hybrid 5 and Hybrid 6 are identically distributed, except if FAIL was output, the above probability upper bounds the advantage of $\mathcal{A}$ to distinguish between Hybrid 5 and Hybrid 6.

Hybrid 7. Hybrid 7 is given in Algorithm 8 and is identical to Hybrid 6, except we modify the way the CT hash tables are setup (during setup and updates). That is, we always generate T_γ via $(\mathsf{seed}_\gamma, T_\gamma) \leftarrow \mathsf{HTSetup}(\mathsf{Remap}(S_\gamma))$ instead of via $\mathsf{HTSetup}(S_\gamma)$. Here, $\mathsf{Remap}(S_\gamma)$ denotes the tuple where the keys in S_γ are replaced with the keys $1,\ldots,n$. Formally, $\mathsf{Remap}((x_i,\tau_i)_{i=1}^{\mathsf{cap}})$ returns the tuple $(x_i,i)_{i=1}^{\mathsf{cap}}$. After the call to $\mathsf{HTSetup}$, we reprogram $\mathcal{H}_{\mathsf{CT}}$ such that its output is distributed as if $\mathsf{HTSetup}$ was run with S_γ. In more detail, we set $\mathcal{H}_{\mathsf{CT}}[s,\tau_j] = \mathcal{H}_{\mathsf{CT}}[s,j]$ and $\mathcal{H}_{\mathsf{CT}}[s,j] \leftarrow \bot$, for all seeds s tried during $\mathsf{HTSetup}$ and all keys τ_j in S_γ. Note that because there was never a query of the form $\mathcal{H}_{\mathsf{CT}}(s,\cdot)$ before (otherwise $\mathsf{flg}_{\mathsf{bad}}$ was set), the adversary is unaware of the reprogramming. Note that $\mathcal{H}_{\mathsf{CT}}$ is still a valid random oracle, as $\mathcal{H}_{\mathsf{CT}}[s,j]$ is chosen uniformly and independently at random for all j. As $\mathsf{HTSetup}$ only uses the keys in S_γ in order to query $\mathcal{H}_{\mathsf{CT}}$, and because we reprogrammed $\mathcal{H}_{\mathsf{CT}}$ accordingly, the distribution of the output of $(\mathsf{seed}_\gamma, T_\gamma)$ is the same in Hybrid 6 and Hybrid 7.

The important consequence of Hybrid 7 is that the output $(T_\gamma, \mathsf{seed}_\gamma)$ of $\mathsf{HTSetup}$ reveals no information about the supplied keys. The only part of the experiment that depends on the keys supplied to $\mathsf{HTSetup}$ is the state of $\mathcal{H}_{\mathsf{CT}}$.

Hybrid 8. Hybrid 8 is given in Algorithm 9 and is identical to Hybrid 7, except $\mathsf{flg}_{\mathsf{bad}}$ is set if any of the keys k_i in S_γ chosen during setup or some P_i chosen during (real or dummy) updates is queried in $\mathcal{H}_{\mathsf{CT}}$, for the latter conditioned on not yet being revealed during search. As the adversary has information-theoretically no information about these "bad" values (at the time $\mathcal{H}_{\mathsf{CT}}$ is queried) and as they are chosen uniformly and independently at random, the probability that the value k supplied to a query $\mathcal{H}_{\mathsf{CT}}(s,k)$, for any seed s, is equal to a bad value is at most $\frac{(\mathsf{cap}\cdot m+Q_{\mathsf{Updt}})\cdot Q_{\mathcal{H}_{\mathsf{CT}}}}{2^{2\lambda}}$. As Hybrid 7 and Hybrid 8 are identically distributed, except if FAIL was output after some bad value was queried, the above probability upper bounds the advantage of $\mathcal{A}$ to distinguish between Hybrid 7 and Hybrid 8.

Hybrid 9. Hybrid 9 is given in Algorithm 10 and is identical to Hybrid 8, except the insertion of a new pair $(\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(0), P_i)$ into S_γ during the i-th update is delayed until the search query that reveals P_i to the adversary. Also, the reprogramming of $\mathcal{H}_{\mathsf{CT}}$ for inputs P_i is done during the revealing search query retroactively. Intuitively, the adversary cannot distinguish Hybrid 9 and Hybrid 8, as $\mathcal{H}_{\mathsf{CT}}$ is never queried on these inputs before the reprogramming is done, and the reprogramming is the only operation that depends on the concrete keys in S_γ.

In more detail, we first store in a table T_{len} the number cnt of values matching label ℓ that were revealed during a search. That is, after each search, we set $T_{\mathsf{len}}[\ell] \leftarrow \mathsf{cnt}$, where ℓ is the currently searched label (computed from the search pattern) and cnt is the number of updates on ℓ revealed by the search. Thus, $T_{\mathsf{len}}[\ell]$ denotes the length of the prefix of tokens for label ℓ that has already been revealed to the adversary. By the semantics of the search leakage, these prefixes are monotonically increasing. Hence, for every subsequent search of ℓ with leakage cnt, we have $T_{\mathsf{len}}[\ell] \leq \mathsf{cnt}$. Consequently, if we set $\mathsf{cnt}' = T_{\mathsf{len}}[\ell]$, the tokens that have not yet been revealed are exactly those with indices i satisfying $\mathsf{cnt}' < i \leq \mathsf{cnt}$. This allows us to keep track of which P_i have already been revealed to the adversary.

Further, we (implicitly) keep track of all seeds that were used during the i-th (dummy or real) update in order to reprogram $\mathcal{H}_{\mathsf{CT}}$ later for the revealed P_i during search. Next, we delay the insertion of $(\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(0), P_i)$ into S_γ in the i-th update until the first search, where $\mathsf{up}[i] = 1$, *i.e.*the first search that reveals P_i. Note that this does not impact the output of updates, as $(T_\gamma, S_\gamma^{\mathsf{enc}})$ is independent of the keys in S_γ and all items in S_γ are fresh encryptions of zero. However, because we reprogram the random oracle $\mathcal{H}_{\mathsf{CT}}$ based on the keys in S_γ after $\mathsf{HTSetup}$ during updates in Hybrid 8, the games differ in the outputs of the random oracle.

Notably, all outputs of $\mathcal{H}_{\mathsf{CT}}(\mathsf{seed}, P_i)$ are inconsistent for now. The "correct" outputs are instead stored in $\mathcal{H}_{\mathsf{CT}}[\mathsf{seed}, k_x]$ for some key $k_x \in K_\gamma$ (that should have been replaced during the i-th update). Importantly, these P_i and all such k_x are never queried until revealed during search (otherwise $\mathsf{flg}_{\mathsf{bad}} = 1$). Thus, we simply program $\mathcal{H}_{\mathsf{CT}}(\mathsf{seed}, P_i) \leftarrow \mathcal{H}_{\mathsf{CT}}(\mathsf{seed}, k_x)$ during the first search, where $\mathsf{up}[i] = 1$, for all required seeds seed (*i.e.*seeds that were either used during the i-th update or in subsequent updates that rebuilt the same table). Note that via T_{len}, we can identify this search query.

Now, the outputs of $\mathcal{H}_{\mathsf{CT}}(\mathsf{seed}, P_i)$ are distributed equally in Hybrid 8 and Hybrid 9, because these values are never queried before being correctly programmed (otherwise $\mathsf{flg}_{\mathsf{bad}} = 1$). Note that $\mathcal{H}_{\mathsf{CT}}$ is not a valid oracle anymore as its outputs are not randomly distributed for $\mathcal{H}_{\mathsf{CT}}[\mathsf{seed}, k_x]$ (because $\mathcal{H}_{\mathsf{CT}}[\mathsf{seed}, P_i] = \mathcal{H}_{\mathsf{CT}}[\mathsf{seed}, k_x]$). Fortunately, these values are never queried. Thus, Hybrid 8 and Hybrid 9 are identically distributed.

Hybrid 10. Hybrid 10 is identical to the ideal game with the simulator given in Algorithm 11. Note that real updates and dummy updates are identical in Hybrid 9, as the update specific operations are delayed until the search queries. Hybrid 9 and Hybrid 10 are identically distributed. This concludes the proof.

Efficiency. $\mathsf{Dummy1C}$ achieves a constant efficiency overhead for storage, as well as constant communication and computational complexity for searches. For updates, it incurs a $\widetilde{\mathcal{O}}(\log N)$ overhead in communication and computational complexity, due to the use of the one-choice process. In Section 5, we will use $\mathsf{Dummy1C}$ as a building block that handles full pages of values. In

[15] The required values seed are exactly all seeds that were used to reconstruct T_γ during some (dummy or real) update query issued after the definition of P_j. See Appendix E.3 for more details.

Algorithm 6 The fifth hybrid. All modifications until here are standard in EMM literature. The modifications are marked in blue boxes.

Dummy1C.Setup(K, N)

Set $m \leftarrow \lceil N/(\delta(\lambda)\log(N))\rceil$
Set $\mathsf{cap} \leftarrow \mathcal{O}(\delta(\lambda)\log(N))$
for all $\gamma \in \{0,\ldots,m-1\}$ **do**
  Set $S_\gamma \leftarrow (\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(0), k_i)_{i=1}^{\mathsf{cap}}$ for $k_i \leftarrow \{0,1\}^{2\lambda}$ ▷ Hybrid 3
  $(\mathsf{seed}_\gamma, T_\gamma) \leftarrow \mathsf{HTSetup}(S_\gamma)$
  Set $S_\gamma^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}((0,0)_{i=1}^{\mathsf{cap}})$ ▷ Hybrid 4
  Store S_γ locally ▷ Hybrid 3
Initialize counter $\mathsf{cnt}_{\mathsf{updt}} \leftarrow 0$
Initialize empty tables $\mathcal{H}_{1\mathsf{C}}, \mathcal{H}_{\mathsf{CT}}$
return output $\mathsf{EDB} = (S_\gamma^{\mathsf{enc}}, T_\gamma, \mathsf{seed}_\gamma)_{\gamma=0}^{m-1}$

Dummy1C.Search(K, qp = (sp, up), cnt; EDB)

Client:
if $\mathsf{cnt} = 0$ **then return** $\emptyset$
Recompute value ℓ of the searched label via sp ▷ Hybrid 5
Let $\mathcal{J} = (j : \mathsf{up}[j] = 1)$ be the indices of updates on the searched label ▷ Hybrid 5
for all $j \in \mathcal{J}$ in ascending order **do**
  Let i be the index of j in $\mathcal{J}$ ▷ Hybrid 5
  Set $P_i^\ell \leftarrow P_j$ ▷ Hybrid 5
send $(P_i^\ell)_{i=1}^{\mathsf{cnt}}$
Server:
Initialize empty set R
for all $i \in \{1,\ldots,\mathsf{cnt}\}$ **do**
  Set $\gamma \leftarrow \mathcal{H}_{1\mathsf{C}}(P_i^\ell)$
  Add $\mathsf{HTLookup}(P_i^\ell, T_\gamma, \mathsf{seed}_\gamma)$ to R
send R

Dummy1C.Update(K, op; EDB)

Client:
Set $i \leftarrow \mathsf{cnt}_{\mathsf{updt}} + 1$ and increment $\mathsf{cnt}_{\mathsf{updt}}$
Set $P_i \leftarrow \{0,1\}^{2\lambda}$ ▷ Hybrid 1
Set $\gamma \leftarrow \mathcal{H}_{1\mathsf{C}}(P_i)$ ▷ Hybrid 2
send γ
Server:
Send $(T_\gamma, S_\gamma^{\mathsf{enc}})$
Client:
Reencrypt all values in local S_γ ▷ Hybrid 3
Replace an item in S_γ inserted during setup with $(\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(0), P_i)$ ▷ Hybrid 3
Run $(T_\gamma, \mathsf{seed}_\gamma) \leftarrow \mathsf{HTSetup}(S_\gamma)$
Encrypt $S_\gamma^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}((0,0)_{i=1}^{\mathsf{cap}})$ ▷ Hybrid 4
send $(T_\gamma, S_\gamma^{\mathsf{enc}}, \mathsf{seed}_\gamma)$
Server:
Store updated $T_\gamma, S_\gamma^{\mathsf{enc}}$ and seed_γ

Dummy1C.DummyUpdate(K; EDB)

Client:
Set $i \leftarrow \mathsf{cnt}_{\mathsf{updt}} + 1$ and increment $\mathsf{cnt}_{\mathsf{updt}}$
Set $P_i \leftarrow \{0,1\}^{2\lambda}$ ▷ Hybrid 1
Set $\gamma \leftarrow \mathcal{H}_{1\mathsf{C}}(P_i)$ ▷ Hybrid 2
send γ
Server:
Send $(T_\gamma, S_\gamma^{\mathsf{enc}})$
Client:
Reencrypt all values in local S_γ ▷ Hybrid 3
Run $(T_\gamma, \mathsf{seed}_\gamma) \leftarrow \mathsf{HTSetup}(S_\gamma)$
Encrypt $S_\gamma^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}((0,0)_{i=1}^{\mathsf{cap}})$ ▷ Hybrid 4
send $(T_\gamma, S_\gamma^{\mathsf{enc}}, \mathsf{seed}_\gamma)$
Server:
Store updated $T_\gamma, S_\gamma^{\mathsf{enc}}$ and seed_γ

$\mathcal{H}_{1\mathsf{C}}(x)$

if $\mathcal{H}_{1\mathsf{C}}[x] = \perp$ **then**
  $\mathcal{H}_{1\mathsf{C}}[x] \leftarrow \{0,\ldots,m-1\}$
return $\mathcal{H}_{1\mathsf{C}}[x]$

$\mathcal{H}_{\mathsf{CT}}(x)$

if $\mathcal{H}_{\mathsf{CT}}[x] = \perp$ **then**
  $\mathcal{H}_{\mathsf{CT}}[x] \leftarrow \{0,\ldots,(1+\varepsilon)\mathsf{cap}\}^2$
return $\mathcal{H}_{\mathsf{CT}}[x]$

Algorithm 7 The sixth hybrid. Set $\mathsf{flg}_{\mathsf{bad}} \leftarrow 1$ if chosen seeds during update were already queried in $\mathcal{H}_{\mathsf{CT}}$. Modifications are marked in blue boxes.

$\mathsf{Dummy1C.Setup}(\mathsf{K}, N)$

Set $m \leftarrow \lceil N/(\delta(\lambda)\log(N))\rceil$
Set $\mathsf{cap} \leftarrow \mathcal{O}(\delta(\lambda)\log(N))$
for all $\gamma \in \{0,\ldots,m-1\}$ **do**
    Set $S_\gamma \leftarrow (\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(0), k_i)_{i=1}^{\mathsf{cap}}$ for $k_i \leftarrow \{0,1\}^{2\lambda}$
    $(\mathsf{seed}_\gamma, T_\gamma) \leftarrow \mathsf{HTSetup}(S_\gamma)$
    Set $S_\gamma^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}((0,0)_{i=1}^{\mathsf{cap}})$
    Store S_γ locally
Initialize counter $\mathsf{cnt}_{\mathsf{updt}} \leftarrow 0$
Initialize empty tables $\mathcal{H}_{\mathsf{1C}}, \mathcal{H}_{\mathsf{CT}}$
Set $\mathsf{flg}_{\mathsf{bad}} \leftarrow 0$
return output $\mathsf{EDB} = (S_\gamma^{\mathsf{enc}}, T_\gamma, \mathsf{seed}_\gamma)_{\gamma=0}^{m-1}$

$\mathsf{Dummy1C.Search}(\mathsf{K}, \mathsf{qp} = (\mathsf{sp}, \mathsf{up}), \mathsf{cnt}; \mathsf{EDB})$

Client:

if $\mathsf{cnt} = 0$ **then return** $\emptyset$
Recompute value ℓ of the searched label via sp
Let $\mathcal{J} = (j : \mathsf{up}[j] = 1)$ be the indices of updates on the searched label
for all $j \in \mathcal{J}$ in ascending order **do**
    Set $P_i^\ell \leftarrow P_j$, where i is the index of j in $\mathcal{J}$
send $(P_i^\ell)_{i=1}^{\mathsf{cnt}}$

Server:

Initialize empty set R
for all $i \in \{1,\ldots,\mathsf{cnt}\}$ **do**
    Set $\gamma \leftarrow \mathcal{H}_{\mathsf{1C}}(P_i^\ell)$
    Add $\mathsf{HTLookup}(P_i^\ell, T_\gamma, \mathsf{seed}_\gamma)$ to R
send R

$\mathsf{Dummy1C.Update}(\mathsf{K}, \mathsf{op}; \mathsf{EDB})$

Client:

Set $i \leftarrow \mathsf{cnt}_{\mathsf{updt}} + 1$ and increment $\mathsf{cnt}_{\mathsf{updt}}$
Set $P_i \leftarrow \{0,1\}^{2\lambda}$
Set $\gamma \leftarrow \mathcal{H}_{\mathsf{1C}}(P_i)$
send γ

Server:

Send $(T_\gamma, S_\gamma^{\mathsf{enc}})$

Client:

Reencrypt all values in S_γ (stored by simulator)
Replace an item in S_γ inserted during setup with $(\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(0), P_i)$
Save state of $\mathcal{H}'_{\mathsf{CT}} \leftarrow \mathcal{H}_{\mathsf{CT}}$
Run $(S_{\mathsf{seeds}}, T_\gamma, \mathsf{seed}_\gamma) \leftarrow \mathsf{HTSetup}(S_\gamma)$
if $\exists s \in S_{\mathsf{seeds}} : \mathcal{H}'_{\mathsf{CT}}[s,\cdot] \neq \perp$ **then**
    $\mathsf{flg}_{\mathsf{bad}} \leftarrow 1$
Encrypt $S_\gamma^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}((0,0)_{i=1}^{\mathsf{cap}})$
send $(T_\gamma, S_\gamma^{\mathsf{enc}}, \mathsf{seed}_\gamma)$

Server:

Store updated $T_\gamma, S_\gamma^{\mathsf{enc}}$ and seed_γ

$\mathsf{Dummy1C.DummyUpdate}(\mathsf{K}; \mathsf{EDB})$

Client:

Set $i \leftarrow \mathsf{cnt}_{\mathsf{updt}} + 1$ and increment $\mathsf{cnt}_{\mathsf{updt}}$
Set $P_i \leftarrow \{0,1\}^{2\lambda}$
Set $\gamma \leftarrow \mathcal{H}_{\mathsf{1C}}(P_i)$
send γ

Server:

Send $(T_\gamma, S_\gamma^{\mathsf{enc}})$

Client:

Reencrypt all values in S_γ (stored by simulator)
Save state of $\mathcal{H}'_{\mathsf{CT}} \leftarrow \mathcal{H}_{\mathsf{CT}}$
Run $(S_{\mathsf{seeds}}, T_\gamma, \mathsf{seed}_\gamma) \leftarrow \mathsf{HTSetup}(S_\gamma)$
if $\exists s \in S_{\mathsf{seeds}} : \mathcal{H}'_{\mathsf{CT}}[s,\cdot] \neq \perp$ **then**
    $\mathsf{flg}_{\mathsf{bad}} \leftarrow 1$
Encrypt $S_\gamma^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}((0,0)_{i=1}^{\mathsf{cap}})$
send $(T_\gamma, S_\gamma^{\mathsf{enc}}, \mathsf{seed}_\gamma)$

Server:

Store updated $T_\gamma, S_\gamma^{\mathsf{enc}}$ and seed_γ

$\mathcal{H}_{\mathsf{1C}}(x)$

if $\mathcal{H}_{\mathsf{1C}}[x] = \perp$ **then**
    $\mathcal{H}_{\mathsf{1C}}[x] \leftarrow \{0,\ldots,m-1\}$
return $\mathcal{H}_{\mathsf{1C}}[x]$

$\mathcal{H}_{\mathsf{CT}}(x)$

if $\mathcal{H}_{\mathsf{CT}}[x] = \perp$ **then**
    $\mathcal{H}_{\mathsf{CT}}[x] \leftarrow \{0,\ldots,(1+\varepsilon)\mathsf{cap}\}^2$
return $\mathcal{H}_{\mathsf{CT}}[x]$

Algorithm 8 The seventh hybrid. Use different keys during $\mathsf{HTSetup}$ and reprogram $\mathcal{H}_{\mathsf{CT}}$ afterwards. $\mathsf{Remap}(S_\gamma)$ denotes the function that maps tuples $S_\gamma = (x_i, \tau_i)_{i=1}^{\mathsf{cap}}$ to tuple $(x_i, i)_{i=1}^{\mathsf{cap}}$, *i.e.*, it replaces the keys with $[1, \mathsf{cap}]$.

$\mathsf{Dummy1C.Setup}(\mathsf{K}, N)$

Set $m \leftarrow \lceil N/(\delta(\lambda)\log(N))\rceil$
Set $\mathsf{cap} \leftarrow \mathcal{O}(\delta(\lambda)\log(N))$
for all $\gamma \in \{0, \ldots, m-1\}$ **do**
  Set $S_\gamma \leftarrow (\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(0), k_i)_{i=1}^{\mathsf{cap}}$ for $k_i \leftarrow \{0,1\}^{2\lambda}$
  Run $(S_{\mathsf{seeds}}, T_\gamma, \mathsf{seed}_\gamma) \leftarrow \mathsf{HTSetup}(\mathsf{Remap}(S_\gamma))$
  for all $s \in S_{\mathsf{seeds}}$ and $(x_j, \tau_j) \in S_\gamma$ **do**
    $\mathcal{H}_{\mathsf{CT}}[s, \tau_j] = \mathcal{H}_{\mathsf{CT}}[s, j]$
    $\mathcal{H}_{\mathsf{CT}}[s, j] = \perp$
  Set $S_\gamma^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}((0,0)_{i=1}^{\mathsf{cap}})$
  Store S_γ locally
Initialize counter $\mathsf{cnt}_{\mathsf{updt}} \leftarrow 0$
Initialize empty tables $\mathcal{H}_{\mathsf{1C}}, \mathcal{H}_{\mathsf{CT}}$
Set $\mathsf{flg}_{\mathsf{bad}} \leftarrow 0$
return output $\mathsf{EDB} = (S_\gamma^{\mathsf{enc}}, T_\gamma, \mathsf{seed}_\gamma)_{\gamma=0}^{m-1}$

$\mathsf{Dummy1C.Update}(\mathsf{K}, \mathsf{op}; \mathsf{EDB})$

Client:
Set $i \leftarrow \mathsf{cnt}_{\mathsf{updt}} + 1$ and increment $\mathsf{cnt}_{\mathsf{updt}}$
Set $P_i \leftarrow \{0,1\}^{2\lambda}$
Set $\gamma \leftarrow \mathcal{H}_{\mathsf{1C}}(P_i)$
send γ
Server:
Send $(T_\gamma, S_\gamma^{\mathsf{enc}})$
Client:
Reencrypt all values in S_γ (stored by simulator)
Replace an item in S_γ inserted during setup with $(\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(0), P_i)$
Save state of $\mathcal{H}'_{\mathsf{CT}} \leftarrow \mathcal{H}_{\mathsf{CT}}$
Run $(S_{\mathsf{seeds}}, T_\gamma, \mathsf{seed}_\gamma) \leftarrow \mathsf{HTSetup}(\mathsf{Remap}(S_\gamma))$
for all $s \in S_{\mathsf{seeds}}$ and $(x_j, \tau_j) \in S_\gamma$ **do**
  $\mathcal{H}_{\mathsf{CT}}[s, \tau_j] = \mathcal{H}_{\mathsf{CT}}[s, j]$
  $\mathcal{H}_{\mathsf{CT}}[s, j] = \perp$
if $\exists s \in S_{\mathsf{seeds}} : \mathcal{H}'_{\mathsf{CT}}[s, \cdot] \neq \perp$ **then**
  $\mathsf{flg}_{\mathsf{bad}} \leftarrow 1$
Encrypt $S_\gamma^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}((0,0)_{i=1}^{\mathsf{cap}})$
send $(T_\gamma, S_\gamma^{\mathsf{enc}}, \mathsf{seed}_\gamma)$
Server:
Store updated $T_\gamma, S_\gamma^{\mathsf{enc}}$ and seed_γ

$\mathcal{H}_{\mathsf{1C}}(x)$

if $\mathcal{H}_{\mathsf{1C}}[x] = \perp$ **then**
  $\mathcal{H}_{\mathsf{1C}}[x] \leftarrow \{0, \ldots, m-1\}$
return $\mathcal{H}_{\mathsf{1C}}[x]$

$\mathsf{Dummy1C.Search}(\mathsf{K}, \mathsf{qp} = (\mathsf{sp}, \mathsf{up}), \mathsf{cnt}; \mathsf{EDB})$

Client:
if $\mathsf{cnt} = 0$ **then return** $\emptyset$
Recompute value ℓ of the searched label via sp
Let $\mathcal{J} = (j : \mathsf{up}[j] = 1)$ be the indices of updates on the searched label
for all $j \in \mathcal{J}$ in ascending order **do**
  Set $P_i^\ell \leftarrow P_j$, where i is the index of j in $\mathcal{J}$
send $(P_i^\ell)_{i=1}^{\mathsf{cnt}}$
Server:
Initialize empty set R
for all $i \in \{1, \ldots, \mathsf{cnt}\}$ **do**
  Set $\gamma \leftarrow \mathcal{H}_{\mathsf{1C}}(P_i^\ell)$
  Add $\mathsf{HTLookup}(P_i^\ell, T_\gamma, \mathsf{seed}_\gamma)$ to R
send R

$\mathsf{Dummy1C.DummyUpdate}(\mathsf{K}; \mathsf{EDB})$

Client:
Set $i \leftarrow \mathsf{cnt}_{\mathsf{updt}} + 1$ and increment $\mathsf{cnt}_{\mathsf{updt}}$
Set $P_i \leftarrow \{0,1\}^{2\lambda}$
Set $\gamma \leftarrow \mathcal{H}_{\mathsf{1C}}(P_i)$
send γ
Server:
Send $(T_\gamma, S_\gamma^{\mathsf{enc}})$
Client:
Reencrypt all values in S_γ (stored by simulator)
Save state of $\mathcal{H}'_{\mathsf{CT}} \leftarrow \mathcal{H}_{\mathsf{CT}}$
Run $(S_{\mathsf{seeds}}, T_\gamma, \mathsf{seed}_\gamma) \leftarrow \mathsf{HTSetup}(\mathsf{Remap}(S_\gamma))$
for all $s \in S_{\mathsf{seeds}}$ and $(x_j, \tau_j) \in S_\gamma$ **do**
  $\mathcal{H}_{\mathsf{CT}}[s, \tau_j] = \mathcal{H}_{\mathsf{CT}}[s, j]$
  $\mathcal{H}_{\mathsf{CT}}[s, j] = \perp$
if $\exists s \in S_{\mathsf{seeds}} : \mathcal{H}'_{\mathsf{CT}}[s, \cdot] \neq \perp$ **then**
  $\mathsf{flg}_{\mathsf{bad}} \leftarrow 1$
Encrypt $S_\gamma^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}((0,0)_{i=1}^{\mathsf{cap}})$
send $(T_\gamma, S_\gamma^{\mathsf{enc}}, \mathsf{seed}_\gamma)$
Server:
Store updated $T_\gamma, S_\gamma^{\mathsf{enc}}$ and seed_γ

$\mathcal{H}_{\mathsf{CT}}(x)$

if $\mathcal{H}_{\mathsf{CT}}[x] = \perp$ **then**
  $\mathcal{H}_{\mathsf{CT}}[x] \leftarrow \{0, \ldots, (1+\varepsilon)\mathsf{cap}\}^2$
return $\mathcal{H}_{\mathsf{CT}}[x]$

Algorithm 9 The eight hybrid. Set $\mathsf{flg}_{\mathsf{bad}} \leftarrow 1$ if some keys from S_γ are queried to $\mathcal{H}_{\mathsf{CT}}$ before being revealed in Search.

Dummy1C.Setup(K, N)

Set $m \leftarrow \lceil N/(\delta(\lambda)\log(N))\rceil$
Set $\mathsf{cap} \leftarrow \mathcal{O}(\delta(\lambda)\log(N))$
for all $\gamma \in \{0,\dots,m-1\}$ **do**
  Set $S_\gamma \leftarrow (\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(0), k_i)_{i=1}^{\mathsf{cap}}$ for $k_i \leftarrow \{0,1\}^{2\lambda}$
  Run $(S_{\mathsf{seeds}}, T_\gamma, \mathsf{seed}_\gamma) \leftarrow \mathsf{HTSetup}(\mathsf{Remap}(S_\gamma))$
  for all $s \in S_{\mathsf{seeds}}$ and $(x_j, \tau_j) \in S_\gamma$ **do**
    $\mathcal{H}_{\mathsf{CT}}[s, \tau_j] = \mathcal{H}_{\mathsf{CT}}[s, j]$
    $\mathcal{H}_{\mathsf{CT}}[s, j] = \perp$
  Set $S^{\mathsf{enc}}_\gamma \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}((0,0)_{i=1}^{\mathsf{cap}})$
  Store S_γ and its keys K_γ locally
Initialize empty set S_P
Initialize counter $\mathsf{cnt}_{\mathsf{updt}} \leftarrow 0$
Initialize empty tables $\mathcal{H}_{\mathsf{1C}}, \mathcal{H}_{\mathsf{CT}}$
Set $\mathsf{flg}_{\mathsf{bad}} \leftarrow 0$
return output $\mathsf{EDB} = (S^{\mathsf{enc}}_\gamma, T_\gamma, \mathsf{seed}_\gamma)_{\gamma=0}^{m-1}$

Dummy1C.Update(K, op; EDB)

Client:
Set $i \leftarrow \mathsf{cnt}_{\mathsf{updt}} + 1$ and increment $\mathsf{cnt}_{\mathsf{updt}}$
Set $P_i \leftarrow \{0,1\}^{2\lambda}$
Set $\gamma \leftarrow \mathcal{H}_{\mathsf{1C}}(P_i)$
send γ
Server:
Send $(T_\gamma, S^{\mathsf{enc}}_\gamma)$
Client:
Reencrypt all values in S_γ (stored by simulator)
Replace an item in S_γ inserted during setup with $(\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(0), P_i)$
Save state of $\mathcal{H}'_{\mathsf{CT}} \leftarrow \mathcal{H}_{\mathsf{CT}}$
Run $(S_{\mathsf{seeds}}, T_\gamma, \mathsf{seed}_\gamma) \leftarrow \mathsf{HTSetup}(\mathsf{Remap}(S_\gamma))$
for all $s \in S_{\mathsf{seeds}}$ and $(x_j, \tau_j) \in S_\gamma$ **do**
  $\mathcal{H}_{\mathsf{CT}}[s, \tau_j] = \mathcal{H}_{\mathsf{CT}}[s, j]$
  $\mathcal{H}_{\mathsf{CT}}[s, j] = \perp$
if $\exists s \in S_{\mathsf{seeds}} : \mathcal{H}'_{\mathsf{CT}}[s, \cdot] \neq \perp$ **then**
  $\mathsf{flg}_{\mathsf{bad}} \leftarrow 1$
Encrypt $S^{\mathsf{enc}}_\gamma \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}((0,0)_{i=1}^{\mathsf{cap}})$
send $(T_\gamma, S^{\mathsf{enc}}_\gamma, \mathsf{seed}_\gamma)$
Server:
Store updated $T_\gamma, S^{\mathsf{enc}}_\gamma$ and seed_γ

$\mathcal{H}_{\mathsf{1C}}(x)$

if $\mathcal{H}_{\mathsf{1C}}[x] = \perp$ **then**
  $\mathcal{H}_{\mathsf{1C}}[x] \leftarrow \{0,\dots,m-1\}$
return $\mathcal{H}_{\mathsf{1C}}[x]$

Dummy1C.Search(K, qp = (sp, up), cnt; EDB)

Client:
if $\mathsf{cnt} = 0$ **then return** $\emptyset$
Recompute value ℓ of the searched label via sp
Let $\mathcal{J} = (j : \mathsf{up}[j] = 1)$ be the indices of updates on the searched label
for all $j \in \mathcal{J}$ in ascending order **do**
  Set $P_i^\ell \leftarrow P_j$, where i is the index of j in $\mathcal{J}$
Add $\{P_i^\ell\}_{i=1}^{\mathsf{cnt}}$ to S_P
send $(P_i^\ell)_{i=1}^{\mathsf{cnt}}$
Server:
Initialize empty set R
for all $i \in \{1,\dots,\mathsf{cnt}\}$ **do**
  Set $\gamma \leftarrow \mathcal{H}_{\mathsf{1C}}(P_i^\ell)$
  Add $\mathsf{HTLookup}(P_i^\ell, T_\gamma, \mathsf{seed}_\gamma)$ to R
send R

Dummy1C.DummyUpdate(K; EDB)

Client:
Set $i \leftarrow \mathsf{cnt}_{\mathsf{updt}} + 1$ and increment $\mathsf{cnt}_{\mathsf{updt}}$
Set $P_i \leftarrow \{0,1\}^{2\lambda}$
Set $\gamma \leftarrow \mathcal{H}_{\mathsf{1C}}(P_i)$
send γ
Server:
Send $(T_\gamma, S^{\mathsf{enc}}_\gamma)$
Client:
Reencrypt all values in S_γ (stored by simulator)
Save state of $\mathcal{H}'_{\mathsf{CT}} \leftarrow \mathcal{H}_{\mathsf{CT}}$
Run $(S_{\mathsf{seeds}}, T_\gamma, \mathsf{seed}_\gamma) \leftarrow \mathsf{HTSetup}(\mathsf{Remap}(S_\gamma))$
for all $s \in S_{\mathsf{seeds}}$ and $(x_j, \tau_j) \in S_\gamma$ **do**
  $\mathcal{H}_{\mathsf{CT}}[s, \tau_j] = \mathcal{H}_{\mathsf{CT}}[s, j]$
  $\mathcal{H}_{\mathsf{CT}}[s, j] = \perp$
if $\exists s \in S_{\mathsf{seeds}} : \mathcal{H}'_{\mathsf{CT}}[s, \cdot] \neq \perp$ **then**
  $\mathsf{flg}_{\mathsf{bad}} \leftarrow 1$
Encrypt $S^{\mathsf{enc}}_\gamma \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}((0,0)_{i=1}^{\mathsf{cap}})$
send $(T_\gamma, S^{\mathsf{enc}}_\gamma, \mathsf{seed}_\gamma)$
Server:
Store updated $T_\gamma, S^{\mathsf{enc}}_\gamma$ and seed_γ

$\mathcal{H}_{\mathsf{CT}}(x)$

Parse $x = (s, k)$
if $k \in K_\gamma$ for some γ or $k = P_i \notin S_P$ **then**
  Set $\mathsf{flg}_{\mathsf{bad}} \leftarrow 1$
if $\mathcal{H}_{\mathsf{CT}}[x] = \perp$ **then**
  $\mathcal{H}_{\mathsf{CT}}[x] \leftarrow \{0,\dots,(1+\varepsilon)\mathsf{cap}\}^2$
return $\mathcal{H}_{\mathsf{CT}}[x]$

Algorithm 10 The ninth hybrid. Delay reprogramming the oracle for search.

Dummy1C.Setup(K, N)

Set $m \leftarrow \lceil N/(\delta(\lambda)\log(N))\rceil$
Set $\mathsf{cap} \leftarrow \mathcal{O}(\delta(\lambda)\log(N))$
for all $\gamma \in \{0,\dots,m-1\}$ **do**
    Set $S_\gamma \leftarrow (\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(0), k_i)_{i=1}^{\mathsf{cap}}$ for $k_i \leftarrow \{0,1\}^{2\lambda}$
    Run $(S_{\mathsf{seeds}}, T_\gamma, \mathsf{seed}_\gamma) \leftarrow \mathsf{HTSetup}(\mathsf{Remap}(S_\gamma))$
    for all $s \in S_{\mathsf{seeds}}$ and $(x_j, \tau_j) \in S_\gamma$ **do**
        $\mathcal{H}_{\mathsf{CT}}[s, \tau_j] = \mathcal{H}_{\mathsf{CT}}[s, j]$
        $\mathcal{H}_{\mathsf{CT}}[s, j] = \perp$
    Set $S_\gamma^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}((0,0)_{i=1}^{\mathsf{cap}})$
    Store S_γ and its keys K_γ locally
Initialize empty set S_P
Store table T_{len} initialized with 0
Initialize counter $\mathsf{cnt}_{\mathsf{updt}} \leftarrow 0$
Initialize empty tables $\mathcal{H}_{\mathsf{1C}}, \mathcal{H}_{\mathsf{CT}}$
Set $\mathsf{flg}_{\mathsf{bad}} \leftarrow 0$
return output $\mathsf{EDB} = (S_\gamma^{\mathsf{enc}}, T_\gamma, \mathsf{seed}_\gamma)_{\gamma=0}^{m-1}$

Dummy1C.Search(K, qp = (sp, up), cnt; EDB)

Client:

if $\mathsf{cnt} = 0$ **then return** $\emptyset$
Recompute value ℓ of the searched label via sp
Let $\mathcal{J} = (j : \mathsf{up}[j] = 1)$ be the indices of updates on the searched label
for all $j \in \mathcal{J}$ in ascending order **do**
    Set $P_i^\ell \leftarrow P_j$, where i is the index of j in $\mathcal{J}$
Set $\mathsf{cnt}' \leftarrow T_{\mathsf{len}}[\ell]$
for all $P_i^\ell = P_j$ with $i > \mathsf{cnt}'$ **do**
    Replace some key k_x from S_γ with P_i^ℓ
    Reprogram $\mathcal{H}_{\mathsf{CT}}[\mathsf{seed}, P_i^\ell] \leftarrow \mathcal{H}_{\mathsf{CT}}[\mathsf{seed}, k_x]$ for all required values seed [15]
Set $T_{\mathsf{len}}[\ell] \leftarrow \mathsf{cnt}$
Add $\{P_i^\ell\}_{i=1}^{\mathsf{cnt}}$ to S_P
send $(P_i^\ell)_{i=1}^{\mathsf{cnt}}$

Server:

Initialize empty set R
for all $i \in \{1,\dots,\mathsf{cnt}\}$ **do**
    Set $\gamma \leftarrow \mathcal{H}_{\mathsf{1C}}(P_i^\ell)$
    Add $\mathsf{HTLookup}(P_i^\ell, T_\gamma, \mathsf{seed}_\gamma)$ to R
send R

Dummy1C.Update(K, op; EDB)

Client:

Set $i \leftarrow \mathsf{cnt}_{\mathsf{updt}} + 1$ and increment $\mathsf{cnt}_{\mathsf{updt}}$
Set $P_i \leftarrow \{0,1\}^{2\lambda}$
Set $\gamma \leftarrow \mathcal{H}_{\mathsf{1C}}(P_i)$
send γ

Server:

Send $(T_\gamma, S_\gamma^{\mathsf{enc}})$

Client:

Reencrypt all values in S_γ (stored by simulator)
Save state of $\mathcal{H}'_{\mathsf{CT}} \leftarrow \mathcal{H}_{\mathsf{CT}}$
Run $(S_{\mathsf{seeds}}, T_\gamma, \mathsf{seed}_\gamma) \leftarrow \mathsf{HTSetup}(\mathsf{Remap}(S_\gamma))$
for all $s \in S_{\mathsf{seeds}}$ and $(x_j, \tau_j) \in S_\gamma$ **do**
    $\mathcal{H}_{\mathsf{CT}}[s, \tau_j] = \mathcal{H}_{\mathsf{CT}}[s, j]$
    $\mathcal{H}_{\mathsf{CT}}[s, j] = \perp$
if $\exists s \in S_{\mathsf{seeds}} : \mathcal{H}'_{\mathsf{CT}}[s, \cdot] \neq \perp$ **then**
    $\mathsf{flg}_{\mathsf{bad}} \leftarrow 1$
Encrypt $S_\gamma^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}((0,0)_{i=1}^{\mathsf{cap}})$
send $(T_\gamma, S_\gamma^{\mathsf{enc}}, \mathsf{seed}_\gamma)$

Server:

Store updated $T_\gamma, S_\gamma^{\mathsf{enc}}$ and seed_γ

$\mathcal{H}_{\mathsf{1C}}(x)$

if $\mathcal{H}_{\mathsf{1C}}[x] = \perp$ **then**
    $\mathcal{H}_{\mathsf{1C}}[x] \leftarrow \{0,\dots,m-1\}$
return $\mathcal{H}_{\mathsf{1C}}[x]$

Dummy1C.DummyUpdate(K, op; EDB)

Client:

Set $i \leftarrow \mathsf{cnt}_{\mathsf{updt}} + 1$ and increment $\mathsf{cnt}_{\mathsf{updt}}$
Set $P_i \leftarrow \{0,1\}^{2\lambda}$
Set $\gamma \leftarrow \mathcal{H}_{\mathsf{1C}}(P_i)$
send γ

Server:

Send $(T_\gamma, S_\gamma^{\mathsf{enc}})$

Client:

Reencrypt all values in S_γ (stored by simulator)
Save state of $\mathcal{H}'_{\mathsf{CT}} \leftarrow \mathcal{H}_{\mathsf{CT}}$
Run $(S_{\mathsf{seeds}}, T_\gamma, \mathsf{seed}_\gamma) \leftarrow \mathsf{HTSetup}(\mathsf{Remap}(S_\gamma))$
for all $s \in S_{\mathsf{seeds}}$ and $(x_j, \tau_j) \in S_\gamma$ **do**
    $\mathcal{H}_{\mathsf{CT}}[s, \tau_j] = \mathcal{H}_{\mathsf{CT}}[s, j]$
    $\mathcal{H}_{\mathsf{CT}}[s, j] = \perp$
if $\exists s \in S_{\mathsf{seeds}} : \mathcal{H}'_{\mathsf{CT}}[s, \cdot] \neq \perp$ **then**
    $\mathsf{flg}_{\mathsf{bad}} \leftarrow 1$
Encrypt $S_\gamma^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}((0,0)_{i=1}^{\mathsf{cap}})$
send $(T_\gamma, S_\gamma^{\mathsf{enc}}, \mathsf{seed}_\gamma)$

Server:

Store updated $T_\gamma, S_\gamma^{\mathsf{enc}}$ and seed_γ

$\mathcal{H}_{\mathsf{CT}}(x)$

Parse $x = (s, k)$
if $k \in K_\gamma$ for some γ or $k = P_i \notin S_P$ **then**
    Set $\mathsf{flg}_{\mathsf{bad}} \leftarrow 1$
if $\mathcal{H}_{\mathsf{CT}}[x] = \perp$ **then**
    $\mathcal{H}_{\mathsf{CT}}[x] \leftarrow \{0,\dots,(1+\varepsilon)\mathsf{cap}\}^2$
return $\mathcal{H}_{\mathsf{CT}}[x]$

Algorithm 11 A description of the simulator.

$\mathsf{Setup}(N = \mathcal{L}^{\mathsf{dummy}}_{\mathsf{Stp}}(N, D, M))$

Sample encryption key $\mathsf{K}_{\mathsf{Enc}}$
Set $m \leftarrow \lceil N/(\delta(\lambda)\log(N))\rceil$
Set $\mathsf{cap} \leftarrow \mathcal{O}(\delta(\lambda)\log(N))$
for all $\gamma \in \{0, \ldots, m-1\}$ **do**
    Set $S_\gamma \leftarrow (\mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(0), k_i)_{i=1}^{\mathsf{cap}}$ for $k_i \leftarrow \{0,1\}^{2\lambda}$
    Run $(S_{\mathsf{seeds}}, T_\gamma, \mathsf{seed}_\gamma) \leftarrow \mathsf{HTSetup}(\mathsf{Remap}(S_\gamma))$
    for all $s \in S_{\mathsf{seeds}}$ and $(x_j, \tau_j) \in S_\gamma$ **do**
        $\mathcal{H}_{\mathsf{CT}}[s, \tau_j] = \mathcal{H}_{\mathsf{CT}}[s, j]$
        $\mathcal{H}_{\mathsf{CT}}[s, j] = \perp$
    Set $S^{\mathsf{enc}}_\gamma \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}((0,0)^{\mathsf{cap}}_{i=1})$
    Store S_γ and its keys K_γ locally
Initialize empty set S_P
Store table T_{len} initialized with 0
Initialize counter $\mathsf{cnt}_{\mathsf{updt}} \leftarrow 0$
Initialize empty tables $\mathcal{H}_{1\mathsf{C}}, \mathcal{H}_{\mathsf{CT}}$
Set $\mathsf{flg}_{\mathsf{bad}} \leftarrow 0$
return output $\mathsf{EDB} = (S^{\mathsf{enc}}_\gamma, T_\gamma, \mathsf{seed}_\gamma)_{\gamma=0}^{m-1}$

$\mathsf{Search}(((\mathsf{sp}, \mathsf{up}), \mathsf{cnt}) = \mathcal{L}^{\mathsf{dummy}}_{\mathsf{Srch}}(\ell))$

Client:

if $\mathsf{cnt} = 0$ **then return** $\emptyset$
Recompute value ℓ of the searched label via sp
Let $\mathcal{J} = (j : \mathsf{up}[j] = 1)$ be the indices of updates on the searched label
for all $j \in \mathcal{J}$ in ascending order **do**
    Set $P^\ell_i \leftarrow P_j$, where i is the index of j in $\mathcal{J}$
Set $\mathsf{cnt}' \leftarrow T_{\mathsf{len}}[\ell]$
for all $P^\ell_i = P_j$ with $i > \mathsf{cnt}'$ **do**
    Replace some key k_x from S_γ with P^ℓ_i
    Reprogram $\mathcal{H}_{\mathsf{CT}}[\mathsf{seed}, P^\ell_i] \leftarrow \mathcal{H}_{\mathsf{CT}}[\mathsf{seed}, k_x]$ for all required values seed
Set $T_{\mathsf{len}}[\ell] \leftarrow \mathsf{cnt}$
Add $\{P^\ell_i\}_{i=1}^{\mathsf{cnt}}$ to S_P
send $(P^\ell_i)_{i=1}^{\mathsf{cnt}}$

Server:

Initialize empty set R
for all $i \in \{1, \ldots, \mathsf{cnt}\}$ **do**
    Set $\gamma \leftarrow \mathcal{H}_{1\mathsf{C}}(P^\ell_i)$
    Add $\mathsf{HTLookup}(P^\ell_i, T_\gamma, \mathsf{seed}_\gamma)$ to R
send R

$(\mathsf{Dummy})\mathsf{Update}(\perp = \mathcal{L}^{\mathsf{dummy}}_{\mathsf{Updt}}(\mathsf{op}, \ell, v))$

Client:

Set $i \leftarrow \mathsf{cnt}_{\mathsf{updt}} + 1$ and increment $\mathsf{cnt}_{\mathsf{updt}}$
Set $P_i \leftarrow \{0,1\}^{2\lambda}$
Set $\gamma \leftarrow \mathcal{H}_{1\mathsf{C}}(P_i)$
send γ

Server:

Send $(T_\gamma, S^{\mathsf{enc}}_\gamma)$

Client:

Reencrypt all values in S_γ (stored by simulator)
Save state of $\mathcal{H}'_{\mathsf{CT}} \leftarrow \mathcal{H}_{\mathsf{CT}}$
Run $(S_{\mathsf{seeds}}, T_\gamma, \mathsf{seed}_\gamma) \leftarrow \mathsf{HTSetup}(\mathsf{Remap}(S_\gamma))$
for all $s \in S_{\mathsf{seeds}}$ and $(x_j, \tau_j) \in S_\gamma$ **do**
    $\mathcal{H}_{\mathsf{CT}}[s, \tau_j] = \mathcal{H}_{\mathsf{CT}}[s, j]$
    $\mathcal{H}_{\mathsf{CT}}[s, j] = \perp$
if $\exists s \in S_{\mathsf{seeds}} : \mathcal{H}'_{\mathsf{CT}}[s, \cdot] \neq \perp$ **then**
    $\mathsf{flg}_{\mathsf{bad}} \leftarrow 1$
Encrypt $S^{\mathsf{enc}}_\gamma \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}((0,0)^{\mathsf{cap}}_{i=1})$
send $(T_\gamma, S^{\mathsf{enc}}_\gamma, \mathsf{seed}_\gamma)$

Server:

Store updated $T_\gamma, S^{\mathsf{enc}}_\gamma$ and seed_γ

$\mathcal{H}_{1\mathsf{C}}(x)$

if $\mathcal{H}_{1\mathsf{C}}[x] = \perp$ **then**
    $\mathcal{H}_{1\mathsf{C}}[x] \leftarrow \{0, \ldots, m-1\}$
return $\mathcal{H}_{1\mathsf{C}}[x]$

$\mathcal{H}_{\mathsf{CT}}(x)$

Parse $x = (s, k)$
if $k \in K_\gamma$ for some γ or $k = P_i \notin S_P$ **then**
    Set $\mathsf{flg}_{\mathsf{bad}} \leftarrow 1$
if $\mathcal{H}_{\mathsf{CT}}[x] = \perp$ **then**
    $\mathcal{H}_{\mathsf{CT}}[x] \leftarrow \{0, \ldots, (1+\varepsilon)\mathsf{cap}\}^2$
return $\mathcal{H}_{\mathsf{CT}}[x]$

that setting, the previous efficiency features translate to constant storage efficiency, constant page efficiency for searches, and page efficiency $\widetilde{\mathcal{O}}(\log N)$ for updates.

Rationale. Instead of using 1C as the underlying allocation algorithm in Dummy1C, we could also try the same approach with 2C, *i.e.*, using m hash tables $T_0, ..., T_{m-1}$, each with a capacity of $\widetilde{\mathcal{O}}(\log \log N)$, where $m = N/\widetilde{\mathcal{O}}(\log \log N)$. During an update, the i-th value matching label cnt is inserted in the least loaded table amongst T_μ and T_ν for $(\mu, \nu) \leftarrow \mathcal{H}_{\mathsf{1C}}(P_i^\ell)$. During a search on label cnt, both tables T_μ and T_ν are queried on key P_i^ℓ for all $i \in \{1, ..., T_{\mathsf{len}}[\ell]\}$.

Indeed, we believe that this approach is fine if during HTSetup, the seed is chosen once at random, and in case of failure, the overflowing items are moved to a stash (see [37]). In that case, the positions $(\alpha_\mu, \beta_\mu) \leftarrow \mathcal{H}_{\mathsf{CT}}(\mathsf{seed}_\mu, P_i^\ell)$ (resp. $(\alpha_\nu, \beta_\nu) \leftarrow \mathcal{H}_{\mathsf{CT}}(\mathsf{seed}_\nu, P_i^\ell)$) and seed seed_μ (resp. seed_ν) in T_μ (resp. T_ν) are uniformly random. But as there are M tables, we cannot allow for client-side stashes due to efficiency. Therefore, we must repeat with a new seed if allocation fails (as is done in our 1C-based construction). For 2C, the retry procedure induces a specific distribution for the positions $\mathcal{H}_{\mathsf{CT}}(\mathsf{seed}_i, \cdot)$ for tables T_i. Indeed, observe that the set of positions $\{(\alpha_i, \beta_i) \mid (\alpha_i, \beta_i) \leftarrow \mathcal{H}_{\mathsf{CT}}(\mathsf{seed}_i, k_i)\}$ always constitute a valid cuckoo hashing assignment, where k_i are the keys used during setup. If on the other hand some of the keys in the set of positions were not part of the hash table setup input, then it might not constitute a valid cuckoo hashing assignment. Thus, if during a search the server observes that some α_i, β_i are "incompatible" (with respect to cuckoo hashing) with previously observed keys of table T_i, she can guess that one of the accesses in T_i was not a real access.

In summary, as the seeds are conditioned on yielding a valid cuckoo assignment for the keys supplied in setup, the server might infer information about which of the two tables T_μ or T_ν contains a searched value, and which does not. This allows to infer some information on the loads of the tables T_i.

Further, as $\mathsf{poly}(\lambda)$ dummy updates are allowed, some of the pairs of random bins dummy updates might be "incompatible" (with respect to 2C) with the inferred loads. This, in theory, would allow the server to non-trivially distinguish between dummy updates and real updates. While the above sketch does not constitute a concrete attack, we are not aware of an approach to argue that this information cannot be leveraged by an adversary on semantic dummy security.

F Deferred Security Analysis

F.1 Security of our Static EMM

We analyze security of S1C formally (cf. Section 3).

Theorem 2. *Let $N \geq \lambda$, F be a secure F mapping to $\{0,1\}^{4\lambda+\lceil \log N \rceil}$ and* Enc *be an* IND-CPA*-secure encryption scheme. We model the hash functions $\mathcal{H}_P$ and $\mathcal{H}_{1C}$ as random oracles. The scheme* S1C *is correct and has leakage $\mathcal{L}^{\mathsf{S1C}} = (\mathcal{L}^{\mathsf{S1C}}_{\mathsf{Stp}}, \mathcal{L}^{\mathsf{S1C}}_{\mathsf{Srch}})$, where $\mathcal{L}^{\mathsf{S1C}}_{\mathsf{Stp}}(\mathsf{MM}, N, M) = N$ and $\mathcal{L}^{\mathsf{S1C}}_{\mathsf{Srch}}(\ell) = (\mathsf{sp}, \mathsf{cnt})$.*

Proof. Let $\mathcal{A}$ be a PPT adversary. The simulator Sim proceeds as follows.

Setup. During setup, Sim receives $\mathcal{L}^{\mathsf{S1C}}_{\mathsf{Stp}}(\mathsf{MM}, N, M) = N$. It sets up a simulated table T_{len} by sampling N keys $K_1, ..., K_N \leftarrow \{0,1\}^{2\lambda}$ and values $X_1, ..., X_N \leftarrow \{0,1\}^{\lceil \log N \rceil}$, and setting $T_{\mathsf{len}}[K_i] \leftarrow X_i$. Also, it simulates T_{full} by sampling $P_i \leftarrow \{0,1\}^{2\lambda}$ and sets $T_{\mathsf{full}}[P_i] \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(0)$. (Note that 0 represents a dummy value of size $p \cdot |\mathcal{V}|_b$, *i.e.*, an entire page.) Similarly, it simulates $T_{\mathsf{val,i}}$ by setting $T_{\mathsf{val,i}}[P_{i,j}] \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(0)$, where $P_{i,j} \leftarrow \{0,1\}^{2\lambda}$ for $i \in [0, m-1]$ and $j \in \mathsf{cap}$. (Note that in this case, 0 represents a dummy value of size $|\mathcal{V}|_b$, *i.e.*, a single value.) It outputs the simulated $\mathsf{EMM} = (T_{\mathsf{len}}, T_{\mathsf{full}}, (T_{\mathsf{val},0}, ..., T_{\mathsf{val},m-1}))$.
The simulator stores the sampled keys K_i in a set $\mathcal{K}$, and the tokens P_i and $P_{i,j}$ in sets $\mathcal{P} = \{P_i\}_i$ and $\mathcal{P}_i = \{P_{i,j}\}_j$, respectively, for later.
Search. During a search query, the simulator obtains $\mathcal{L}^{\mathsf{S1C}}_{\mathsf{Srch}}(\ell) = (\mathsf{sp}, \mathsf{cnt})$ and simulates (K, μ, P') as follows. If the label ℓ was searched in the past, it outputs the same value (K, μ, P') as in the prior query.

Otherwise, it chooses a value $P' \leftarrow \{0,1\}^{2\lambda}$ and chooses $K \in \mathcal{K}$ at random, removing K from $\mathcal{K}$ in the process. It then sets $\mu \leftarrow T_{\mathsf{len}}[K] \oplus \mathsf{cnt}$. Finally, it programs $\mathcal{H}_P$ as follows. For $\mathsf{pgcnt} = \lceil \mathsf{cnt}/p \rceil$ and $1 \leq j < \mathsf{pgcnt}$, it programs $\mathcal{H}_P(P', j) \leftarrow P_j$ for some token $P_j \in \mathcal{P}$ and removes P_j from $\mathcal{P}$. Next, it sets $\gamma \leftarrow \mathcal{H}_{\mathsf{1C}}(P', \mathsf{pgcnt})$ and programs $\mathcal{H}_P(P', \mathsf{pgcnt}, j] \leftarrow P_{\gamma,j}$ for $j \in [\mathsf{cnt} \bmod p]$, where $P_{\gamma,j} \in \mathcal{P}_\gamma$ is chosen at random and removed subsequently from the set $\mathcal{P}_\gamma$. Finally, the simulator outputs (K, μ, P').

We show that the real and ideal world are indistinguishable with a standard hybrid argument.

Hybrid 0. Hybrid 0 is identical to the real game. Note that $\mathcal{H}_P$ and $\mathcal{H}_{\mathsf{1C}}$ is modeled as programmable random oracle.

Hybrid 1. Hybrid 1 is identical to Hybrid 0, except the F outputs (K_i, μ_i, P'_i) are drawn uniformly at random instead of being the output of F during setup. Under the PRF security of F, hybrid 1 and hybrid 0 are indistinguishable.

Hybrid 2. Hybrid 2 is identical to Hybrid 1, except all ciphertexts are replaced with encryptions of 0 (of appropriate length). under IND-CPA security of Enc, hybrid 2 and hybrid 1 are indistinguishable.

Hybrid 3. Hybrid 3 is identical to Hybrid 2, except the tables T_{full} and $T_{\mathsf{val},0}, \ldots, T_{\mathsf{val},\mathsf{m}-1}$ are replaced with the simulated tables, and the oracle $\mathcal{H}_P$ is programmed during search as in the simulator. Due to the change in hybrid 2, the values stored in the simulated tables are distributed as in the previous hybrid. Furthermore, observe that the tokens $P_i = \mathcal{H}_P(P', j)$ in search are distributed uniform over the set $\mathcal{P}$ (conditioned on not being chosen yet) as these are the keys for T_{full}. Also, the keys $P_{\mathsf{pgcnt},j} \leftarrow \mathcal{H}_P(P', \mathsf{pgcnt}, j)$ are distributed uniform over the remaining keys in $\mathcal{P}_\gamma$, as these are the keys for $T_{\mathsf{val},\gamma}$. Thus, the keys are identically distributed unless either (1) $\mathcal{A}$ queries some P'_i before the random oracle $\mathcal{H}_P$ was reprogrammed in search, (2) there is a collision in $\mathcal{H}_P$ or P'_i, or (3) either set $\mathcal{P}$ or $\mathcal{P}_\gamma$ is empty but a value must be chosen for simulation.

For (1), as P'_i is hidden from $\mathcal{A}$ until search and has high min-entropy, the reprogrammed oracle is distributed as in hybrid 2 in $\mathcal{A}$'s view. For (2), the probability that there are collisions is negligible due to a birthday bound argument. For (3), observe that for $\mathcal{P}$ contains $\lfloor \mathsf{cnt}/p \rfloor$ tokens after setup. Note that the multi-map contains at most $\lfloor \mathsf{cnt}/p \rfloor$ full pages of values. In search, exactly one token is removed for each full page, therefore $\mathcal{P}$ will always be non-empty when a token is retrieved and removed. Similarly, observe that γ is uniform for distinct search queries (except with negligible probability). Therefore, lemma 1 applies and we conclude that $\mathcal{P}_\gamma$ remains non-empty (if a token must be chosen from the set). In more detail, observe that per distinct search query at most p tokens are removed from $\mathcal{P}_\gamma$, as only incomplete pages are stored in $T_{\mathsf{val},\gamma}$. Further, γ is uniform for each distinct query (except with negligible probability as we excluded collisions amongst the tokens). In total, at most N distinct values are removed from all sets $\mathcal{P}_0, \ldots, \mathcal{P}_{m-1}$. Per search query, we can interpret the removal of $\mathsf{cnt} \bmod p$ tokens from $\mathcal{P}_\gamma$ as a ball of size $\mathsf{cnt} \bmod p$ that is inserted in bin γ in a 1C process. A bin overflows in the 1C process if and only if we must retrieve a token from an empty set $\mathcal{P}_\gamma$. This happens with negligible probability by lemma 1.

Hybrid 4. Hybrid 4 is identical to Hybrid 3, except the table T_{len} is replaced with the simulated table T_{len} and the values (K_i, μ_i) in the search output are computed as in the simulator. The distribution of both hybrids is identical: The table T_{len} contains uniform keys and entries in both hybrids, the distribution of (K_i, μ_i) in hybrid 3 for a query on a fresh label ℓ is uniform in the set $\mathcal{K}$, and μ_i is distributed such that $T_{\mathsf{len}}[K_i] = \mathsf{cnt}_i \oplus \mu_i$.

Observe that Hybrid 4 is distributed as the ideal game with simulator Sim. The statement follows.

F.2 Security of our Dynamic EMM

We analyze security of D1C formally (cf. Section 5).

Theorem 3. *Let $N \geq \lambda$, F be a secure F mapping to $\{0,1\}^{2\lambda}$ and* Enc *be an* IND-CPA*-secure encryption scheme. We model the hash functions $\mathcal{H}_P$ and $\mathcal{H}_{1C}$ as random oracles. The scheme* S1C *is correct and has leakage $\mathcal{L}^{\mathsf{D1C}} = (\mathcal{L}^{\mathsf{D1C}}_{\mathsf{Stp}}, \mathcal{L}^{\mathsf{D1C}}_{\mathsf{Srch}}, \mathcal{L}^{\mathsf{D1C}}_{\mathsf{Updt}})$, where $\mathcal{L}^{\mathsf{D1C}}_{\mathsf{Stp}}(N, M) = N$, $\mathcal{L}^{\mathsf{D1C}}_{\mathsf{Srch}}(\ell) = (\mathsf{qp}, \mathsf{cnt})$ and $\mathcal{L}^{\mathsf{D1C}}_{\mathsf{Updt}}(\ell, v) = \bot$.*

Proof. Let $\mathcal{A}$ be a PPT adversary. The simulator Sim is given access to a simulator Sim_{Σ} and proceeds as follows.

Setup. On setup, Sim receives $\mathcal{L}^{\mathsf{D1C}}_{\mathsf{Stp}}(N, M) = N$ and runs $\mathsf{EMM}_{\mathsf{full}} \leftarrow \mathsf{Sim}_{\Sigma}(N)$. Sim initializes an update counter $\mathsf{ctr} = 0$. Further, it sets up empty tables T_γ for $\gamma \in \{0, ..., m-1\}$ as follows. For table T_γ, it initializes an empty table, and then samples cap keys $k_{\gamma,1}, ..., k_{\gamma,\mathsf{cap}} \leftarrow \{0,1\}^{2\lambda}$, and sets $T_\gamma[k_{\gamma,i}] = \mathsf{Enc}(0\|0)$. It outputs $\mathsf{EMM} \leftarrow (\mathsf{EMM}_{\mathsf{full}}, (T_0, ..., T_{m-1}))$. The simulator stores the sampled keys $k_{\gamma,i}$ in a set $\mathcal{K}_\gamma$ for later.
Search. On search on label ℓ, Sim receives $\mathcal{L}^{\mathsf{D1C}}_{\mathsf{Srch}}(\ell) = (\mathsf{qp}, \mathsf{cnt}')$ with $\mathsf{qp} = (\mathsf{sp}, \mathsf{up})$. It runs $\mathsf{Sim}_{\Sigma}(\mathsf{qp}_{\Sigma}, \mathsf{pgcnt}_{\Sigma})$, where qp_{Σ} and pgcnt_{Σ} are chosen as follows.
- We set qp_{Σ} as the zero vector $(0, ..., 0)$ of length ctr. Note that Sim must set $\mathsf{qp}_{\Sigma}(j) = 1$ if a page matching ℓ was pushed to Σ for simulation to succeed. As the pages are pushed in random order, it is possible to recompute a suitable pattern qp_{Σ} given up: If up indicates that a full page is added or an incomplete page becomes full in epoch', then $\mathsf{qp}(\mathsf{epoch}' \cdot M + j) = 1$, where j is chosen at random not-yet chosen values in $[M]$. (If such a value j was chosen in a prior search query on the same label, as indicated by sp, then j is chosen consistently with the prior query.)
- We set pgcnt_{Σ} as the number of 1s in qp_{Σ}.

Then, if the search pattern sp indicates that ℓ was queried previously within the same epoch, then Sim outputs $(P_\ell, \mathsf{cnt}, \gamma)$ or nothing as in the prior query.
Else, it sets $\mathsf{cnt} \leftarrow \mathsf{cnt}' \bmod p$ and sets $\mathsf{epoch} \leftarrow \lceil \mathsf{ctr}/M \rceil$. If T_γ was not updated this epoch, then sets $\mathsf{epoch} \leftarrow \mathsf{epoch} - 1$. If $\mathsf{cnt} = 0$, then outputs nothing. Otherwise, samples $P_\ell \leftarrow \{0,1\}^{2\lambda}$ at random. To sample γ, it proceeds different depending on the regime:
- If $N < pM$, on the first search on ℓ, samples $\gamma \leftarrow \{0, ..., m-1\}$ and outputs γ. Else, it chooses γ as in the prior query.
- If $N \geq pM$, on the first search on ℓ, it samples a fresh value $\gamma \in \{0, ..., m-1\}$ (that was not yet chosen in a prior search query). Else, it chooses γ as in the prior query.

For $i \in [\mathsf{cnt}]$, it then programs $\mathcal{H}_P(P_\ell, i) = k_i$, where $k_i \in \mathcal{K}_\gamma$ and removes k_i from $\mathcal{K}_\gamma$ in the process. Finally, the simulator outputs $(P_\ell, \mathsf{cnt}, \gamma)$.
Update. On updates, Sim simply invokes Sim_{Σ} to simulate an update query. Also, it sets $\gamma \leftarrow \mathsf{ctr} - 1 \bmod M$ and sends γ if $\gamma < m$. On reception of T_γ, Sim sends back an empty table T'_γ that is simulated as in $\mathsf{Sim.Setup}$. Also, Sim updates $\mathcal{K}_\gamma$ accordingly.

We show that the real and ideal world are indistinguishable with a hybrid argument.

Hybrid 0. Hybrid 0 is identical to the real game. Note that $\mathcal{H}_P$ and $\mathcal{H}_{\mathsf{1C}}$ is modeled as programmable random oracle.

Hybrid 1. Hybrid 1 is identical to Hybrid 0, except the F outputs P are drawn uniformly at random from $\{0,1\}^{2\lambda}$ instead of being the output of F during setup. Under the PRF security of F, hybrid 1 and hybrid 0 are indistinguishable.

Hybrid 2. Hybrid 2 is identical to Hybrid 1, except all ciphertexts are replaced with encryptions of 0 (of appropriate length). under $\mathsf{IND\text{-}CPA}$ security of Enc, hybrid 2 and hybrid 1 are indistinguishable.

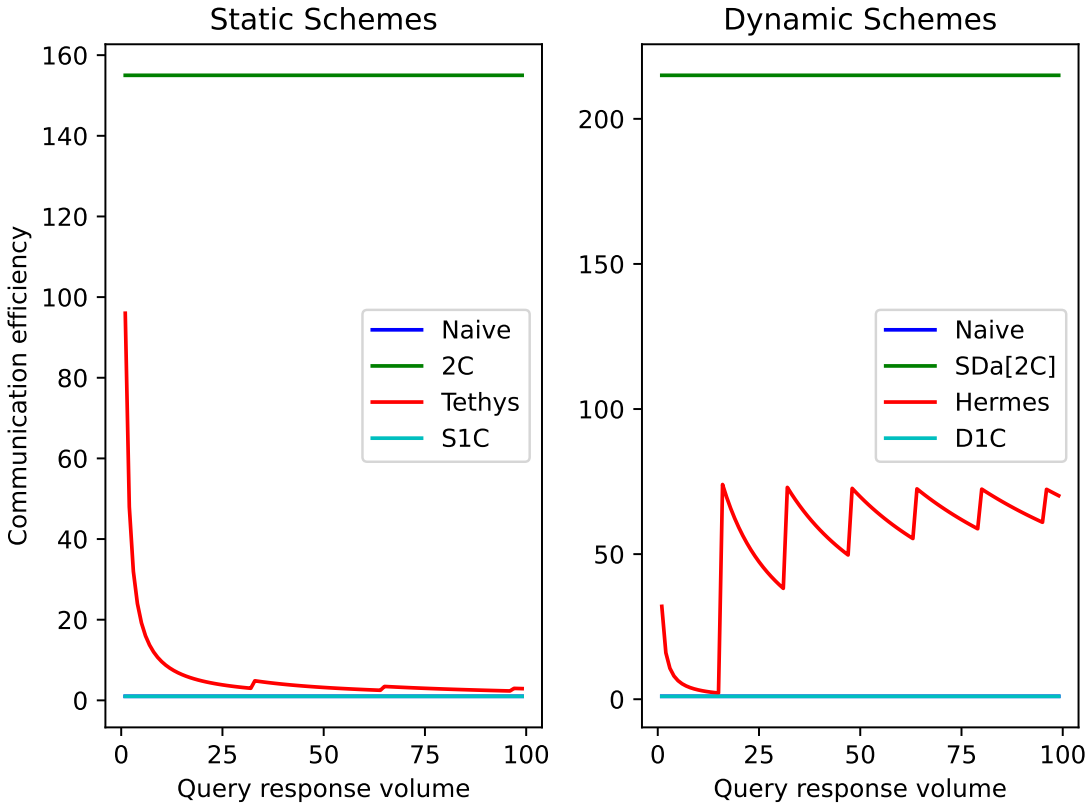
Hybrid 3. Hybrid 3 is identical to Hybrid 2, except the updates on Σ are induced by the simulated update pattern up_{Σ} for Σ. That is, the updates on Σ are reordered according to up_{Σ}. As π is a random permutation, both Hybrids are identically distributed.

Hybrid 4. Hybrid 4 is identical to Hybrid 3, except Σ is simulated as specified in Sim. Under EMM security with dummy updates of Σ both hybrids are indistinguishable.

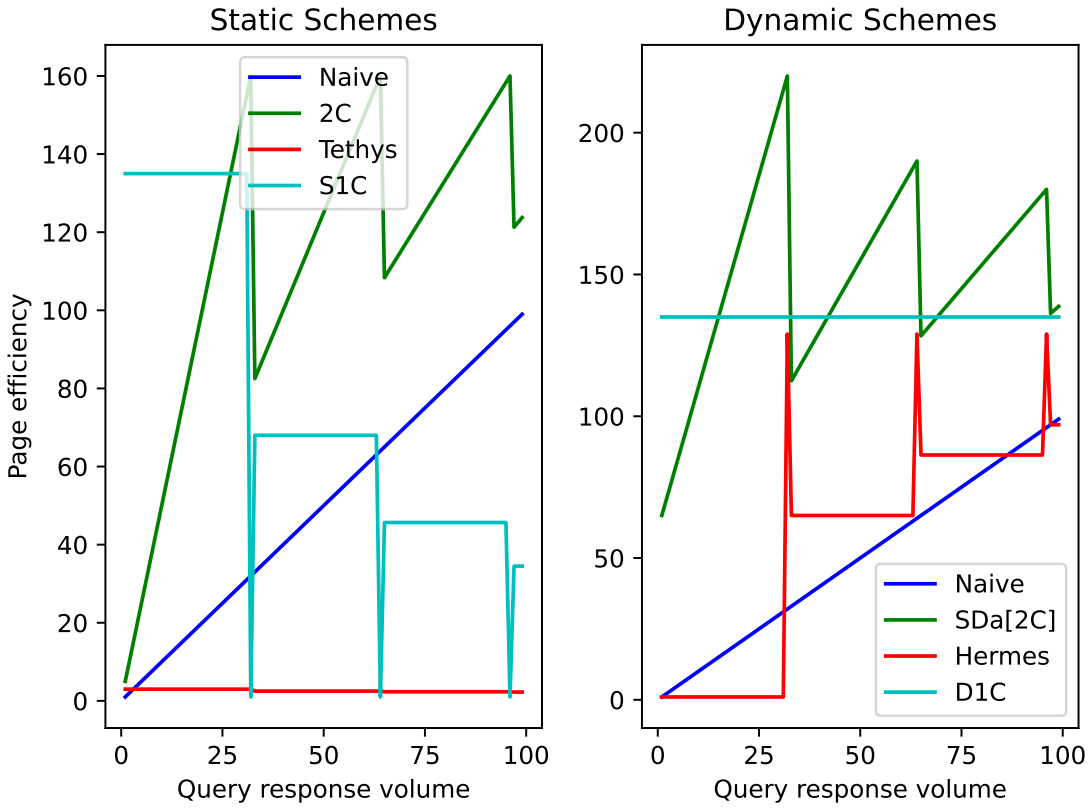
Finally, observe that Hybrid 4 is distributed as the ideal game.

G Additional In-Memory Experiments

In this section, we provide additional experiments on $\mathsf{S1C}$ and $\mathsf{D1C}$ run in memory. These experiments use the multi-maps derived from 200K emails. The results can be found in Figure 8.



(a) Communication efficiency.



(b) Page efficiency.

Fig. 7 – Comparisons of communication efficiency and page efficiency between S1C, D1C, and selected schemes. We use $N = 13,950,793$ and $p = 32$ in the comparison.

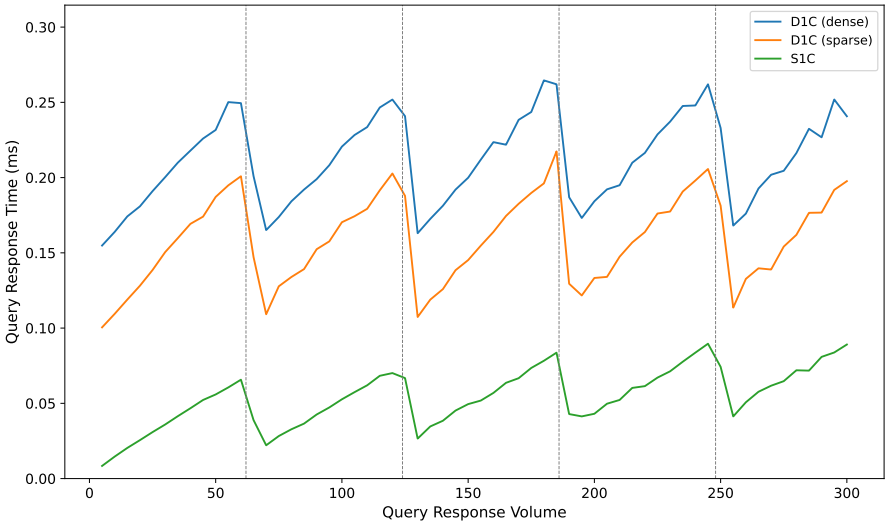


Fig. 8 – Search performance of S1C and D1C in memory.