Updatable Oblivious Key Value Stores with Access Control and Application to Multi Key Searchable Encryption

Benjamin Fuller*, Ariel Hamlin†, Arinjita Paul‡, Maryam Rezapour§, Ronak Sahu¶, Amey Shukla‖, Mason Stuart**

August 3, 2026

Abstract

Oblivious Key-Value Stores (OKVS) (Garimella et al., CRYPTO 2021), once encoded, provide indistinguishability over keys and random values. This is an important property in many secure computation applications, such as private set intersection and multi-key searchable encryption. We introduce an Updatable Oblivious Key-Value Store with access control (UOKVS), a dynamic extension of OKVS that supports insertions over time. We provide meaningful security in the presence of updates by equipping UOKVS with fine-grained access control. As a building block in UOKVS, we provide the first analysis of oblivious insertions for Cuckoo hashing, which may be of independent interest.

We show the application of UOKVS to multi-key searchable encryption where a data owner wishes to share parts of a multimap with multiple clients. We construct an oblivious multimap with insertions from UOKVS and private information retrieval (PIR). Unlike prior multi-key searchable encryption schemes, our construction supports sharing without replicating data across authorized users, substantially reducing storage costs in addition to stronger privacy guarantees.

We implement our multi-key searchable encryption construction on a dataset containing up to 24 million entries using the Enron email dataset. For keywords matching 100 documents on a WAN, query processing completes in 0.6 seconds using FrodoPIR as the underlying PIR protocol. By comparison, the scheme of Wang and Papadopoulos (Cloud Computing 2023) achieves a query time of 0.6 seconds and also incurs data replication and leaks access patterns. Our construction reduces leakage, maintains performance, and only requires a 3.1x storage overhead.

*University of Connecticut. Email: benjamin.fuller@uconn.edu
†Northeastern University. Email: a.hamlin@northeastern.edu
‡Niobium Microsystems/University of Connecticut. Email: arinjita.paul@gmail.com
§University of Connecticut. Email: maryam.rezapour@uconn.edu.
¶University of Connecticut. Email: ronak.sahu@uconn.edu.
‖University of Connecticut/Persona Identities. Email: amey.shukla@uconn.edu.
**Northeastern University. Email: stuart.ma@northeastern.edu.

Contents

1 Introduction

Oblivious Key-Value Stores (OKVS) [GPR+21, RR22, BPSY23, ZCL+24] have emerged as a fundamental primitive for secure multi-party computation. An OKVS allows a party to encode a key-value map so that another party can recover the value associated with a queried key while learning nothing else about the map. Consequently, OKVS constructions have become a common building block for private set intersection and union (PSI), private joins, and related secure computation protocols.

Existing OKVS constructions are inherently static: the encoded map is fixed at creation time. In contrast, many practical applications involve data that evolves over time and is stored on outsourced infrastructure. A data owner continuously inserts new records and wishes to selectively authorize external parties to access subsets of the data without revealing the remainder of the map. Existing OKVS definitions do not capture these requirements.

In this work, we introduce *Updatable Oblivious Key-Value Stores* (`UOKVS`). A `UOKVS` allows a data owner to incrementally insert key-value pairs while preserving the privacy guarantees of an `OKVS`. We further augment the primitive with access control, enabling the owner to authorize specific users to access selected portions of the stored data. The resulting abstraction models a private outsourced key-value store in which the server learns neither the contents of the map nor which portions are accessible to different users. This is true even when colluding with an arbitrary subset of clients.

Access control is essential for achieving meaningful privacy. It enables efficient updates while preventing the server from learning which keys may have been modified. Information about a key is revealed only to clients that have been authorized to access it.

Application to Multi-Key Searchable Encryption Our primary application is multi-key searchable encryption (MKSE) [PZ13]. Searchable encryption [SWP00, BHJP14, FVY+17, KKM+22, AGY+22, LMM+23] considers three types of parties: a data owner `Ow`, a server `S`, and clients, denoted as `C`. The owner stores a multimap `MM` associating keywords with collections of documents [KM18, KMO18, AG22, APP+23, GPPW24], while authorized clients retrieve documents matching selected keywords.

Existing MKSE systems leak substantial information through access-pattern and query-equality leakage across all clients, or replicate encrypted data for every authorized client and leak the above per client [PZ13, PSV+14, KOR+16, HSWW18, WP21, WP23]. Replication causes storage to grow linearly in the number of users and presents an obstacle to deployment.

Using `UOKVS` together with private information retrieval (PIR), we construct the first MKSE scheme that supports sharing without data replication. Rather than replicating encrypted data across users, the owner stores a single encrypted representation and distributes authorization information separately. This decouples storage from sharing while preserving strong privacy guarantees.

Our privacy guarantees are qualitatively stronger than prior work. Adversarial clients, in addition to their authorized items, learn only the insertion times of items they are permitted to access and the times at which those items are relocated. They learn nothing about the status of unauthorized keywords. In contrast, prior MKSE constructions reveal access patterns and query-equality information even when only the server is considered adversarial. Looking ahead to Section 6, we do this while **retaining the search efficiency of prior work on the benchmark Enron dataset.**

1.1 Overview of `UOKVS`

We review OKVS based on Cuckoo hashing. A keyword, denoted as `key`, is associated with a small set of s random candidate locations, and the value, denoted as `val`, is encoded so that only a party possessing the appropriate cryptographic material can recover it. Items are packed using standard Cuckoo hashing techniques such as random walk [PR01] or breadth first search (BFS) eviction [FPSS05]. Clients may retrieve these locations using PIR [ALP+21], thereby hiding their queries from the server [CGN97, MK22, PSY23, CD24, ALZ+25].

Supporting dynamic updates is considerably more challenging. PIR protects reads but not writes. During an insertion, the server directly observes the locations being updated. Combined with knowledge of the hash functions, this enables dictionary attacks [HSWW18] that substantially narrow the set of keys consistent

with an observed update. Avoiding such leakage appears to require touching many locations during every insertion. We address this challenge using two ideas.

First, keys are transformed using a pseudorandom function before hashing. Consequently, updated locations reveal no information about the underlying keyword while still allowing the owner to delegate access rights efficiently.

Second, we eliminate insertion access-pattern leakage by ensuring that memory locations accessed during insertion are pseudorandom. Achieving this property requires a new variant of Cuckoo hashing that we call *Cuckoo hashing with invalidation*. We analyze this for both random walk based eviction and BFS based eviction. We start with random walk based eviction to build intuition. Consider an array of ciphertexts and a small stash of ciphertexts for key, value pairs that could not be placed.

Random Walk To insert, we perform α random walk steps of the following form:

1. For $\mathsf{key}, \mathsf{val}$, choose a random position pos out of the other $s-1$ positions for key,
2. If key has already been in pos, place $\mathsf{key}, \mathsf{val}$ in the stash, switch walk to accessing random positions.
3. Else store $(\mathsf{key}, \mathsf{val})$ in pos, encrypt the pair. Let $(\mathsf{key}, \mathsf{val})$ be the previous contents of pos. If $(\mathsf{key}, \mathsf{val}) = (\perp, \perp)$ switch walk to accessing random positions.

When the walk accesses random positions, it always re-encrypts the current contents of accessed positions. The stash is re-encrypted with every insertion. To prevent access pattern leakage, positions for each key are invalidated once used (Step 2 prevents accessing twice).

To analyze this construction, we consider a graph where nodes are of the form $\mathsf{key}, \mathsf{pos}$ for each key and the s possible positions for key. To maintain obliviousness, insertions can only reach a node once. Whenever the walk terminates early or would encounter a previously seen node, we continue by accessing uniformly random positions. This modification ensures that the sequence of accessed locations is pseudorandom from the server's perspective. Even if an adversarial client has access to some keys, the rest of the sequence is pseudorandom.

This change comes at a cost. For γ total map size, recent work shows random-walk Cuckoo hashing achieves negligible stash size when $\alpha = \Theta(\log \gamma)$ [BF24]. Most analysis is in terms of the current load of the system, $\rho := \gamma/\mu$ where μ is the size of the storage array. We show that the expected stash size is at most

$$\mu \left(\rho^{\alpha+1} + \frac{\alpha \rho^2}{4(s-1)} + O(\rho^3) \right).$$

Our bound is larger than the ideal $\rho^{\alpha+1}$ for random-walk Cuckoo hashing [BF24], our experiments indicate that the bound is nearly tight when the invalidation constraints are turned on from the beginning. However, our analysis requires considering invalidations for all insertions, so our results are loose when many items are jointly prepared at setup rather than being inserted one at a time.

BFS This large stash can be overcome using BFS insertion. Rather than following a single random eviction chain, BFS explores a local search tree in a queue, placing all other positions of an item in a queue. The BFS eviction succeeds as long as an empty position is popped before popping $\alpha + 1$ items. As in random walk insertion, an item can reside in a position at most once to prevent access pattern leakage. The main benefit of BFS eviction is that popping an item that has been invalidated does not terminate the legitimate eviction search. Other positions for the item can still be considered.

For BFS search, the dominant source of stash entries is the probability that the BFS search queue empties before finding an empty position. The queue-extinction probability (see Section 3.2.4) is well approximated by

$$\left(\frac{\rho}{2(s-1)} \right)^s + O(\rho^{s+1}).$$

The BFS failure probability decreases as the s-th power of the load, in contrast to the $\Theta(\alpha\rho^2/(s-1))$ invalidation term arising from random-walk insertion. For BFS, we focus on $s = 4$, the queue-extinction contribution scales as $\Theta(\rho^4)$.

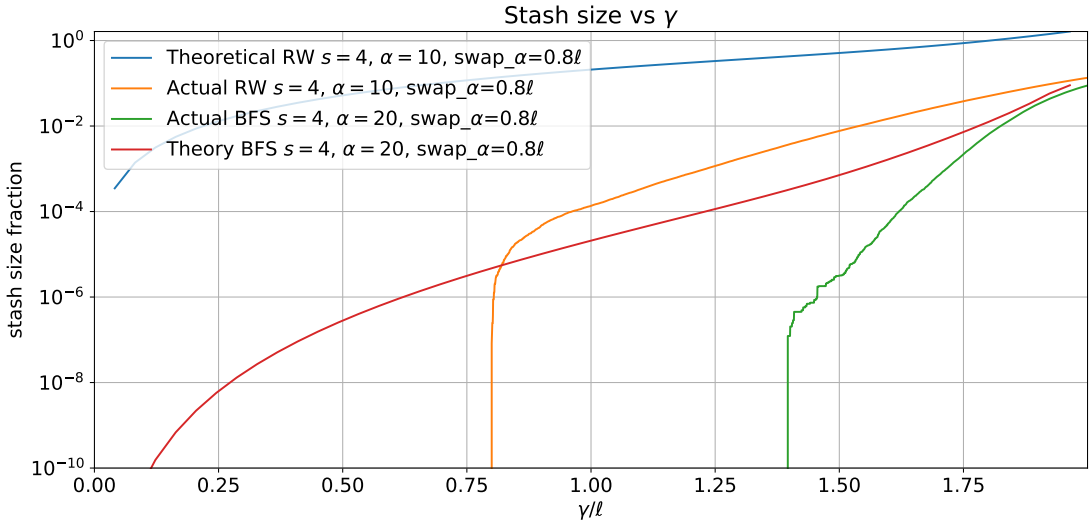

Figure 1: Stash size computed using Theorems 2 and BFS approximation in Section 3.2.1 with $s = 4$, $\ell = 24$M (size of Enron dataset), $\mu = 2.0\ell$, *swap_α* represents the size of the database that is jointly prepared. BFS implementation evicts at most 2 items at 10 levels of the graph.

For example, consider $\mu = 2.0\gamma$, $s = 4$, walk length $\alpha = 10$, if we jointly prepare the first $0.8\gamma =$ items and then insert individually the next $.2\gamma$ items, we observe stash sizes of roughly $10^{-4}\gamma$ for random walk based eviction and no stash for BFS based eviction. This is shown in Figure 1. We note that our implementation use a slightly different versions of RW and BFS (invalidating a pair only when it is moved).

In our analysis, we invalidate a $(\mathsf{key}, \mathsf{pos})$ once it is popped from the queue. A stronger version would invalidate the pair only when it is in the set of moved items. We were not able to analyze this stronger version, if edges can be popped multiple times, their distribution is not uniform. This creates a gap between our analysis and implementation that we hope can be closed by future work. We discuss more in Section 6.

Recursive Use of `UOKVS` As in many oblivious constructions, the stash itself can be stored in another `UOKVS` with a recursive construction. There are natural techniques to move things from level i of this recursion to level $i-1$ such as selecting k random locations from $\texttt{UOKVS}_{i-1}$ and trying to move items currently $\texttt{UOKVS}_i$ to those locations. Such a recursive organization may provide a path toward supporting overwrites and deletions without unbounded stash growth; we leave its formal construction and analysis to future work.

1.2 Other applications of `UOKVS`

In the same way that PIR and a `OKVS` yield a keyword PIR [ALP+21, HLP+24], our `UOKVS` plus PIR yield an updatable keyword PIR with access control. This is a crucial component in our MKSE construction.

`UOKVS` also has applications in the area of oblivious PRFs [HKLS24], updatable private set intersection [ADMT22, BMS+24, BMX22, WZC+24] with delegated storage. Many PSI protocols [GPR+21, LTT+25, XLW25] utilize `OKVS`. `UOKVS` provides a natural approach to support one party adding items to its set that is stored with a third party server.

1.3 Our Multi-Key Searchable Encryption

With our `UOKVS` in hand, we add PIR to hide `C`'s queries and show how to build a MKSE for multimaps that does not replicate data. The recipe is standard, we build two maps called `VolMap` and `ValMap` that map from key to volume i, and from (key, i) to document identifiers. We also implement a third map called `DocMap` from document identifiers to documents. When comparing with prior MKSE schemes, for consistency, we benchmark only the first two lookups.

We use FrodoPIR [DPC23] as the underlying PIR that we couple with our UOKVS. We use the Enron dataset [KY04] as our experimental database. The Enron dataset has 24M or $2^{24.5}$ keyword, identifier pairs and 33K or 2^{15} distinct keywords.

Our solution requires two network rounds, yielding a .6s search for a keyword with 100 matching documents on a 100Mbps network with 100ms latency. We compare with the MKSE state of the art, MUSSE [WP21], which has the strongest security guarantees. MUSSE requires 6 roundtrips and at least .6s delay assuming no computation time or bandwidth delay. MUSSE has query equality and access pattern leakage [CGPR15] on ValMap (which we do not have). Specific access pattern leakage has been thoroughly attacked in the literature [IKK12, CGPR15, KKNO16, WLD+17, GSB+17, GLMP18, MT19, KE19, KPT20, GPP23]. Our OKVS has an expansion factor of 2.0 for ValMap leading to an overall storage overhead of 3.1 over the raw Enron dataset.

As part of our construction, we implement the following improvements for FrodoPIR: 1) a hint update procedure, 2) an effective parallelization of search without sharding, and 3) a procedure for a client to issue a single query that works for all owners.

Recent leakage mitigation techniques for document retrieval such as SWiSSSE [GPPW24] and DDR-SSE [GMPP26] do not work in our setting as the owner can fingerprint the client's queries, we use PIR to protect the final lookup of DocMap as well. We do leak the padded volume of search. We bucket 256 items together. We do not propose new techniques for hiding volume [DPPS20, GKL+20, MVA+23]

Organization The rest of this work is organized as follows, Section 2 covers preliminaries, Section 3 introduces our UOKVS construction and presents bounds on stash size. Section 4 presents our MKSE design, Section 5 presents our access control mechanisms, Section 6 presents our implementation and experimental results, and Section 7 further reviews prior work. Code is available at our GitHub repository .

2 Preliminaries

We consider protocols involving at most two of the following three parties: a data owner Ow, server S, and a client C. Parties are always listed in the above order. As an example of notation, for an interactive protocol Prot between two parties A and B, we use notation $\begin{pmatrix} o_{\mathsf{A}} \\ o_{\mathsf{B}} \end{pmatrix} \leftarrow \mathsf{Prot}^{\mathsf{A},\mathsf{B}} \begin{pmatrix} i_{\mathsf{A}} \\ i_{\mathsf{B}} \end{pmatrix}$ with $i_{\mathsf{A}}, o_{\mathsf{A}}, i_{\mathsf{B}}, o_{\mathsf{B}}$ denoting A and B's inputs and outputs respectively. Throughout, we consider m clients and n owners but only a single client or owner participates in any protocol. The numbers n, m are not input to any protocol initialization. If a party does not participate in a protocol, they are elided. For variables, parties are listed in superscript while other indexing (such as sequence) is done in subscript.

A map M is an association between keywords key and a value val. Note that key does not represent a cryptographic key; for that purpose we use K, KOwn and KClt. A multimap MM associates keywords key and $\mathsf{vals}_{\mathsf{key}}$, the set of documents associated with each keyword. Keywords are over a universe $\mathcal{W}$ and each value is of the same length η. Values can be split and padded into chunks of size η. The total size of MM is $\ell \cdot \eta := \sum_{\mathsf{key} \in \mathcal{W}} |\mathsf{vals}_{\mathsf{key}}|$ where ℓ is the number of $(\mathsf{key}, \mathsf{vals}_{\mathsf{key}})$ pairs. Looking ahead we consider three distinct maps as part of our overall solution – VolMap, ValMap and DocMap – the type of key, val depends on the map.

For a multimap and some arbitrary binary operator $\leq$, $\mathsf{Arr}_{\mathtt{MM}}$ outputs the pairs (key, val) and is indexed starting from position 1 where $\mathsf{val} \in \mathsf{vals}_{\mathsf{key}}$, and $\mathsf{key} \in \mathtt{MM}$. The pairs $(\mathsf{key}_1, \mathsf{val}_1), (\mathsf{key}_2, \mathsf{val}_2)$ are ordered by keyword using $\leq$. In the case of pairs $(\mathsf{key}, \mathsf{val}_1), (\mathsf{key}, \mathsf{val}_2)$, they are ordered by comparing $\mathsf{val}_1, \mathsf{val}_2$ using $\leq$. We assume that the list MM[key] is returned in the same order as in $\mathsf{Arr}_{\mathtt{MM}}$ (using the comparator $\leq$). For a vector $\vec{a}$ of length ℓ and permutation $\pi : [1, \ell] \to [1, \ell]$ we use $\pi(\vec{a})$ to be a new array in the order specified by π. We use similar notation for a M to refer to the permutation applied to the underlying Arr. Simulators and parties are stateful. Notation is summarized in Table 1.

Definition 1 (Updatable Oblivious Key-Value Store)**.** *An updatable oblivious key-value store (UOKVS) is parameterized by a key and value universes $\mathcal{K}, \mathcal{V}$, security parameter λ, and consists of the following functions:* $\begin{pmatrix} \mathsf{KOwn} \\ \mathtt{EMap}, st \end{pmatrix} \leftarrow \mathsf{Encode}^{\mathtt{Ow},\mathtt{S}} \begin{pmatrix} \mathsf{M} \\ \perp \end{pmatrix}$*: The encode algorithm inputs* $\mathsf{M} = \{(\mathsf{key}_i, \mathsf{val}_i)\}_{i=1}^{\ell}$ *where* $(\mathsf{key}_i, \mathsf{val}_i) \in$

Notation	Meaning
M	Map
$\mathtt{MM}$	Multimap
Arr	Array representation of M or $\mathtt{MM}$
λ	Security parameter
η	Size of each val
ℓ	Number of keyword-document pairs
κ	Number of keywords in $\mathtt{MM}$
key	Keyword
$\mathsf{vals}_{\mathsf{key}}$	All values associated with keyword key
$\mathtt{Shr}$	Keywords that are requested to be shared
$\mathtt{Plcy}$	Policy
q	Number of queries/policy changes, superscripted by parties
m	Number of clients
n	Number of owners
KOwn	Owner's key
K	Key, superscripted by parties, subscripted by map, vector, or the scheme being encrypted.
q	PIR query
ans	Server's response to PIR query
μ	Number of total positions
s	Number of possible positions for each key in DIP
Pos	Possible locations for an item
σ	Size of the stash
α	Items considered in stash analysis
γ	Currently inserted items
ρ	Load of array γ/μ
δ	PIR state update
st	PIR hint update
B	Bucket capacity

Table 1: Summary of Notation

$\{\mathcal{K}, \mathcal{V}\}$*, and outputs to the server the encoding* $\mathtt{EMap}$ *or* $\perp$*.*
$\begin{pmatrix}\mathsf{KOwn}\\ \mathtt{EMap}\end{pmatrix} \leftarrow \mathsf{Insert}^{\mathtt{Ow},\mathtt{S}}\begin{pmatrix}\mathsf{KOwn},(\mathsf{key},\mathsf{val})\\ \mathtt{EMap}\end{pmatrix}$*: The insert algorithm takes a key-value pair* $(\mathsf{key},\mathsf{val})$*,* KOwn*, and the encoding* $\mathtt{EMap}$*, and outputs the updated encoding or* $\perp$*.*
$\begin{pmatrix}\perp\\ \mathsf{KClt}\end{pmatrix} \leftarrow \mathsf{Grant}^{\mathtt{Ow},\mathtt{C}}\begin{pmatrix}\mathsf{KOwn},\mathtt{Plcy}\\ \vec{\mathsf{key}}\end{pmatrix}$*: Delivers the secret keys capable of decrypting values associated with* $\vec{\mathsf{key}}$ *if the* $\mathtt{Plcy}$ *is satisfied.*
$\begin{pmatrix}\perp\\ \mathsf{val}\end{pmatrix} \leftarrow \mathsf{Decode}^{\mathtt{S},\mathtt{C}}\begin{pmatrix}\mathtt{EMap}\\ \mathsf{KClt},\mathsf{key}\end{pmatrix}$*: The decode algorithm takes as input* $\mathsf{key} \in \mathcal{K}$*, secret key* KClt*, and* $\mathtt{EMap}$*, and outputs the associated value* $\mathsf{val} \in \mathcal{V}$*.*

Correctness *Consider the Correctness Experiment shown in Figure 2. A* $\mathtt{UOKVS}$ *is correct if for all PPT adversaries* $\mathcal{A}$*,* $\Pr[\mathsf{Exp}_{\mathtt{Correct},\mathcal{A},\mathtt{UOKVS}}(1^\lambda) = 1] \geq 1 - \mathtt{ngl}(\lambda)$*.*

Obliviousness *Let* λ *be a security parameter. Consider the security game shown in Figure 2. A* $\mathtt{UOKVS}$ *is secure if for all PPT adversaries* $\mathcal{A}$ *there exists a PPT simulator* Sim *such that:*

$$\left|\begin{matrix}\Pr[\mathsf{Exp}_{\mathtt{Sec},\mathcal{A},\mathtt{UOKVS}}(1^\lambda) = 1]\\ -\Pr[\mathsf{Exp}_{\mathcal{A},\mathsf{Sim},\mathtt{UOKVS}}(1^\lambda) = 1]\end{matrix}\right| \leq \mathtt{ngl}(\lambda).$$

$\underline{\mathtt{Exp}_{\mathtt{Correct},\mathcal{A},\mathtt{UOKVS}}(1^\lambda, q)}$

$\mathcal{A}$ sends to $\mathcal{C}$:

1. Clients $\mathtt{Clients} = \{\mathtt{C}^j\}_{j=1}^n$ and owners $\mathtt{Owners} = \{\mathtt{Ow}^i\}_{i=1}^m$.
2. For each $\mathtt{Ow}$, a number of operation executions $q^{\mathtt{Ow}}$.
3. For each $\mathtt{Ow}$ and $i \le q^{\mathtt{Ow}}$, one of $(\mathsf{Insert}, \mathsf{key}_i, \mathsf{val}_i)$, $(\mathsf{Decode}, \mathtt{C}, \mathsf{key}_i)$, $(\mathsf{Grant}, \mathtt{C}, \vec{\mathsf{key}}_i, \mathtt{Plcy}_i)$.
4. For each $\mathtt{Ow}$, $\mathsf{M}^{\mathtt{Ow}}$.

$\mathcal{C}$ performs:

1. For each $\mathtt{Ow}$, run
$$\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}\\ \mathtt{EMap}_0^{\mathtt{Ow}}\end{pmatrix} \leftarrow \mathsf{Encode}^{\mathtt{Ow},\mathtt{S}}\begin{pmatrix}\mathsf{M}^{\mathtt{Ow}}\\ \bot\end{pmatrix}.$$
2. For $i = 1, \ldots, q^{\mathtt{Ow}}$:
   (a) If $\mathsf{op} = (\mathsf{Insert}, \mathsf{key}_i, \mathsf{val}_i)$:
      i. If key_i already exists in $\mathsf{M}^{\mathtt{Ow}}$, output 1.
      ii. Else,
$$\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}\\ \mathtt{EMap}_i^{\mathtt{Ow}}\end{pmatrix} \leftarrow \mathsf{Insert}^{\mathtt{Ow},\mathtt{S}}\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}, (\mathsf{key}_i, \mathsf{val}_i)\\ \mathtt{EMap}_{i-1}^{\mathtt{Ow}}\end{pmatrix}.$$
      Update $\mathsf{M}^{\mathtt{Ow}}$ to have the new pair.
   (b) If $\mathsf{op} = (\mathsf{Decode}, \mathtt{C}, \mathsf{key}_i)$:
$$\begin{pmatrix}\bot\\ \mathsf{val}_i\end{pmatrix} \leftarrow \mathsf{Decode}^{\mathtt{S},\mathtt{C}}\begin{pmatrix}\mathtt{EMap}^{\mathtt{Ow}}\\ \mathsf{KClt}, \mathsf{key}_i,\end{pmatrix}.$$
   (c) If $\mathsf{op} = (\mathsf{Grant}, \mathtt{C}, \vec{\mathsf{key}}_i, \mathtt{Plcy}_i)$:
$$\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}\\ \mathsf{KClt}\end{pmatrix} \leftarrow \mathsf{Grant}^{\mathtt{Ow},\mathtt{C}}\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}, \mathtt{Plcy}_i\\ \vec{\mathsf{key}}_i\end{pmatrix}.$$

Output 1 iff for all $(\mathtt{Ow}, \mathtt{C})$ and all key_i such that $\exists j \le i$ where $\mathsf{key}_i \in \vec{\mathsf{key}}_j$, $\mathtt{Plcy}_j(\vec{\mathsf{key}}_i) = 1$, it holds that
$$\mathsf{Decode}^{\mathtt{S},\mathtt{C}}\begin{pmatrix}\mathtt{EMap}^{\mathtt{Ow}}\\ \mathsf{key}_i, \mathsf{KClt}\end{pmatrix} = \mathsf{M}^{\mathtt{Ow}}[\mathsf{key}_i].$$

$\underline{\mathtt{Exp}_{\mathtt{Sec},\mathcal{A},\mathtt{UOKVS}}(1^\lambda, q)}$

$\mathcal{A}$ sends to $\mathcal{C}$:

1. $\mathtt{Clients} = \{\mathtt{C}^j\}_{j=1}^n$, $\mathtt{Owners} = \{\mathtt{Ow}^i\}_{i=1}^m$, and $q^{\mathtt{Ow}} < q$.
2. Corrupted parties $\mathtt{Owners}_{\mathcal{A}} \subseteq \mathtt{Owners}$ and $\mathtt{Clients}_{\mathcal{A}} \subseteq \mathtt{Clients}$.
3. For each $\mathtt{Ow}$, $\mathsf{M}^{\mathtt{Ow}} \leftarrow \mathcal{A}(1^\lambda)$ and $\mathsf{op}_1, \ldots, \mathsf{op}_{q^{\mathtt{Ow}}}$.

$\mathcal{C}$ performs:

1. For each $\mathtt{Ow}$,
$$\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}\\ \mathtt{EMap}^{\mathtt{Ow}}\end{pmatrix} \leftarrow \mathsf{Encode}^{\mathtt{Ow},\mathtt{S}}\begin{pmatrix}\mathsf{M}^{\mathtt{Ow}}\\ \bot\end{pmatrix}.$$
2. For $i = 1, \ldots, q$:
   (a) If $(\mathsf{Insert}, \mathsf{key}_i, \mathsf{val}_i)$:
$$\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}\\ \mathtt{EMap}^{\mathtt{Ow}}\end{pmatrix} \leftarrow \mathsf{Insert}^{\mathtt{Ow},\mathtt{S}}\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}, (\mathsf{key}_i, \mathsf{val}_i)\\ \mathtt{EMap}_{i-1}^{\mathtt{Ow}}\end{pmatrix}.$$
   (b) If $(\mathsf{Grant}, \mathtt{C}, \vec{\mathsf{key}}_i, \mathtt{Plcy}_i)$:
$$\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}\\ \mathsf{KClt}\end{pmatrix} \leftarrow \mathsf{Grant}^{\mathtt{Ow},\mathtt{C}}\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}, \mathtt{Plcy}_i\\ \mathsf{KClt}, \vec{\mathsf{key}}_i\end{pmatrix}.$$
   (c) If $(\mathsf{Decode}, \mathtt{C}, \mathsf{key}_i)$:
$$\begin{pmatrix}\bot\\ \mathsf{val}_i\end{pmatrix} \leftarrow \mathsf{Decode}^{\mathtt{S},\mathtt{C}}\begin{pmatrix}\mathtt{EMap}^{\mathtt{Ow}}\\ \mathsf{KClt}, \mathsf{key}_i\end{pmatrix}.$$
3. Send the views of parties in $\mathtt{Owners}_{\mathcal{A}} \cup \mathtt{Clients}_{\mathcal{A}}$ and $\mathtt{S}$ to $\mathcal{A}$.

$\mathcal{A}$ outputs a bit b.

$\underline{\mathtt{Exp}_{\mathsf{Sim},\mathcal{A},\mathtt{UOKVS}}(1^\lambda, q, \mathcal{L})}$

1. $\mathcal{A}$ sends all inputs as in $\mathtt{Exp}_{\mathtt{Sec},\mathcal{A},\mathtt{UOKVS}}$.
$$\mathsf{View} \leftarrow \mathsf{Sim}(\mathcal{L}(\mathtt{Owners}, \mathtt{Ow}_{\mathcal{A}}, \mathtt{C}_{\mathcal{A}}, \vec{\mathsf{M}}, \vec{\mathsf{op}})).$$
2. $\mathcal{A}$ outputs a bit b.

Figure 2: Correctness and security experiments for UOKVS.

2.1 Technical Tools

Data-independent packing We use a data-independent packing or DIP [BBF+21] scheme $\mathsf{DIP} = (\mathsf{Size}, \mathsf{Build}, \mathsf{Lookup})$ with interface
$$\begin{aligned}(\mu, \sigma) &\leftarrow \mathsf{DIP.Size}(\ell),\\ (\mathsf{Buckets}, \mathsf{Stash}, K_{\mathsf{DIP}}) &\leftarrow \mathsf{DIP.Build}(\mathsf{M}),\\ \vec{\mathsf{pos}} &\leftarrow \mathsf{DIP.Lookup}(\mu, K_{\mathsf{DIP}}, \mathsf{key}).\end{aligned}$$

Here, μ is the number of buckets, σ is the stash capacity, and $\mathsf{DIP.Lookup}$ returns the s candidate bucket positions associated with key. We use μ_ℓ, σ_ℓ to denote these values, which are fixed for a given ℓ. Correctness requires every $\mathsf{M}[\mathsf{key}]$ to occur either in one of these candidate buckets or in the stash, except with negligible probability.

TethysDIP [BBF+21] showed that the minimum size of the stash was equal to the max-flow of a graph with edges being keywords and nodes being the set of possible buckets. TethysDIP does not naturally support

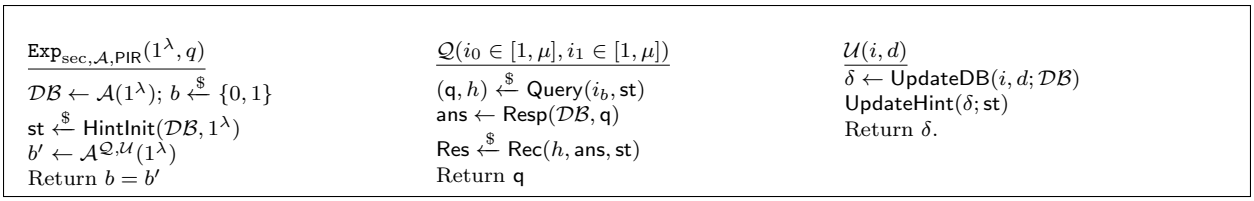

Figure 3: Security experiments for Updatable PIR.

insertions.

Our instantiation generates candidate positions using independent uniform hash functions. The build procedure places the initial items among their candidate buckets and the stash. We use random-walk or BFS Cuckoo eviction [Yeo23] for subsequent insertions; their precise randomness assumptions and stash-size bounds are given in Sections 3.1 and 3.2.

Notation We use notation $\mathsf{Buckets}[\mathsf{key}]$ to denote all buckets that contain a map value that was associated with key. When we define a multimap $\mathtt{MM}$, all documents have size η. When creating $\mathtt{ValMap}$, this would yield a map of size $\ell \sum_{\mathsf{key}} |\mathtt{MM}[\mathsf{key}]|\eta$. Buckets are of size B and can fit $\lfloor B/\eta \rfloor$ values, so one obtains a map of size approximately $\ell \sum_{\mathsf{key}} \lceil \frac{B|\mathtt{MM}[\mathsf{key}]|}{\rho} \rceil$. Notationally, we ignore the task of packing ρ/B items together in buckets, but this is handled by our implementation.

Definition 2. *(Updatable PIR [HPPY25]) An Updatable PIR (Private Information Retrieval) is a tuple of PPT algorithms* $(\mathsf{HintInit}, \mathsf{UpdateDB}, \mathsf{UpdateHint}, \mathsf{Query}, \mathsf{Resp}, \mathsf{Rec})$ *acting over a database* $\mathcal{DB} \in \{0,1\}^{B \times \mu}$ *with the following syntax:*

- $\mathit{st} \leftarrow \mathsf{HintInit}(\mathcal{DB}, 1^\lambda)$. *Inputs* $\mathcal{DB}$ *and outputs state* st.
- $\delta \leftarrow \mathsf{UpdateDB}(i, \mathsf{val}; \mathcal{DB})$. *Takes an update* (i, val)*, outputs a summary of the update* δ*, and changes* $\mathcal{DB}[i] \leftarrow \mathsf{val}$.
- $\mathit{st} \leftarrow \mathsf{UpdateHint}(\delta; \mathit{st})$. *Takes the summary of updates* δ *and modifies* st.
- $(\mathit{q}, h) \leftarrow \mathsf{Query}(i; \mathit{st})$. *Takes an* $i \in [1, \mu]$*, reads from* st*, returns a query* q*, and a reconstruction hint* h.
- $\mathit{ans} \leftarrow \mathsf{Resp}(\mathcal{DB}, \mathit{q})$. *Returns the server's response* ans *from* $\mathcal{DB}$ *and* q.
- $\mathsf{Res} \leftarrow \mathsf{Rec}(h, \mathit{ans}, \mathit{st})$. *Takes the hint* h*,* ans*,* st*, and outputs a result* Res*. May modify* st.

Correctness. *Fix an initial database* $\mathcal{DB}_0 \in \{0,1\}^{B \times \mu}$ *and let* $(i_1, \mathsf{val}_1), \ldots, (i_t, \mathsf{val}_t)$ *be any polynomial-length sequence of updates. Define recursively* $\delta_j \leftarrow \mathsf{UpdateDB}(i_j, \mathsf{val}_j; \mathcal{DB}_{j-1})$*, and* $\mathit{st}_j \leftarrow \mathsf{UpdateHint}(\delta_j; \mathit{st}_{j-1})$*, where* $\mathit{st}_0 \leftarrow \mathsf{HintInit}(\mathcal{DB}_0, 1^\lambda)$*, and let* $\mathcal{DB}_j$ *denote the resulting database after the* j*-th update. We say that the Updatable PIR is correct if for every* $t = \mathtt{poly}(\lambda)$ *and every index* $i \in [1, \mu]$*,*

$$\Pr\left[\begin{array}{c|r} \mathsf{Res} = & \forall j \in [t]: \\ \mathcal{DB}_t[i] & \mathit{st}_0 \leftarrow \mathsf{HintInit}(\mathcal{DB}_0, 1^\lambda) \\ & \delta_j \leftarrow \mathsf{UpdateDB}(i_j, \mathsf{val}_j; \mathcal{DB}_{j-1}) \\ & \mathit{st}_j \leftarrow \mathsf{UpdateHint}(\delta_j; \mathit{st}_{j-1}) \\ & (\mathit{q}, h) \leftarrow \mathsf{Query}(i; \mathit{st}_t) \\ & \mathit{ans} \leftarrow \mathsf{Resp}(\mathcal{DB}_t, \mathit{q}) \\ & \mathsf{Res} \leftarrow \mathsf{Rec}(h, \mathit{ans}, \mathit{st}_t) \end{array}\right] \geq 1 - \mathtt{ngl}(\lambda).$$

Security *We say that an Updatable PIR is secure if for all PPT adversaries* $\mathcal{A}$ *issuing at most* $q = \mathtt{poly}(\lambda)$ *queries to the oracles defined in Figure 3,* $\Pr[\mathsf{Exp}_{\mathrm{sec},\mathcal{A},\mathsf{PIR}}(\lambda, \mathcal{A}) = 1] \leq 1/2 + \mathtt{ngl}(\lambda)$.

2.2 Access Control

An *access control* mechanism encrypts an array corresponding to a M and, for every keyword, derives two cryptographic keys. The first is a pseudorandom lookup key used by the $\mathtt{UOKVS}$ to locate the keyword; the second decrypts values associated with the keyword. Access control can be viewed as a multi-key version of real-or-random security that additionally guarantees pseudorandomness of the lookup keys.

Definition 3. *Let* $\mathsf{AcCtrl} = (\mathsf{Setup}, \mathsf{Update}, \mathsf{Grant}, \mathsf{Decrypt})$*, where* Setup *and* Update *are randomized algorithms,* Grant *is a protocol between* $\mathtt{Ow}$ *and* $\mathtt{C}$*, and* $\mathsf{Decrypt}$ *is deterministic. Let* η, ℓ *be integers and let* $\mathcal{W}$ *be a finite keyword domain. For a keyword* key*, write* $K^{\mathsf{L}}_{\mathsf{key}}$ *for its lookup key and* $K^{\mathsf{D}}_{\mathsf{key}}$ *for its decryption key.*

1. $(\mathsf{CArr}, \mathsf{KOwn}) \leftarrow \mathsf{Setup}^{\mathtt{Ow}}(\mathsf{Arr})$*: takes an array* Arr *of length* ℓ*, whose entries are pairs* $(\mathsf{key}, \mathsf{val})$*, and returns an array of ciphertexts and a cryptographic key which consists of two maps* $K^{\mathsf{L}}, K^{\mathsf{D}}$*. If* $\mathsf{Arr}[i] = (\mathsf{key}, \mathsf{val})$*, then* $\mathsf{CArr}[i]$ *encrypts* val *under* $K^{\mathsf{D}}_{\mathsf{key}}$*. For* $\mathsf{key} \neq \perp$*, we write* $\mathsf{Loc}_{\mathsf{key}}$ *for the unique* i *such that the first component of* $\mathsf{Arr}[i]$ *is* key*. We also write* $\vec{K}^{\mathsf{L}}[i] = K^{\mathsf{L}}_{\mathsf{key}}$ *to indicate the lookup key for the item currently stored in location* i*.*

2. $(ct, \mathsf{KOwn}) \leftarrow \mathsf{Update}^{\mathtt{Ow}}(\mathsf{KOwn}, \mathsf{key}, \mathsf{val})$*: returns a fresh ciphertext encrypting* val *under* $K^{\mathsf{D}}_{\mathsf{key}}$*, and updated owner state. The caller chooses the array position at which the returned pair is installed.*

3. *The protocol*
$$\begin{pmatrix} \perp \\ \mathsf{KClt} \end{pmatrix} \leftarrow \mathsf{Grant}^{\mathtt{Ow},\mathtt{C}} \begin{pmatrix} \mathsf{KOwn}, \mathtt{Plcy} \\ \mathsf{KClt}, \vec{\mathsf{key}} \end{pmatrix}$$
augments KClt *with* $\mathsf{KClt}[\mathsf{key}] = (K^{\mathsf{L}}_{\mathsf{key}}, K^{\mathsf{D}}_{\mathsf{key}})$ *for each authorized* $\mathsf{key} \in \vec{\mathsf{key}}$*. On the first invocation for a client,* $\mathsf{KClt} = \perp$*.*

4. $\vec{\mathsf{val}} = \mathsf{Decrypt}^{\mathtt{C}}(\vec{c}, \{K^{\mathsf{D}}_{\mathsf{key}}\}_{\mathsf{key} \in \mathcal{A}})$ *decrypts a vector* $\vec{c} \subseteq \mathsf{CArr}$ *using the decryption-key components held by the client, where* $\mathcal{A}$ *is the set of keywords authorized for that client.*

An AcCtrl *is secure for leakage* $\mathcal{L} = (\mathcal{L}^{\mathsf{Setup}}, \mathcal{L}^{\mathsf{op}})$ *if the correctness and security requirements below hold for the experiments in Figure 4.*

The owner state contains persistent maps K^L *and* K^D*. On the first occurrence of* key*,* Setup *or* Update *initializes both entries; subsequent operations reuse them.*

Correctness *For every PPT* $\mathcal{A}$ *there is a negligible function* $\mathtt{ngl}$ *such that*

$$\Pr[\mathsf{Exp}_{\mathtt{Correct},\mathcal{A},\mathsf{AcCtrl}}(1^\lambda, q) = 1] \geq 1 - \mathtt{ngl}(\lambda).$$

Security *For every PPT* $\mathcal{A}$ *there exists a PPT* Sim *such that*

$$\left| \begin{array}{l} \Pr[\mathsf{Exp}_{\mathsf{Sec},\mathcal{A},\mathsf{AcCtrl}}(1^\lambda, q, \ell) = 1] \\ -\Pr[\mathsf{Exp}_{\mathsf{Sec},\mathcal{A},\mathsf{Sim},\mathsf{AcCtrl}}(1^\lambda, q, \ell, \mathcal{L}) = 1] \end{array} \right| \leq \mathtt{ngl}(\lambda).$$

Lookup-key pseudorandomness and collision freedom *Let* $\mathcal{A}$ *be any PPT adversary that does not receive the owner key, and let* $\vec{\mathsf{key}} = (\mathsf{key}_1, \ldots, \mathsf{key}_t)$ *be any polynomial-size sequence of pairwise distinct keywords chosen by* $\mathcal{A}$*, possibly adaptively. The joint distribution* $\left(K^{\mathsf{L}}_{\mathsf{key}_1}, \ldots, K^{\mathsf{L}}_{\mathsf{key}_t}\right)$ *of the lookup keys exposed through* Setup*,* Update*, and* Grant *is computationally indistinguishable from* $\big(R(\mathsf{key}_1), \ldots, R(\mathsf{key}_t)\big)$*, where* $R : \mathcal{K} \to \{0,1\}^\lambda$ *is a uniformly sampled random function. In particular,*

$$\Pr\Big[\exists\, i \neq j : K^{\mathsf{L}}_{\mathsf{key}_i} = K^{\mathsf{L}}_{\mathsf{key}_j}\Big] \leq \mathtt{ngl}(\lambda).$$

$\underline{\mathsf{Process}(\mathsf{op}_i)}$
First set

$$(\vec{K}^{\mathsf{L}}_i, \mathsf{CArr}_i, \mathsf{Arr}_i) := (\vec{K}^{\mathsf{L}}_{i-1}, \mathsf{CArr}_{i-1}, \mathsf{Arr}_{i-1}).$$

1. If $\mathsf{op}_i = (\mathsf{Update}, \mathsf{key}, \mathsf{val}, \mathcal{P}_i)$, run
$$(\mathsf{ct}, \mathsf{KOwn}) \leftarrow \mathsf{Update}^{\mathtt{Ow}}(\mathsf{KOwn}, \mathsf{key}, \mathsf{val}).$$
Let $j \leftarrow \mathcal{P}_i(K^{\mathsf{L}}[\mathsf{key}], \mathsf{st}_{\mathtt{Ow}})$, where $\mathcal{P}_i$ is executed by the owner. Set $\vec{K}^{\mathsf{L}}_i[j] := K^{\mathsf{L}}[\mathsf{key}]$, $\mathsf{CArr}_i[j] := \mathsf{ct}$, and $\mathsf{Arr}_i[j] := (\mathsf{key}, \mathsf{val})$.
2. If $\mathsf{op}_i = (\mathsf{Grant}, \mathtt{Plcy}, \vec{\mathsf{key}})$, run
$$\begin{pmatrix} \perp \\ \mathsf{KClt} \end{pmatrix} \leftarrow \mathsf{AcCtrl.Grant}^{\mathtt{Ow},\mathtt{C}} \begin{pmatrix} \mathsf{KOwn}, \mathtt{Plcy} \\ \mathsf{KClt}, \vec{\mathsf{key}} \end{pmatrix}.$$
3. If $\mathsf{op}_i = (\mathsf{Decrypt}, \mathsf{key}, \vec{\mathsf{pos}})$, set
$$\vec{\mathsf{val}}_i := \mathsf{AcCtrl.Decrypt}^{\mathtt{C}}\big(\mathsf{CArr}_i[\vec{\mathsf{pos}}], \mathsf{KClt}[\mathsf{key}].\mathsf{D}\big).$$

$\underline{\mathsf{Exp}^{\mathsf{Correct},\mathcal{A},\mathsf{AcCtrl}}(1^\lambda, q)}$

1. $\mathcal{A}$ sends an array Arr_0 of length ℓ.
2. Run $(\mathsf{CArr}_0, \mathsf{KOwn}_0) \leftarrow \mathsf{AcCtrl.Setup}^{\mathtt{Ow}}(\mathsf{Arr}_0)$, initialize $\mathsf{KClt} := \perp$, let $K^{\mathsf{L}}_0, K^{\mathsf{D}}_0 = \mathsf{KOwn}_0$, give $(K^{\mathsf{L}}_0, \mathsf{CArr}_0)$ to $\mathcal{A}$.
3. For $i = 1, \ldots, q$:
   (a) $\mathcal{A}$ sends op_i as a function of its current view.
   (b) Run $\mathsf{Process}(\mathsf{op}_i)$.
   (c) Give the resulting server state and protocol view to $\mathcal{A}$.
4. Output 1 iff:
   (a) distinct active keywords have distinct lookup keys; and
   (b) whenever key was authorized before round i and its current location belongs to $\vec{\mathsf{pos}}$, decryption returns the value associated with key.

   Otherwise output 0.

$\underline{\mathsf{Exp}^{\mathsf{Sec},\mathcal{A},\mathsf{AcCtrl}}(1^\lambda, q, \ell)}$

1. $\mathcal{A}$ sends Arr_0, $\mathsf{Auth}^\star$, and $1_{\mathtt{Ow},\mathcal{A}}, 1_{\mathtt{C},\mathcal{A}}$, where $1_{\mathtt{Ow},\mathcal{A}} + 1_{\mathtt{C},\mathcal{A}} \le 1$. Here $\mathsf{Auth}^\star$ contains the keywords that may eventually be authorized to a corrupted client.
2. Perform setup as in the correctness experiment. Give $\mathcal{A}$ the initial transcript and adversarial view.
3. For $i = 1, \ldots, q$:
   (a) $\mathcal{A}$ sends op_i as a function of its current view.
   (b) If the client is corrupted and a Grant authorizes a keyword outside $\mathsf{Auth}^\star$, output 0.
   (c) Run $\mathsf{Process}(\mathsf{op}_i)$.
   (d) Give $\mathcal{A}$ the resulting server state and adversarial protocol view.
4. Output the bit produced by $\mathcal{A}$.

$\underline{\mathsf{Exp}^{\mathsf{Sec},\mathcal{A},\mathsf{Sim}}(1^\lambda, q, \ell, \mathcal{L})}$

1. $\mathcal{A}$ sends the same initial inputs with the same constraints as in the real experiment.
2. Run
$$(\mathsf{View}_0, \mathsf{st}_{\mathsf{Sim}}) \leftarrow \mathsf{Sim.Init}\big(\mathcal{L}_{\mathsf{Setup}}(\mathsf{Arr}_0, \mathsf{Auth}^\star, 1_{\mathtt{Ow},\mathcal{A}}, 1_{\mathtt{C},\mathcal{A}})\big),$$
and send View_0 to $\mathcal{A}$.
3. For $i = 1, \ldots, q$:
   (a) $\mathcal{A}$ sends op_i.
   (b) Set $\tau_i := (\mathsf{Arr}_0, \mathsf{op}_1, \ldots, \mathsf{op}_i)$ and run
   $$(\mathsf{View}_i, \mathsf{st}_{\mathsf{Sim}}) \leftarrow \mathsf{Sim.Next}\big(\mathsf{st}_{\mathsf{Sim}}, \mathcal{L}_{\mathsf{Op}}(\tau_i)\big).$$
   (c) Send View_i to $\mathcal{A}$.
4. Output the bit produced by $\mathcal{A}$.

Figure 4: Correctness and security experiments for AcCtrl. The authorization envelope $\mathsf{Auth}^\star$ is fixed before setup, while the owner-local placement procedures and resulting update positions may depend on the current owner state and the preceding transcript.

3 Updatable Oblivious Key-Value Store

This section introduces our UOKVS. The construction combines DIP, access control, and PIR. DIP assigns each key to a small set of candidate positions, access control protects both the lookup keys and stored values, and PIR hides the positions queried by the client. The construction is presented in Figures 5–7 and consists of four functions.

Encode: The owner pads the input map to the combined capacity of the bucket array and stash using entries with distinct dummy labels. It then invokes AcCtrl.Setup on the padded array. For every key key, access control produces a lookup key $K^{\mathsf{L}}_{\mathsf{key}}$ and a ciphertext encrypting the corresponding value under a decryption key $K^{\mathsf{D}}_{\mathsf{key}}$.

The owner constructs a map whose keys are the lookup keys $K^{\mathsf{L}}_{\mathsf{key}}$ and invokes DIP.Build to assign the real entries to bucket or stash positions. The resulting placement induces a permutation of the aligned ciphertext array. The owner applies this permutation, randomly assigns the dummy ciphertexts to the

remaining positions, and divides the result into an encrypted bucket array CBuck and encrypted stash CStash. The owner retains the plaintext bucket and stash contents, together with a history of the lookup keys previously stored in each bucket. The server initializes the PIR state over CBuck and stores CStash separately.

Insert: To insert $(\mathsf{key}, \mathsf{val})$, the owner invokes $\mathsf{AcCtrl.Update}$ to create the lookup key $K^{\mathsf{L}}_{\mathsf{key}}$ and a fresh ciphertext for val. The lookup key determines the s candidate bucket positions through $\mathsf{DIP.Lookup}$. The owner chooses one candidate position uniformly and begins a random-walk insertion.

If the selected bucket is empty, the owner installs the item and its ciphertext. Otherwise, the current item replaces the item occupying that bucket, and the displaced item is moved to one of its other candidate positions. For every modified bucket, the owner records its position and new ciphertext in an update list Δ.

The history associated with a bucket records the lookup keys previously stored at that position. An item is not placed in a bucket that it previously occupied. If the next position violates this restriction, or if the walk reaches its maximum length α, the displaced item is placed in a uniformly random dummy stash position. The insertion fails if no dummy stash position remains.

After completing the walk, the owner re-encrypts every stash position, including dummy positions, and sends Δ and the complete encrypted stash to the server. For each $(\mathsf{pos}, \mathsf{ct}) \in \Delta$, the server updates the corresponding PIR database entry and its hint state. It then replaces the stored encrypted stash by the newly received CStash.

Grant: The grant protocol authorizes a client for a vector of keys according to the policy chosen by the owner. For every authorized key key, the client receives and stores $\mathsf{KClt}[\mathsf{key}] = \left(K^{\mathsf{L}}[\mathsf{key}], K^{\mathsf{D}}[\mathsf{key}]\right)$. The lookup key enables the client to determine the candidate DIP positions, while the decryption key enables it to recover ciphertexts associated with key. The access-control mechanism determines what the owner and client learn beyond the intended functionality.

Decode: To retrieve the value associated with key, the client parses $\mathsf{KClt}[\mathsf{key}] = (K^{\mathsf{L}}[\mathsf{key}], K^{\mathsf{D}}[\mathsf{key}])$ and computes the candidate positions

$$\vec{\mathsf{pos}} = \mathsf{DIP.Lookup}(\mu, K_{\mathsf{DIP}}, K^{\mathsf{L}}[\mathsf{key}]).$$

The client issues a PIR query for each candidate bucket position. The server returns the corresponding PIR responses together with the stash. The client recovers the candidate ciphertexts, appends the stash ciphertexts, and invokes $\mathsf{AcCtrl.Decrypt}$ using $K^{\mathsf{D}}[\mathsf{key}]$. Correctness of the lookup and decryption keys ensures that the ciphertext associated with key is among the examined ciphertexts and decrypts properly.

Remarks The construction is presented using random-walk Cuckoo hashing; Section 3.2 considers BFS eviction. Although the formal presentation assigns at most one item to each bucket, the implementation supports buckets containing multiple encrypted values.

Pseudorandom lookup keys hide the relationship between plaintext keys and their candidate DIP positions. At the same time, their consistency allows the owner to determine the positions needed for an insertion without rebuilding the data structure. The client learns a lookup key only when it is authorized for the corresponding plaintext key.

Our analysis has the owner retain the current contents of $\mathsf{Buckets}$ and Stash, together with the position history. A lower-storage variant can instead include the necessary state in the ciphertexts and have the owner download these items using PIR and decrypt an affected entry before moving it. This variant adds a round to the insertion protocol.

The construction inserts one key-value pair at a time, but multiple insertions can be processed as a batch. Batching is recommended because the encrypted stash is refreshed after every insertion.

We present the construction and its formal analysis for insertions. This setting lets each operation begin with a new item and follow a random walk or BFS traversal through the Cuckoo graph.

$$\begin{pmatrix}\mathsf{KOwn}\\ \mathtt{EMap},\mathsf{st}\end{pmatrix} \leftarrow \mathsf{Encode}^{\mathtt{Ow},\mathtt{S}}\begin{pmatrix}\mathsf{M}\\ \perp\end{pmatrix}$$

$\mathtt{Ow}$ does:

1. Compute $(\mu,\sigma) \leftarrow \mathsf{DIP.Size}(|\mathsf{M}|)$, set $\mathsf{Size}_\perp := \mu+\sigma-\ell$, and initialize $\mathsf{history}[1..\mu] = []$.
2. Append $\mathsf{Size}_\perp$ entries $(\perp_i,\perp)$ with distinct dummy labels, obtaining $\mathsf{Arr}_{\mathsf{M},\perp}$.
3. Run $(\mathsf{CArr},\mathsf{KOwn}^{\mathsf{AcCtrl}}) \leftarrow \mathsf{AcCtrl.Setup}^{\mathtt{Ow}}(\mathsf{Arr}_{\mathsf{M},\perp})$.
4. For each real entry $\mathsf{Arr}_\mathsf{M}[i] = (\mathsf{key}_i,\mathsf{val}_i)$, define $\mathsf{M_L}[\vec{K}^\mathsf{L}[i]] = (\mathsf{key}_i,\mathsf{val}_i)$.
5. $(\mathsf{Buckets},\mathsf{Stash},K_{\mathsf{DIP}}) \leftarrow \mathsf{DIP.Build}(\mathsf{M_L})$. Store each real entry as $(\mathsf{key},\mathsf{val})$, append $K^\mathsf{L}_{\mathsf{key}}$ to the history of its bucket.
6. Fill all empty bucket and stash positions with the distinct dummy entries.
7. Let π align each ciphertext with the bucket or stash position of its lookup key and randomly assign the dummy ciphertexts. Set $\mathsf{CBuck} \,\|\, \mathsf{CStash} = \pi(\mathsf{CArr})$, where $|\mathsf{CBuck}| = \mu$ and $|\mathsf{CStash}| = \sigma$.
8. Set $\mathsf{KOwn} = \mathsf{KOwn}^{\mathsf{AcCtrl}}$, retain $(\mathsf{Buckets},\mathsf{Stash},\mathsf{history})$, and send $(\mathsf{CBuck},\mathsf{CStash},\mu,\sigma,K_{\mathsf{DIP}})$ to $\mathtt{S}$.

$\mathtt{S}$ receives $(\mathsf{CBuck},\mathsf{CStash},\mu,\sigma,K_{\mathsf{DIP}})$, runs $\mathsf{st}^{\mathsf{PIR}} \leftarrow \mathsf{PIR.HintInit}(\mathsf{CBuck})$, and sets

$$\mathtt{EMap} := (\mathsf{st}^{\mathsf{PIR}},\mathsf{CBuck},\mathsf{CStash},\mu,\sigma,K_{\mathsf{DIP}}).$$

$$\begin{pmatrix}\mathsf{KOwn}\\ \mathsf{KClt}\end{pmatrix} \leftarrow \mathsf{Grant}^{\mathtt{Ow},\mathtt{C}}\begin{pmatrix}\mathsf{KOwn},\mathtt{Plcy}\\ \mathsf{KClt},\vec{\mathsf{key}}\end{pmatrix}$$

1. Run
$$\begin{pmatrix}\perp\\ \mathsf{KClt}\end{pmatrix} \leftarrow \mathsf{AcCtrl.Grant}^{\mathtt{Ow},\mathtt{C}}\begin{pmatrix}\mathsf{KOwn},\mathtt{Plcy}\\ \mathsf{KClt},\vec{\mathsf{key}}\end{pmatrix}.$$
2. For each authorized key, the client stores $\mathsf{KClt}[\mathsf{key}] = (K^\mathsf{L}_{\mathsf{key}},K^\mathsf{D}_{\mathsf{key}})$.

Figure 5: UOKVS encoding and authorization.

Updates and Deletes An existing value can be updated by inserting its new version through an unused candidate position and performing a complete eviction chain. The previous ciphertext remains in its former position, but this does not affect security because the server may retain all prior ciphertext versions. The same approach can represent deletion by inserting a distinguished deletion value.

This procedure applies as long as the item has an unused candidate position. Unlike the insertion-only setting, the number of remaining positions depends on the update pattern, so the insertion analysis does not directly provide a comparable stash-size bound for arbitrary sequences of updates and deletions.

Rebuild After sufficiently many insertions or updates, the owner can rebuild the encrypted data structure with fresh access-control state and a fresh DIP layout. Rebuilding restores the supply of unused candidate positions and reduces stash occupancy. The stash-size analysis therefore also provides guidance for choosing rebuild frequency.

Security We now turn to security which holds for both the random walk and BFS versions of the algorithm.

Theorem 1. *The construction in Figures 5–7 is a PD-OKVS with $\mathcal{L} = \mathcal{L}^{\mathsf{AcCtrl}}$ when the hash functions used in the* DIP *are replaced by random functions.*

Proof of Theorem 1. We separately consider correctness and security based on Definition 1.

Correctness By correctness of AcCtrl, all calls to Setup, Update, and Grant associate a fixed keyword key with the same lookup key $K^\mathsf{L}_{\mathsf{key}}$ and decryption key $K^\mathsf{D}_{\mathsf{key}}$, and lookup keys of distinct keywords collide only with negligible probability. Conditioned on the absence of such a collision, DIP.Build and DIP.Lookup use the same key and therefore identify the same candidate positions. Correctness then follows from correctness of DIP, AcCtrl, and PIR, provided every item can be placed in either the buckets or the stash. The stash-size theorems bound the probability that the chosen stash capacity is exceeded.

$$\begin{pmatrix} \mathsf{KOwn} \\ \mathtt{EMap}, \mathsf{st} \end{pmatrix} \leftarrow \mathsf{Insert}^{\mathtt{Ow},\mathtt{S}} \begin{pmatrix} \mathsf{KOwn}, (\mathsf{key}, \mathsf{val}) \\ \mathtt{EMap} \end{pmatrix}$$

$\mathtt{Ow}$ does:
Let $\mathsf{PutStash}(\mathsf{key}, \mathsf{val})$ replace a uniformly random dummy stash entry by $(\mathsf{key}, \mathsf{val})$, or output $\perp$ if no dummy entry remains. Let $\mathsf{Reencrypt}(\mathsf{pos})$ parse $\mathsf{Buckets}[\mathsf{pos}] = (\hat{\mathsf{key}}, \hat{\mathsf{val}})$, run $(\hat{\mathsf{ct}}, \mathsf{KOwn}^{\mathsf{AcCtrl}}) \leftarrow \mathsf{AcCtrl.Update}^{\mathtt{Ow}}(\mathsf{KOwn}^{\mathsf{AcCtrl}}, \hat{\mathsf{key}}, \hat{\mathsf{val}})$, set $\mathsf{CBuck}[\mathsf{pos}] = \hat{\mathsf{ct}}$, and append $(\mathsf{pos}, \hat{\mathsf{ct}})$ to Δ. This operation does not modify $\mathsf{Buckets}$ or $\mathsf{history}$.

1. Add $(\mathsf{key}, \mathsf{val})$ to M and initialize $(\mathsf{key}_0, \mathsf{val}_0) := (\mathsf{key}, \mathsf{val})$, $\Delta := []$, and $\mathsf{pseudorandom} := 0$.
2. For $i = 0, \ldots, \alpha - 1$:
   (a) If $\mathsf{pseudorandom} = 1$, sample $\mathsf{pos}_i \xleftarrow{\$} [1, \mu]$, run $\mathsf{Reencrypt}(\mathsf{pos}_i)$, and continue to the next iteration.
   (b) Otherwise, run $(\mathsf{ct}_i, \mathsf{KOwn}^{\mathsf{AcCtrl}}) \leftarrow \mathsf{AcCtrl.Update}^{\mathtt{Ow}}(\mathsf{KOwn}^{\mathsf{AcCtrl}}, \mathsf{key}_i, \mathsf{val}_i)$ and let $K_i^{\mathsf{L}} := K^{\mathsf{L}}[\mathsf{key}_i]$. If $i = 0$, sample $j_i \xleftarrow{\$} [1, s]$ and set $\mathsf{pos}_i := \mathsf{DIP.Lookup}(\mu, K_{\mathsf{DIP}}, K_i^{\mathsf{L}})[j_i]$.
   (c) If $K_i^{\mathsf{L}} \in \mathsf{history}[\mathsf{pos}_i]$, run $\mathsf{PutStash}(\mathsf{key}_i, \mathsf{val}_i)$, outputting $\perp$ if it fails. Then run $\mathsf{Reencrypt}(\mathsf{pos}_i)$ and set $\mathsf{pseudorandom} := 1$.
   (d) Else, if $\mathsf{Buckets}[\mathsf{pos}_i]$ is dummy, set $\mathsf{Buckets}[\mathsf{pos}_i] = (\mathsf{key}_i, \mathsf{val}_i)$, and $\mathsf{CBuck}[\mathsf{pos}_i] = \mathsf{ct}_i$. Append $(\mathsf{pos}_i, \mathsf{ct}_i)$ to Δ, append K_i^{L} to $\mathsf{history}[\mathsf{pos}_i]$, and set $\mathsf{pseudorandom} := 1$.
   (e) Otherwise, parse $\mathsf{Buckets}[\mathsf{pos}_i] = (\mathsf{key}_{i+1}, \mathsf{val}_{i+1})$. Set $\mathsf{Buckets}[\mathsf{pos}_i] = (\mathsf{key}_i, \mathsf{val}_i)$, and $\mathsf{CBuck}[\mathsf{pos}_i] = \mathsf{ct}_i$. Append $(\mathsf{pos}_i, \mathsf{ct}_i)$ to Δ and append $K^{\mathsf{L}}[\mathsf{key}_i]$ to $\mathsf{history}[\mathsf{pos}_i]$.
   (f) If $i = \alpha - 1$, run $\mathsf{PutStash}(\mathsf{key}_{i+1}, \mathsf{val}_{i+1})$, outputting $\perp$ if it fails. Otherwise, let j^* satisfy $\mathsf{DIP.Lookup}(\mu, K_{\mathsf{DIP}}, K^{\mathsf{L}}[\mathsf{key}_{i+1}])[j^*] = \mathsf{pos}_i$, sample $j_{i+1} \xleftarrow{\$} [1, s] \setminus \{j^*\}$, and set $\mathsf{pos}_{i+1} := \mathsf{DIP.Lookup}(\mu, K_{\mathsf{DIP}}, K^{\mathsf{L}}[\mathsf{key}_{i+1}])[j_{i+1}]$.
3. Re-encrypt every stash position. For $1 \le r \le \sigma$, letting $\mathsf{Stash}[r] = (\mathsf{key}_r, \mathsf{val}_r)$, run $(\mathsf{CStash}[r], \mathsf{KOwn}^{\mathsf{AcCtrl}}) \leftarrow \mathsf{AcCtrl.Update}^{\mathtt{Ow}}(\mathsf{KOwn}^{\mathsf{AcCtrl}}, \mathsf{key}_r, \mathsf{val}_r)$.
4. Set $\mathsf{KOwn} = \mathsf{KOwn}^{\mathsf{AcCtrl}}$ and send $(\Delta, \mathsf{CStash})$ to $\mathtt{S}$.

$\mathtt{S}$ receives $(\Delta, \mathsf{CStash})$. For every $(\mathsf{pos}, \mathsf{ct}) \in \Delta$, in order, it runs $\delta \leftarrow \mathsf{PIR.UpdateDB}(\mathsf{pos}, \mathsf{ct}; \mathsf{CBuck})$, and $\mathsf{PIR.UpdateHint}(\delta; \mathsf{st}^{\mathsf{PIR}})$. After processing all entries, it replaces its encrypted stash by CStash and sets $\mathtt{EMap} := (\mathsf{st}^{\mathsf{PIR}}, \mathsf{CBuck}, \mathsf{CStash}, \mu, \sigma, K_{\mathsf{DIP}})$.

Figure 6: $\mathtt{UOKVS}$ insertion. The update list Δ contains exactly α bucket updates; after the eviction chain terminates, the remaining updates re-encrypt uniformly random bucket positions.

Security Let $\mathcal{A}_{\mathtt{UOKVS}}$ be a PPT adversary. We construct a simulator $\mathsf{Sim}_{\mathtt{UOKVS}}$ using $\mathsf{Sim}_{\mathsf{AcCtrl}}$ and the security of PIR. Order the owner–client pairs and consider hybrids that replace one pair's real view at a time. It suffices to show that adjacent hybrids are indistinguishable.

For the selected owner–client pair, let $\mathsf{Auth}^\star$ be the union of the keywords authorized by the Grant operations in the precommitted UOKVS operation sequence. Initialize the online AcCtrl experiment with the initial array and $\mathsf{Auth}^\star$. Whenever the UOKVS owner invokes an access-control operation, forward that operation to the experiment online.

For an Update operation, $\mathcal{P}_i$ is the current UOKVS insertion. After Update initializes the keyword's persistent key-map entries, $\mathcal{P}_i$ reads $K^{\mathsf{L}}[\mathsf{key}]$ and the owner's placement state and returns the selected position. Consequently, the access-control experiment produces exactly the sequence of ciphertext updates and positions recorded by $\mathcal{L}_{\mathsf{AcCtrl}}$. In particular, if key is new, Update initializes its persistent lookup- and decryption-key entries before $\mathcal{P}_i$ selects its position. Hence a PD-OKVS insertion induces exactly the same access-control updates, including their positions, as are recorded by $\mathcal{L}^{\mathsf{AcCtrl}}$. Thus, the simulation of the lookup keys and their associated ciphertexts is entirely provided by $\mathsf{Sim}_{\mathsf{AcCtrl}}$.

For every honest owner, lookup-key pseudorandomness further implies that the joint distribution of

$$\begin{pmatrix} \perp \\ \mathsf{val} \end{pmatrix} \leftarrow \mathsf{Decode}^{\mathtt{S},\mathtt{C}} \begin{pmatrix} \mathtt{EMap} \\ \mathsf{key}, \mathsf{KClt} \end{pmatrix}$$

$\mathtt{C}$ does:

1. Parse $\mathsf{KClt}[\mathsf{key}] = (K^{\mathsf{L}}_{\mathsf{key}}, K^{\mathsf{D}}_{\mathsf{key}})$ and compute $\vec{\mathsf{pos}} = \mathsf{DIP.Lookup}(\mu, K_{\mathsf{DIP}}, K^{\mathsf{L}}_{\mathsf{key}})$.
2. For each $\mathsf{pos}_r \in \vec{\mathsf{pos}}$, run $(\mathsf{q}_r, h_r) \leftarrow \mathsf{PIR.Query}(\mathsf{pos}_r, \mathsf{st}^{\mathsf{PIR}})$ and send q_r to $\mathtt{S}$.

$\mathtt{S}$ computes $\mathsf{ans}_r = \mathsf{PIR.Resp}(\mathsf{CBuck}, \mathsf{q}_r)$ for each query and sends all ans_r, together with CStash, to $\mathtt{C}$.
$\mathtt{C}$ does:

1. Recover the encrypted candidate values: $\mathsf{CRes} = \|_r \, \mathsf{PIR.Rec}(h_r, \mathsf{ans}_r, \mathsf{st}^{\mathsf{PIR}})$.
2. Append the encrypted stash by setting $\mathsf{CRes} := \mathsf{CRes} \,\|\, (\|_{\mathsf{ct} \in \mathsf{CStash}} \mathsf{ct})$.
3. Output $\mathsf{val} = \mathsf{AcCtrl.Decrypt}(\mathsf{CRes}, K^{\mathsf{D}}_{\mathsf{key}})$.

Figure 7: UOKVS decoding. The lookup key determines the PIR positions, while the decryption key recovers the associated value.

the lookup keys exposed throughout the execution is computationally indistinguishable from a consistent random-function table. Consequently, constructing the DIP layout from these keys reveals no additional structure about the underlying keywords beyond the information already permitted by $\mathcal{L}^{\mathsf{AcCtrl}}$.

Replace the access-control view, including its lookup keys and ciphertexts, by the output of $\mathsf{Sim}_{\mathsf{AcCtrl}}$. Run DIP.Build and the insertion algorithm on the simulated lookup keys exactly as in the real construction. The resulting bucket and stash positions are distributed identically to those in the corresponding simulated access-control experiment. Finally, replace each PIR query by a query to a uniformly random position and generate the associated responses and hints using the current simulated encrypted array. Query indistinguishability of PIR make this final replacement indistinguishable.

If the selected client is corrupted and the owner is honest, the reduction forwards the client's Grant inputs to the access-control challenger and uses the returned authorized pairs $(K^{\mathsf{L}}[\mathsf{key}], K^{\mathsf{D}}[\mathsf{key}])$ to answer subsequent decodes. If the owner is corrupted and the client is honest, the reduction forwards the owner's inputs and uses the simulated client protocol view. If both are honest, it uses the simulated views of both parties. These are precisely the three cases covered by access-control security; the PIR replacement is the same in every case.

Thus a non-negligible difference between adjacent hybrids contradicts either security of AcCtrl or security of PIR. A polynomial-size hybrid argument over all owner–client pairs proves the theorem. □

3.1 Random Walk Stash Size

Theorem 2 (Random Walk Stash size after γ insertions)**.** *Consider an array of μ buckets and a sequence of γ insertions performed by* UOKVS.Insert *using random-walk eviction with s candidate locations per item and walk length α. Assume no lookup-key collisions and independent uniform hashes. Let $S_\gamma = \sum_{i=1}^{\gamma} Y_i$, where Y_i indicates that insertion i enters the stash. Let $H_i = \sum_{j=1}^{i} 1/j$. Define*

$$\Lambda = \sum_{i=1}^{\gamma} \left(\frac{i-1}{\mu}\right)^{\alpha} + \frac{\alpha(\alpha+1)\log\gamma}{s-1} + \sum_{i=1}^{\gamma} \frac{\alpha}{(s-1)(i-1)} \Big(\mu(H_\mu - H_{\mu-i+1}) - (i-1)\Big).$$

Then $\mathbb{E}[S_\gamma] \le \Lambda$. Furthermore, for every $\delta > 0$,

$$\Pr[S_\gamma \ge (1+\delta)\Lambda] \le \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{\Lambda}.$$

The rough intuition for the theorem is that $(i-1)/\mu$ is the current load denoted ρ_i of the array. If every location accessed were random, the probability that all α steps of the random walk were accessed would be

ρ_i^α. The main work of the theorem is to measure the probability that the random walk selects a position for an item key that has already been used. This immediately results in the item going into the stash. We do this by measuring the expected length of all previous walks, which is the third term in Λ. Conditioned on this not occurring, each step in the random walk is a (i, key) pair that has not been used in any prior insertions, and so $\mathsf{pos}_{i,\mathsf{key}}$ is uniform. The concentration bound follows from Freedman's inequality [Fre75] for martingale differences applied to the stash indicators $Y_1, \ldots, Y_\gamma$.

3.1.1 Proving Theorem 2

Before discussing the stash size, we first bound the probability that a random-walk insertion revisits a historical position of a previously inserted item. For insertion i, let $U_{i-1} = \sum_{j=1}^{i-1} L_j$ denote the total number of locations visited during the first $i-1$ insertions, and define

$$D_{i-1} = U_{i-1} - (i-1) = \sum_{j=1}^{i-1} (L_j - 1).$$

Since insertion j ultimately occupies exactly one location, $L_j - 1$ counts the locations visited during insertion j that are no longer occupied by that item. Thus D_{i-1} is the number of historical locations that have previously been occupied and later vacated. Let $\mathcal{F}_{i-1}$ denote the history of the first $i-1$ insertions.

Lemma 1 (Revisit probability)**.** *Conditioned on $\mathcal{F}_{i-1}$, let R_i denote the number of times insertion i attempts to place an item into one of its own historical locations. Then*

$$\Pr[R_i \geq 1 \mid \mathcal{F}_{i-1}] \leq \frac{\alpha D_{i-1} + \binom{\alpha}{2}}{(s-1)(i-1)}.$$

Proof. The random walk performs at most α relocation steps. Before the first relocation there are exactly D_{i-1} historical $(\mathsf{key}, \mathsf{pos})$ pairs among the $(s-1)(i-1)$ alternate locations of previously inserted items.

Each successful relocation vacates one additional position, creating at most one new historical location. Consequently, immediately before the $(t+1)$-st relocation ($0 \leq t < \alpha$), there are at most $D_{i-1} + t$ historical locations. Therefore the conditional probability that the next relocation selects a historical location is at most

$$\frac{D_{i-1} + t}{(s-1)(i-1)}.$$

Applying a union bound over the at most α relocation steps,

$$\begin{aligned}
\Pr(R_i \geq 1 \mid \mathcal{F}_{i-1}) &\leq \sum_{t=0}^{\alpha-1} \frac{D_{i-1} + t}{(s-1)(i-1)} \\
&= \frac{\alpha D_{i-1} + \sum_{t=0}^{\alpha-1} t}{(s-1)(i-1)} \\
&= \frac{\alpha D_{i-1} + \binom{\alpha}{2}}{(s-1)(i-1)},
\end{aligned}$$

which proves the claim.

□

Lemma 2 (Expected stash contribution)**.** *Let Y_i indicate that insertion i enters the stash. Then*

$$\mathbb{E}[Y_i] \leq \left(\frac{i-1}{\mu}\right)^\alpha + \frac{\alpha^2}{(s-1)(i-1)} + \frac{\alpha}{(s-1)(i-1)} \Big(\mu(H_\mu - H_{\mu-i+1}) - (i-1)\Big),$$

where $H_n = \sum_{k=1}^{n} \frac{1}{k}$ is the n-th harmonic number.

Random Walk	
L_i	Number of positions visited during insertion i
$U_i = \sum_{j=1}^{i} L_j$	Total positions visited after i insertions
$D_i = U_i - i$	Historical (invalid) positions created by prior walks
R_i	Number of revisits during insertion i
Breadth-First Search	
L_i	Number of positions explored during BFS insertion i
$v_i = \sum_{j=1}^{i} L_j$	Total positions explored after i insertions
λ_i	Queue branching parameter
q_i	Upper bound on queue-extinction probability
Shared Parameters	
$\rho_i = (i-1)/\mu$	Table load before insertion i
μ	Number of table positions
γ	Number of insertions
s	Candidate positions per key
α	Maximum walk/BFS queue pops
Y_i	Indicator that insertion i enters the stash
$S_\gamma = \sum_{i=1}^{\gamma} Y_i$	Total stash size after γ insertions

Table 2: Notation used throughout the random-walk and BFS analyses.

Proof. Consider insertion i, and let Y_i be the indicator that the item enters the stash. By definition, the stash event occurs if either the random-walk eviction exceeds α steps, or $R_i \geq 1$.

We first note that having items in the stash cannot increase the probability that a random walk exceeds α steps or encounters a historical location. Moving an item from the array to the stash may cause a future walk to terminate earlier, but it cannot extend a walk or create a revisit that would not already occur if the item remained in the array. Therefore, for the purpose of obtaining an upper bound, we may assume that all previously inserted items reside in the array. By a union bound,

$$\Pr[Y_i = 1 \mid \mathcal{F}_{i-1}] \leq \Pr[L_i > \alpha \mid \mathcal{F}_{i-1}] + \Pr[R_i \geq 1 \mid \mathcal{F}_{i-1}].$$

For the overflow term,

$$\Pr[L_i > \alpha \mid \mathcal{F}_{i-1}] \leq \left(\frac{i-1}{\mu}\right)^{\alpha}.$$

For the revisit term, Lemma 1 gives

$$\Pr[R_i \geq 1 \mid \mathcal{F}_{i-1}] \leq \frac{\alpha D_{i-1} + \binom{\alpha}{2}}{(s-1)(i-1)}.$$

Therefore

$$\Pr[Y_i = 1 \mid \mathcal{F}_{i-1}] \leq \left(\frac{i-1}{\mu}\right)^{\alpha} + \frac{\alpha D_{i-1} + \binom{\alpha}{2}}{(s-1)(i-1)}.$$

Taking expectations yields

$$\mathbb{E}[Y_i] \leq \left(\frac{i-1}{\mu}\right)^{\alpha} + \frac{\alpha\, \mathbb{E}[D_{i-1}] + \binom{\alpha}{2}}{(s-1)(i-1)}.$$

Since $D_{i-1} = \sum_{j=1}^{i-1}(L_j - 1)$, we have

$$\mathbb{E}[D_{i-1}] = \sum_{j=1}^{i-1}(\mathbb{E}[L_j] - 1).$$

Using the tail-sum formula, $\mathbb{E}[L_j] = \sum_{t\geq 1} \Pr(L_j \geq t)$. At insertion j, each eviction succeeds with probability at most $\rho_j = (j-1)/\mu$, so $\Pr(L_j \geq t) \leq \rho_j^{t-1}$. Hence

$$\mathbb{E}[L_j] \leq \sum_{t\geq 1} \rho_j^{t-1} = \frac{1}{1-\rho_j} = \frac{\mu}{\mu - j + 1}.$$

Therefore

$$\mathbb{E}[D_{i-1}] \le \sum_{j=1}^{i-1} \left(\frac{\mu}{\mu - j + 1} - 1 \right).$$

Reindexing gives

$$\mathbb{E}[D_{i-1}] \le \mu(H_\mu - H_{\mu-i+1}) - (i-1).$$

Substituting this bound yields

$$\begin{aligned}\mathbb{E}[Y_i] \le \left(\frac{i-1}{\mu}\right)^\alpha + \left(\frac{\binom{\alpha}{2}}{(s-1)(i-1)}\right) \\ + \frac{\alpha(\alpha+1)}{(s-1)(i-1)}\Big(\mu(H_\mu - H_{\mu-i+1}) - (i-1)\Big),\end{aligned}$$

which completes the proof of Lemma 2. □

Summing the additive $\alpha^2/((s-1)(i-1))$ term from Lemma 2 over $i = 2, \ldots, \gamma$ gives

$$\sum_{i=2}^{\gamma} \frac{\alpha^2}{(s-1)(i-1)} = \frac{\alpha^2}{s-1} H_{\gamma-1} \le \frac{\alpha^2 \log \gamma}{s-1},$$

which yields the logarithmic contribution in Λ.

Theorem 3 (Concentration of stash size)**.** *Let $S_\gamma = \sum_{i=1}^{\gamma} Y_i$. Then for every $\delta > 0$,*

$$\Pr(S_\gamma \ge (1+\delta)\mathbb{E}[S_\gamma]) \le \exp\left(-\frac{\delta^2 \mathbb{E}[S_\gamma]}{2(1+\delta/3)}\right).$$

Proof. Define $M_i = Y_i - \mathbb{E}[Y_i \mid \mathcal{F}_{i-1}]$. Then (M_i) is a martingale difference sequence and $|M_i| \le 1$. Furthermore,

$$\mathrm{Var}(M_i \mid \mathcal{F}_{i-1}) = \mathrm{Var}(Y_i \mid \mathcal{F}_{i-1}) \le \mathbb{E}[Y_i \mid \mathcal{F}_{i-1}].$$

Hence

$$\sum_{i=1}^{\gamma} \mathbb{E}[\mathrm{Var}(M_i \mid \mathcal{F}_{i-1})] \le \sum_{i=1}^{\gamma} \mathbb{E}[Y_i] = \mathbb{E}[S_\gamma].$$

Applying Freedman's inequality for martingale differences [Fre75] gives, for every $t > 0$,

$$\Pr(S_\gamma - \mathbb{E}[S_\gamma] \ge t) \le \exp\left(-\frac{t^2}{2(\mathbb{E}[S_\gamma] + t/3)}\right).$$

Substituting $t = \delta \mathbb{E}[S_\gamma]$ completes the proof. □

3.1.2 Interpreting Theorem 2

Let $\rho = \gamma/\mu$ denote the final load factor. The bound on Λ consists of two contributions: an overflow term and a revisit term. We ignore the term $\alpha(\alpha+1)\log\gamma/(s-1)$.

Overflow contribution At insertion i, the load factor is $\rho_i = (i-1)/\mu$. Each eviction step succeeds with probability at most ρ_i, implying $\Pr(L_i > \alpha) \le \rho_i^\alpha$. Therefore $\sum_{i=1}^{\gamma} \left(\frac{i-1}{\mu}\right)^\alpha \le \gamma\rho^\alpha = \mu\rho^{\alpha+1}$. Thus the total overflow contribution satisfies $O(\mu\rho^{\alpha+1})$.

Revisit contribution The revisit term is

$$R = \frac{\alpha}{s-1} \sum_{i=2}^{\gamma} \frac{\mu(H_\mu - H_{\mu-i+1}) - (i-1)}{i-1}.$$

Suppose that $\gamma = \rho\mu$ for $0 < \rho < 1$. Using the asymptotic expansion of the harmonic numbers, $H_n = \log n + \gamma_E + O(n^{-1})$, where $\gamma_E \approx .577$ denotes the Euler–Mascheroni constant, we obtain

$$H_\mu - H_{\mu-k} = \log\left(\frac{\mu}{\mu-k}\right) + O(\mu^{-1})$$

where $k \le \rho\mu$. Hence

$$\frac{\mu(H_\mu - H_{\mu-k}) - k}{k} = \frac{-\log(1-k/\mu) - k/\mu}{k/\mu} + O(\mu^{-1}).$$

Substituting $x = k/\mu$, and approximating the sum using an integral,

$$R = \frac{\alpha\mu}{s-1} \int_0^\rho \frac{-\log(1-x) - x}{x}\, dx + O(\alpha).$$

The integral can be expressed in terms of the dilogarithm,

$$\mathrm{Li}_2(x) := \sum_{m=1}^{\infty} \frac{x^m}{m^2}, \qquad |x| \le 1,$$

which satisfies

$$\frac{d}{dx}\mathrm{Li}_2(x) = -\frac{\log(1-x)}{x}.$$

Consequently,

$$\int_0^\rho \frac{-\log(1-x) - x}{x}\, dx = \mathrm{Li}_2(\rho) - \rho,$$

and therefore

$$R = \frac{\alpha\mu}{s-1}\left(\mathrm{Li}_2(\rho) - \rho\right) + O(\alpha).$$

The power-series expansion at $\rho = 0$ yields

$$\mathrm{Li}_2(\rho) = \sum_{m=1}^{\infty} \frac{\rho^m}{m^2} = \rho + \frac{\rho^2}{4} + \frac{\rho^3}{9} + \frac{\rho^4}{16} + O(\rho^5)$$

yields

$$R = \frac{\alpha\mu}{s-1}\left(\frac{\rho^2}{4} + \frac{\rho^3}{9} + \frac{\rho^4}{16} + O(\rho^5)\right).$$

Thus, when the power series expansion captures asymptotic behavior,

$$R = \frac{\alpha\mu\rho^2}{4(s-1)} + O\left(\frac{\alpha\mu\rho^3}{s-1}\right).$$

3.2 BFS Eviction Stash Size

The natural idea of BFS eviction is to add all the other positions of a key to a queue and pop the queue until an empty location is found or one has popped α total times. The first chain of items that ends in an empty location is shifted to those new locations. In our implementation, we only mark positions as visited if they are involved in a move. These are the positions that are visible to the server. Unfortunately, we don't know how to provide formal bounds in this model as edges of the Cuckoo graph can be followed multiple

times. Instead, we consider a (i, key) as invalid as soon as it is popped from the queue once. This allows us to recover the uniformity of the next item from the random walk analysis.

The main goal of the BFS analysis is bounding the probability that the queue is emptied before an empty slot is found. The queue can be emptied by popping an item that already had its other positions popped meaning no new items are added to the queue. We stress that the nature of the BFS process means that one cannot argue that the number of other positions popped by the next accessed items is proportional to the total number of items accessed in all previous insertions. Instead, we use a convexity argument in which we assume items have either accessed all s positions or only their current position. This increases the variance of the number of items added to the queue, thus increasing the probability that the queue can be emptied. Similar to the case of random walk, we can use Freedman [Fre75] to provide strong tail bounds on the stash size deviating from its expectation.

Theorem 4 (BFS stash-size bound after γ insertions)**.** *Consider an array of μ buckets and a sequence of γ insertions performed using BFS eviction with s candidate locations per item and search depth α. Let $S_\gamma = \sum_{i=1}^{\gamma} Y_i$, where Y_i indicates that insertion i enters the stash. Define $\rho_i = (i-1)/\mu, m_\alpha(\rho_i) = \sum_{i=1}^{\alpha} \rho_i$, and λ_i is the fraction of $(s-1)(i-1)$ states that have been exposed before insertion i. Then*

$$\Lambda_{\mathrm{BFS}} = \sum_{i=1}^{\gamma} \left(\frac{i-1}{\mu}\right)^{\alpha} + \sum_{i=1}^{\gamma} q_i^s,$$

where q_i is the smallest solution of

$$q = (1-\lambda_i)q^{s-1} + \lambda_i$$

and

$$\begin{aligned} \mathbb{E}[\lambda_i] &\le \frac{m_\alpha(\rho_i) - 1}{s-1} + \frac{\alpha - s + 1}{(s-1)(i-1)}, \\ \mathbb{E}[S_\gamma] &\le \Lambda_{\mathrm{BFS}}, \\ \Pr[S_\gamma \ge (1+\delta)\Lambda_{\mathrm{BFS}}] &\le \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{\Lambda_{\mathrm{BFS}}}, \quad \forall \delta > 0. \end{aligned}$$

Looking ahead, we show a natural interpretation of these parameters (assuming accuracy of the power series approximation) it suffices for stash size on the order of

$$\mu\left(\left(\frac{\rho}{2(s-1)}\right)^s + \rho^\alpha\right).$$

3.2.1 BFS Stash Size Analysis

Algorithm 1 describes the BFS eviction procedure. This algorithm only sketches how to compute the items to be moved, denoted as $\mathsf{MoveChain}$ to compute. To retain obliviousness, the $\mathsf{MoveChain}$ of length must be padded to be α as in the random walk version. In this section, we assume that pos are sampled at insertion time rather than being specified by the relevant random functions. A state is a pair $(\mathsf{key}, \mathsf{pos})$ consisting of an item and one of its candidate positions. We ignore val to be inserted for the purposes of analyzing the algorithm.

To avoid repeatedly exploring the same portion of the cuckoo graph, a state is considered for moving at most once over the lifetime of the data structure. The global set of previously considered states is denoted by V for visited. By abuse of notation, L_i denotes the number of positions explored during the i-th BFS insertion; this generalizes the walk length used in the random-walk analysis.

For insertion i, define the following random variables and notation:

1. L_i the number of states popped from the BFS queue during insertion i.
2. $V_i = \{(\mathsf{key}, \mathsf{pos}) | (\mathsf{key}, \mathsf{pos})$ popped during first i insertions$\}$. Let $v_i = |V_i|$. Since every popped state is added to V exactly once, $v_i = \sum_{j=1}^{i} L_j$.

Input: table $(\mathsf{Contents}[1], \ldots, \mathsf{Contents}[\mu])$, global explored-state set V, item $(\mathsf{key}, \mathsf{val})$
Output: updated table state or stash insertion

Sample $\mathsf{pos}_{\mathsf{key},1}, \ldots, \mathsf{pos}_{\mathsf{key},s} \xleftarrow{\$} [1, \mu]$.
$\mathsf{MoveChain} \leftarrow \perp$;
$\mathsf{Queue} \leftarrow \emptyset$;
for $j = 1, \ldots, s$ **do**
  Put $(\mathsf{key}, \mathsf{pos}_{\mathsf{key},j}, [(\mathsf{key}, \mathsf{pos}_{\mathsf{key},j})])$ in Queue;
end
$\mathsf{steps} \leftarrow 0$;
while $\mathsf{Queue} \neq \emptyset$ ***and*** $\mathsf{steps} < \alpha$ **do**
  $(\mathsf{key}, \mathsf{pos}, \mathsf{Chain}) \leftarrow \mathsf{Queue.pop}()$;
  if $(\mathsf{key}, \mathsf{pos}) \in V$ **then**
    continue;
  end
  $V \leftarrow V \cup \{(\mathsf{key}, \mathsf{pos})\}$;
  if $\mathsf{Contents}[\mathsf{pos}] = \perp$ **then**
    $\mathsf{MoveChain} \leftarrow \mathsf{Chain}$;
    break;
  end
  $\mathsf{key}' \leftarrow \mathsf{Contents}[\mathsf{pos}]$;
  for $j = 1, \ldots, s$ **do**
    $\mathsf{Chain}' \leftarrow \mathsf{Chain} || [(\mathsf{key}', \mathsf{pos}_{\mathsf{key}',j})]$;
    enqueue $(\mathsf{key}', \mathsf{pos}_{\mathsf{key}',j}, \mathsf{Chain}')$ into Queue;
  end
  $\mathsf{steps} \leftarrow \mathsf{steps} + 1$;
end
if $\mathsf{MoveChain} \neq \perp$ **then**
  while $\mathsf{MoveChain} \neq \emptyset$ **do**
    $(\mathsf{key}, \mathsf{pos}) \leftarrow \mathsf{MoveChain}$
    $\mathsf{Contents}[\mathsf{pos}] = \mathsf{key}$
  end
end
else
  place key in Stash;
end

Algorithm 1: Core of `UOKVS.Insert`$(K, \mathsf{key}, \mathsf{val})$ using BFS eviction. V is initialized to empty during Encode.

3. Let $\rho_i = (i-1)/\mu$, denote the load before insertion i. As with Theorem 2 we proceed under the assumption that all items are stored in the array at all times.
4. For insertion i,
   (a) Let Q_t denote the BFS queue size before the t-th queue pop.
   (b) Let W_t denote the new states popped (to be added to V_i) before time t, define $w_t = |W_t|$
   (c) Let Z_t denote the number of newly enqueued states produced by the t-th pop.
5. Define D_i, an indicator random variable to be 1 if and only if the BFS queue becomes empty before an empty position is found or α items are popped from the queue.
6. $\mathcal{F}_i = \sigma$(all hash choices, queue explorations, and table states after insertion i) be the natural filtration of the history of the algorithm.

7. Define Y_i as the indicator random variable that insertion i enters the stash. Define $S_\gamma = \sum_{i=1}^{\gamma} Y_i$, the total number of stash insertions among the first γ insertions.

Theorem 5 (BFS stash bound)**.** *Fix i and condition on $\mathcal{F}_{i-1}$. Define*

$$\lambda_i = \frac{v_{i-1} + \alpha - s - (i-1)}{(s-1)(i-1)}.$$

Assume $0 \le \lambda_i < 1$. Let $q_i \in [0,1]$ denote the smallest solution of $q_i = (1-\lambda_i) q_i^{s-1} + \lambda_i$. Then

$$\Pr(D_i = 1 \mid \mathcal{F}_{i-1}) \le q_i^{\,s},$$

$$\Pr(Y_i = 1 \mid \mathcal{F}_{i-1}) \le \rho_i^{\alpha} + q_i^{\,s},$$

$$\mathbb{E}[S_\gamma] \le \sum_{i=1}^{\gamma} \left(\rho_i^{\alpha} + \mathbb{E}[q_i^{\,s}]\right).$$

Proof. We consider the bound on insertion i. As a reminder, for the purposes of analyzing the stash for insertion i, we assume that all items are in the array. Moving an item permanently to the stash can only reduce walk lengths (it also only removes items from V_i). Let $\mathcal{G}_t = \sigma(\mathcal{F}_{i-1}, W_t, Q_t)$ denote the filtration available immediately before the t-th queue pop. When considered the t-th queue pop, we additionally condition on $\mathcal{G}_t$. Conditioning on $V_{i-1} \cup W_t$ fixes the set of of previously revealed states. The s root states of the newly inserted item are sampled freshly and therefore do not belong to V_{i-1}.

Since V_{i-1} contains v_{i-1} states and W_t contains w_t newly revealed states, of which s correspond to the new item and are guaranteed to be fresh, the total number of revealed states belonging to previously inserted keys is $v_{i-1} + w_t - s$. Define

$$R_t(\mathsf{key}) = \#\Big\{(\mathsf{key}, \mathsf{pos}_{\mathsf{key},j}) \in V_{i-1} \cup W_t : j = 1, \ldots, s\Big\},$$

the number of states of key key that have been revealed by time t. Since every revealed state contributes to exactly one key at time t of insertion i,

$$\sum_{\mathsf{key}} R_t(\mathsf{key}) = v_{i-1} + w_t - s.$$

Let key' denote the key displaced by the state currently being explored. By exchangeability of the previously inserted keys which we denote as $\mathsf{key}_1, ..., \mathsf{key}_{i-1}$ (and since each $\mathsf{key}, \mathsf{pos}$ is only followed once),

$$\Pr[\mathsf{key}' = \mathsf{key}_i \mid \mathcal{G}_t] = \frac{1}{i-1}, \forall \mathsf{key}_1 = 1, \ldots, \mathsf{key}_{i-1}.$$

A state is enqueued only when it has not previously been revealed. Therefore, among the s states of key', exactly $R_t(\mathsf{key}')$ have already been revealed and exactly $s - R_t(\mathsf{key}')$ remain fresh. Consequently, $Z_t = s - R_t(\mathsf{key}')$. Hence

$$\mathbb{E}[x^{Z_t} \mid \mathcal{G}_t] = \frac{1}{i-1} \sum_{k=1}^{i-1} x^{\,s - R_t(\mathsf{key}_k)}.$$

Define

$$m_t = \frac{1}{i-1} \sum_{k=1}^{i-1} R_t(\mathsf{key}_k) = \frac{v_{i-1} + w_t - s}{i-1}.$$

The natural interpretation of m_t is the average number of revealed states for key at the time of the t-th pop from the queue. For each fixed $x \in [0,1]$, let define the function $g_x(r) = x^{\,s-r}$. The function g_x is convex on $r \in [1, s]$ because $g_x''(r) = (\log x)^2 x^{\,s-r} \ge 0$. This means that among all collections $R_t(\mathsf{key}_1), \ldots, R_t(\mathsf{key}_{i-1}) \in [1, s]$ having average m_t, the average value of g is maximized when the mass is concentrated at the endpoints 1 and s. Thus, we proceed as though a fraction $\lambda_t = (m_t - 1)/(s-1)$ of the keys have $R_t(\mathsf{key}) = s$, while the remaining fraction $1 - \lambda_t$ have $R_t(\mathsf{key}) = 1$. Therefore

$$\mathbb{E}[x^{Z_t} \mid \mathcal{G}_t] \le (1-\lambda_t) x^{s-1} + \lambda_t.$$

Substituting the expression for λ_t,

$$\mathbb{E}[x^{Z_t} \mid \mathcal{G}_t] \leq \left(1 - \frac{m_t - 1}{s-1}\right) x^{s-1} + \frac{m_t - 1}{s-1}.$$

Since at most α states can be added to V_i during a single insertion, $w_t \leq \alpha$. Thus,

$$m_t \leq \frac{v_{i-1} + \alpha - s}{i-1}.$$

Define

$$\lambda_i = \frac{v_{i-1} + \alpha - s - (i-1)}{(i-1)(s-1)}.$$

Then $\mathbb{E}[x^{Z_t} \mid \mathcal{G}_t] \leq (1-\lambda_i)x^{s-1} + \lambda_i$. Let $q_i \in [0,1]$ denote the smallest solution of

$$q_i = (1-\lambda_i)q_i^{s-1} + \lambda_i.$$

Substituting $x = q_i$ yields $\mathbb{E}[q_i^{Z_t} \mid \mathcal{G}_t] \leq q_i$. Each queue pop removes one state from the queue and generates Z_t new candidate states. Consequently, $Q_{t+1} = Q_t - 1 + Z_t$. Define $M_t = q_i^{Q_t}$. Then

$$M_{t+1} = q_i^{Q_t - 1 + Z_t} = q_i^{Q_t - 1} q_i^{Z_t}.$$

Taking conditional expectations and using the preceding inequality,

$$\mathbb{E}[M_{t+1} \mid \mathcal{G}_t] = q_i^{Q_t - 1}\mathbb{E}[q_i^{Z_t} \mid \mathcal{G}_t] \leq q_i^{Q_t} = M_t.$$

Thus (M_t) is a supermartingale. To bound the probability of queue exhaustion, define $\tau = \min\{t : Q_t = 0\}$. The BFS traversal terminates at the stopping time $\tau \wedge \alpha$. We now analyze the probability that $\tau \leq \alpha$. By Doob's optional sampling theorem [Wil91]:

$$\mathbb{E}[M_{\min_{\tau,\alpha}} \mid \mathcal{F}_{i-1}] \leq M_0.$$

Since $Q_0 = s$, $M_0 = q_i^s$. On the extinction event D_i, $Q_\tau = 0$, and therefore $M_\tau = q_i^{Q_\tau} = 1$. Hence $\mathbf{1}_{D_i} \leq M_\tau$. Taking conditional expectations and applying optional stopping gives

$$\begin{aligned}\Pr(D_i = 1 \mid \mathcal{F}_{i-1}) &= \mathbb{E}[\mathbf{1}_{D_i} \mid \mathcal{F}_{i-1}] \\ &\leq \mathbb{E}[M_\tau \mid \mathcal{F}_{i-1}] \\ &\leq M_0 = q_i^s.\end{aligned}$$

A stash insertion can occur only if either

1. every examined position is occupied, or
2. the BFS queue becomes extinct.

The first event has probability at most ρ_i^α. This fact relies on the fact that each edge is followed at most once so the next key considered at each step is independent of the conditioned history. Taking expectations and summing over $i = 1, \ldots, \gamma$,

$$\mathbb{E}[S_\gamma] = \sum_{i=1}^{\gamma} \mathbb{E}[Y_i] \leq \sum_{i=1}^{\gamma} \left(\rho_i^\alpha + \mathbb{E}[q_i^s]\right).$$

This proves the theorem. □

3.2.2 Concentration Bounds for BFS Stash

Lemma 3 (Concentration of the explored-state process)**.** *Assume $\rho_i \le 1-\varepsilon$ for every insertion. Define $C_{\alpha,\varepsilon} = \sum_{t=1}^{\alpha}(2t-1)(1-\varepsilon)^{t-1}$, and*

$$m_i = \sum_{j=1}^{i} \frac{1-\rho_j^{\alpha}}{1-\rho_j}.$$

Then for every $\delta \in (0,1)$,

$$v_i \le m_i + 2\sqrt{C_{\alpha,\varepsilon}\, i \log\frac{1}{\delta}} + 2\alpha \log\frac{1}{\delta}$$

with probability at least $1-\delta$. Furthermore, this bound applies simultaneously to all $i \le \gamma$,

$$\Pr\left(\exists i \le \gamma : v_i > m_i + 2\sqrt{C_{\alpha,\varepsilon}\, i \log\frac{\gamma}{\delta}} + 2\alpha \log\frac{\gamma}{\delta}\right) \le \delta.$$

Proof. For the purposes of this lemma we ignore queue exhaustion. Queue exhaustion can only shorten a BFS exploration and therefore can only decrease L_i. For every $t \ge 1$, the event $\{L_i \ge t\}$ implies that the first $t-1$ explored positions are occupied.

Conditioning on $\mathcal{F}_{i-1}$, the load before insertion i is ρ_i. Hence $\Pr(L_i \ge t \mid \mathcal{F}_{i-1}) \le \rho_i^{t-1}$. Using the tail-sum formula,

$$\begin{aligned}\mathbb{E}[L_i \mid \mathcal{F}_{i-1}] &= \sum_{t=1}^{\alpha} \Pr(L_i \ge t \mid \mathcal{F}_{i-1}) \\ &\le \sum_{t=1}^{\alpha} \rho_i^{t-1} = \frac{1-\rho_i^{\alpha}}{1-\rho_i}.\end{aligned}$$

Define $M_i = v_i - \sum_{j=1}^{i} \mathbb{E}[L_j \mid \mathcal{F}_{j-1}]$. Since $v_i - v_{i-1} = L_i$, the martingale increments are

$$\Delta_i = M_i - M_{i-1} = L_i - \mathbb{E}[L_i \mid \mathcal{F}_{i-1}].$$

Because $0 \le L_i \le \alpha$, $|\Delta_i| \le \alpha$. Let $V_i = \sum_{j=1}^{i} \mathrm{Var}(L_j \mid \mathcal{F}_{j-1})$ denote the predictable quadratic variation. Since $\rho_j \le 1-\varepsilon$, the previous tail bound gives $\Pr(L_j \ge t \mid \mathcal{F}_{j-1}) \le (1-\varepsilon)^{t-1}$. Since L_j is nonnegative and integer valued, its square is represented by sums of odd integers:

$$L_j^2 = \sum_{t=1}^{L_j}(2t-1) = \sum_{t=1}^{\alpha}(2t-1)\mathbf{1}[L_j \ge t].$$

Taking conditional expectations,

$$\begin{aligned}\mathbb{E}[L_j^2 \mid \mathcal{F}_{j-1}] &= \sum_{t=1}^{\alpha}(2t-1)\Pr(L_j \ge t \mid \mathcal{F}_{j-1}) \\ &\le \sum_{t=1}^{\alpha}(2t-1)(1-\varepsilon)^{t-1} \\ &= C_{\alpha,\varepsilon}.\end{aligned}$$

Therefore $\mathrm{Var}(L_j \mid \mathcal{F}_{j-1}) \le C_{\alpha,\varepsilon}$, and hence $V_i \le C_{\alpha,\varepsilon} i$. Applying Freedman's inequality to the martingale (M_i),

$$\Pr(M_i \ge t) \le \exp\left(-\frac{t^2}{2(V_i + \alpha t)}\right).$$

Using $V_i \le C_{\alpha,\varepsilon} i$ and choosing $t = 2\sqrt{C_{\alpha,\varepsilon}\, i \log \frac{1}{\delta}} + 2\alpha \log \frac{1}{\delta}$, gives $\Pr(M_i \ge t) \le \delta$. Consequently, with probability at least $1-\delta$,

$$v_i \le \sum_{j=1}^{i} \mathbb{E}[L_j \mid \mathcal{F}_{j-1}] + 2\sqrt{C_{\alpha,\varepsilon}\, i \log \frac{1}{\delta}} + 2\alpha \log \frac{1}{\delta}.$$

Finally, $\sum_{j=1}^{i} \mathbb{E}[L_j \mid \mathcal{F}_{j-1}] \le m_i$, which proves the theorem. □

Remark: The above bound considers the worst bound on the variance of L_j ignoring the fact that load remains lower until insertion $\rho_i = 1-\epsilon$, this has the effect of that smaller v_i have both smaller m_i and less variance. We keep the above bound as it allows us to simply state the simultaneous bound.

Lemma 4 (Cumulative extinction approximation)**.** *Assume the hypotheses of Lemma 3 and let $s \ge 3$. For $x \in \mathbb{R}$, write $[x]_{[0,1]} := \min\{1, \max\{0, x\}\}$. For $i \ge 2$, define the deterministic exposure parameter*

$$\widehat{\lambda}_i := \left[\frac{m_{i-1} + \alpha - s - (i-1)}{(s-1)(i-1)} \right]_{[0,1]},$$

and let $\widehat{q}_i$ denote the smallest solution in $[0,1]$ of

$$q = (1-\widehat{\lambda}_i) q^{s-1} + \widehat{\lambda}_i.$$

Use the same truncation in the definition of λ_i, and let q_i denote the smallest solution associated with λ_i. Set $q_1 = \widehat{q}_1 = 0$. Let $C_{\alpha,\varepsilon}$ be as in Lemma 3. Then, for every $\delta \in (0,1)$,

$$\sum_{i=1}^{\gamma} q_i^s \le \sum_{i=1}^{\gamma} \widehat{q}_i^s + O_s\left(\sqrt{C_{\alpha,\varepsilon}\, \gamma \log(\gamma/\delta)} + \alpha \log(\gamma/\delta) \log \gamma \right)$$

with probability at least $1-\delta$. Here and below, $O_s(\cdot)$ hides constants depending only on s.

Proof. Let $L := \log(\gamma/\delta)$. By the simultaneous conclusion of Lemma 3, with probability at least $1-\delta$, for every $i \le \gamma$, $v_i \le m_i + 2\sqrt{C_{\alpha,\varepsilon} i L} + 2\alpha L$. Condition on this event. For every $i \ge 2$, monotonicity and 1-Lipschitz continuity of truncation to $[0,1]$ give

$$\lambda_i \le \widehat{\lambda}_i + O_s\left(\sqrt{\frac{C_{\alpha,\varepsilon} L}{i-1}} + \frac{\alpha L}{i-1} \right).$$

For $\lambda \in [0,1]$, let $q(\lambda)$ be the smallest solution in $[0,1]$ of $q = (1-\lambda)q^{s-1} + \lambda$, and define $f(\lambda) := q(\lambda)^s$. The function f is nondecreasing. Moreover, for $s \ge 3$, it is Lipschitz continuous on $[0,1]$. To see this, let $\lambda_c := 1 - 1/(s-1)$. For $\lambda < \lambda_c$, the smallest solution satisfies $q(\lambda) < 1$ and can equivalently be parameterized by

$$\lambda = \phi(q) := \frac{q - q^{s-1}}{1 - q^{s-1}}.$$

Direct differentiation shows that $\phi'(q)$ extends continuously and remains strictly positive on $[0,1]$; in particular,

$$\lim_{q \to 1} \phi'(q) = \frac{s-2}{2(s-1)} > 0.$$

Thus the inverse $q(\lambda)$ is Lipschitz on $[0, \lambda_c]$. For $\lambda \ge \lambda_c$, the smallest solution is $q(\lambda) = 1$. Hence q, and therefore f, is Lipschitz on all of $[0,1]$. Let L_s be a corresponding Lipschitz constant. Since $q_i^s = f(\lambda_i)$ and $\widehat{q}_i^s = f(\widehat{\lambda}_i)$, monotonicity and Lipschitz continuity imply

$$q_i^s \le \widehat{q}_i^s + O_s\left(\sqrt{\frac{C_{\alpha,\varepsilon} L}{i-1}} + \frac{\alpha L}{i-1} \right).$$

Summing over $i = 2, \dots, \gamma$ and using

$$\sum_{i=2}^{\gamma} (i-1)^{-1/2} = O(\sqrt{\gamma}) \qquad \text{and} \qquad \sum_{i=2}^{\gamma} (i-1)^{-1} = O(\log \gamma)$$

gives

$$\sum_{i=1}^{\gamma} q_i^s \le \sum_{i=1}^{\gamma} \widehat{q}_i^s + O_s\left(\sqrt{C_{\alpha,\varepsilon}\, \gamma \log(\gamma/\delta)} + \alpha \log(\gamma/\delta) \log \gamma \right),$$

as claimed. □

Theorem 6 (Concentration of stash size)**.** *Define* $\widehat{\sigma}_\gamma := \sum_{i=1}^{\gamma} (\rho_i^\alpha + \widehat{q}_i^s)$ *and* $S_\gamma := \sum_{i=1}^{\gamma} Y_i$*. Let* $c_s > 0$ *be a constant for which the conclusion of Lemma 4 holds, and define*

$$\mathcal{E}_\gamma(\delta) := c_s \left(\sqrt{C_{\alpha,\varepsilon}\, \gamma \log \frac{2\gamma}{\delta}} + \alpha \log \frac{2\gamma}{\delta} \log \gamma \right)$$

and $B_\gamma(\delta) := \widehat{\sigma}_\gamma + \mathcal{E}_\gamma(\delta)$*. Then, for every* $\delta \in (0,1)$*,*

$$S_\gamma \le B_\gamma(\delta) + 2\sqrt{B_\gamma(\delta) \log \frac{2}{\delta}} + 2 \log \frac{2}{\delta}$$

with probability at least $1 - \delta$*.*

Proof. Define $p_i := \Pr(Y_i = 1 \mid \mathcal{F}_{i-1})$ and $P_\gamma := \sum_{i=1}^{\gamma} p_i$. By Theorem 5, $p_i \le \rho_i^\alpha + q_i^s$, and hence

$$P_\gamma \le \sum_{i=1}^{\gamma} (\rho_i^\alpha + q_i^s).$$

Applying Lemma 4 with failure probability $\delta/2$ gives

$$P_\gamma \le \widehat{\sigma}_\gamma + \mathcal{E}_\gamma(\delta) = B_\gamma(\delta)$$

with probability at least $1 - \delta/2$. Now define the martingale

$$M_t := \sum_{i=1}^{t} (Y_i - p_i).$$

Its increments are bounded by 1, and $\mathrm{Var}(Y_i \mid \mathcal{F}_{i-1}) \le p_i$. Applying Freedman's inequality with failure probability $\delta/2$ gives

$$S_\gamma \le P_\gamma + 2\sqrt{P_\gamma \log \frac{2}{\delta}} + 2 \log \frac{2}{\delta}$$

with probability at least $1 - \delta/2$.

On the intersection of these two events, the monotonicity of $x \longmapsto x + 2\sqrt{x \log(2/\delta)}$ gives

$$S_\gamma \le B_\gamma(\delta) + 2\sqrt{B_\gamma(\delta) \log \frac{2}{\delta}} + 2 \log \frac{2}{\delta}.$$

A union bound shows that this intersection has probability at least $1 - \delta$. □

The preceding results provide a deterministic upper envelope for the stash size. Lemma 3 bounds the explored-state process from above, and Lemma 4 converts this into the deterministic extinction estimate $\sum_i \widehat{q}_i^s$. Consequently, with probability at least $1 - \delta$, the stash size exceeds $\widehat{\sigma}_\gamma$ by at most

$$\mathcal{E}_\gamma(\delta) + 2\sqrt{\big(\widehat{\sigma}_\gamma + \mathcal{E}_\gamma(\delta)\big) \log \frac{2}{\delta}} + 2 \log \frac{2}{\delta}.$$

When the load is bounded away from one, α is polylogarithmic in γ, and $\widehat{\sigma}_\gamma = \Theta(\gamma)$, this excess is

$$O_s\left(\sqrt{\gamma \log(\gamma/\delta)} + \alpha \log(\gamma/\delta) \log \gamma \right).$$

3.2.3 Interpretation for $s = 4$

Lemma 3 shows that the revealed-state density is tightly concentrated around its expectation. In particular,

$$\frac{v_i}{i} \le m(\rho_i) + O\left(\sqrt{\frac{\log(1/\delta)}{i}}\right)$$

with high probability. The concentration results of show that the cumulative extinction contribution can be approximated using the quantity

$$\widehat{\lambda}_i = \frac{m_{i-1} + \alpha - s - (i-1)}{(s-1)(i-1)}.$$

Since

$$\frac{m_{i-1}}{i-1} = m(\rho_i) + O\left(\frac{1}{i}\right),$$

we obtain

$$\widehat{\lambda}_i = \frac{m(\rho_i) - 1}{s-1} + O\left(\frac{1}{i}\right).$$

Combining this approximation with Lemma 3 gives

$$\lambda_i \le \frac{m(\rho_i) - 1}{s-1} + O\left(\sqrt{\frac{\log(1/\delta)}{i}}\right) + O\left(\frac{1}{i}\right).$$

Motivated by this approximation, define

$$\widehat{\lambda}(\rho) = \frac{m(\rho) - 1}{s-1}.$$

For $s = 4$,

$$\widehat{\lambda}(\rho) = \frac{m(\rho) - 1}{3}.$$

We evaluate this approximation directly. For $s = 4$,

$$\widehat{\lambda}(\rho) = \frac{m_\alpha(\rho) - 1}{3}, \qquad m_\alpha(\rho) = \sum_{r=0}^{\alpha-1} \frac{\rho^r}{r+1},$$

and $\widehat{q}(\rho)$ is the smallest solution of $q = (1 - \widehat{\lambda}(\rho))q^3 + \widehat{\lambda}(\rho)$.

3.2.4 Power series approximation

The fixed-point expansion below is valid whenever

$$\widehat{\lambda}(\rho) = \frac{m(\rho) - 1}{s-1} \ll 1.$$

Equivalently, $m(\rho) - 1 \ll s - 1$. For $s = 4$, this condition holds throughout nearly the entire load range. For example, $m(0.8) \approx 2.01$, which gives $\widehat{\lambda}(0.8) \approx 0.34$. The parameter $\widehat{\lambda}(\rho)$ does not approach its critical value of $3/4$ until the load is extremely high, for $\alpha \le 20$ this doesn't occur until ($\rho \approx 0.95$). Thus the expansion remains accurate for all but near-full occupancy. Writing $q = \lambda + a\lambda^2 + b\lambda^3 + \cdots$ and substituting into $q = (1-\lambda)q^{s-1} + \lambda$ yields

$$\lambda + a\lambda^2 + b\lambda^3 + \cdots = \lambda + (1-\lambda)\Big(\lambda + a\lambda^2 + b\lambda^3 + \cdots\Big)^{s-1}.$$

Expanding the right-hand side gives

$$q^{s-1} = \lambda^{s-1} + (s-1)a\,\lambda^s + O(\lambda^{s+1}),$$

and therefore $q = \lambda + \lambda^{s-1} + O(\lambda^s)$. Consequently, $q - \lambda = \lambda^{s-1} + O(\lambda^s)$, so that

$$\begin{aligned}
\widehat{q}(\rho) &= \widehat{\lambda}(\rho) + \widehat{\lambda}(\rho)^{\,s-1} + O\big(\widehat{\lambda}(\rho)^s\big), \\
\widehat{q}(\rho)^s &= \Big(\widehat{\lambda}(\rho) + \widehat{\lambda}(\rho)^{\,s-1} + O(\widehat{\lambda}(\rho)^s)\Big)^s \\
&= \widehat{\lambda}(\rho)^s + s\,\widehat{\lambda}(\rho)^{\,2s-2} + O\big(\widehat{\lambda}(\rho)^{\,2s-1}\big) \\
&= \widehat{\lambda}(\rho)^s + O\big(\widehat{\lambda}(\rho)^{\,2s-2}\big) \\
&= \frac{\big(m(\rho)-1\big)^s}{(s-1)^s} + O\Big(\big(m(\rho)-1\big)^{2s-2}\Big).
\end{aligned}$$

where the last step substitutes $\widehat{\lambda}(\rho) = (m(\rho)-1)/(s-1)$. Using the power series,

$$\begin{aligned}
m(\rho) &= \sum_{r=0}^{\alpha-1} \frac{\rho^r}{r+1} \\
&= 1 + \frac{\rho}{2} + \frac{\rho^2}{3} + O(\rho^3).
\end{aligned}$$

Therefore $\widehat{\lambda}(\rho) = \rho/(2(s-1)) + O(\rho^2)$, and hence

$$\widehat{q}(\rho)^s = \left(\frac{\rho}{2(s-1)}\right)^s + O(\rho^{s+1}).$$

4 MKSE from UOKVS

In this section, we use UOKVS to construct a Multi-Key Searchable Encryption (MKSE) scheme without replicating authorized documents across different clients. A secure MKSE is defined as follows.

Definition 4. *(Multi-Key Searchable Encryption) A tuple* $(\mathsf{KeyGen}^{\mathtt{Ow}}, \mathsf{Setup}, \mathsf{Search}, \mathsf{Share})$ *of PPT algorithms is a Multi-Key Searchable Encryption Scheme (*MKSE*) if the following holds:*
$\mathsf{KOwn}^{\mathtt{Ow}} \leftarrow \mathsf{KeyGen}^{\mathtt{Ow}}(1^\lambda)$*: Takes as input the security parameter* 1^λ *and returns a data key* $\mathsf{KOwn}^{\mathtt{Ow}}$*.*
$\begin{pmatrix} \mathsf{KOwn}^{\mathtt{Ow}} \\ \mathtt{EMM} \end{pmatrix} \leftarrow \mathsf{Setup}^{\mathtt{Ow},\mathtt{S}} \begin{pmatrix} \mathsf{KOwn}^{\mathtt{Ow}}, \mathtt{MM} \\ \perp \end{pmatrix}$*: Create an encrypted structure stored on the server.* KOwn *is updated to include the necessary state.*
$\begin{pmatrix} \perp \\ K^{\mathtt{C}} \end{pmatrix} \leftarrow \mathsf{Share}^{\mathtt{Ow},\mathtt{C}} \begin{pmatrix} \mathsf{KOwn}^{\mathtt{Ow}}, \mathtt{Plcy} \\ K^{\mathtt{C}}, \mathsf{key} \end{pmatrix}$*:* C *requests a list of keywords* key*. We assume that* Plcy *is a function of* key *and has any information needed about the encrypted multimap to make policy decisions. The value of* $K^{\mathtt{C}}$ *is updated. On first call,* $K^{\mathtt{C}} = \perp$*.*
$\begin{pmatrix} \perp \\ \mathsf{Res} \end{pmatrix} \leftarrow \mathsf{Search}^{\mathtt{S},\mathtt{C}} \begin{pmatrix} \mathtt{EMM} \\ K^{\mathtt{C}}, \mathsf{key} \end{pmatrix}$*:* C *searches for keyword* key *and receives a response,* Res*.*

Correctness *Let* λ *be a parameter. Consider the correctness experiment shown in Figure 8. The* $\mathsf{MKSE} := \mathsf{MKSE}(\lambda)$ *is correct if for all PPT adversaries* $\mathcal{A}$ *and all* $q := q(\lambda) = \mathtt{poly}(\lambda)$*, it is true that*

$$\Pr[\mathsf{Exp}_{\mathtt{Correct},\mathcal{A},\mathsf{MKSE}}(1^\lambda, q)] \geq 1 - \mathtt{ngl}(\lambda).$$

Security *Let* λ *be a security parameter. Consider the security experiment shown in Figure 9. The* $\mathsf{MKSE} := \mathsf{MKSE}(\lambda)$ *is secure if for all PPT adversaries* $\mathcal{A}$ *there exists a PPT* Sim *such that for all* $q := q(\lambda) = \mathtt{poly}(\lambda)$ *it is true that*

$$\left| \begin{aligned} &\Pr[\mathsf{Exp}_{\mathsf{Sec},\mathcal{A},\mathsf{MKSE}}(1^\lambda, q) = 1] \\ -&\Pr[\mathsf{Exp}_{\mathcal{A},\mathsf{Sim},\mathsf{MKSE}}(1^\lambda, q) = 1] \end{aligned} \right| \leq \mathtt{ngl}(\lambda).$$

$\underline{\mathbf{Exp}_{\mathtt{Correct},\mathcal{A},\mathsf{MKSE}}(1^\lambda, q)}$

$\mathcal{A}$ sends $\mathtt{Clients} = \{\mathtt{C}^j\}_{j=1}^m, \mathtt{Owners} = \{\mathtt{Ow}^i\}_{i=1}^n, \{\mathtt{MM}^{\mathtt{Ow}}\}_{\mathtt{Ow}\in\mathtt{Owners}}$, together with, for every $(\mathtt{Ow},\mathtt{C})$, a length $q^{\mathtt{Ow},\mathtt{C}} \le q$ and a sequence $\left\{(\mathsf{key}_i^{\mathtt{Ow},\mathtt{C}}, \mathtt{Plcy}_i^{\mathtt{Ow},\mathtt{C}}, \mathtt{Shr}_i^{\mathtt{Ow},\mathtt{C}})\right\}_{i=1}^{q^{\mathtt{Ow},\mathtt{C}}}$.

$\mathcal{C}$ performs:

1. For every $\mathtt{Ow} \in \mathtt{Owners}$, run $\mathsf{KOwn}^{\mathtt{Ow}} \leftarrow \mathsf{KeyGen}^{\mathtt{Ow}}(1^\lambda)$,
$$\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}\\ \mathtt{EMM}^{\mathtt{Ow}}\end{pmatrix} \leftarrow \mathsf{Setup}^{\mathtt{Ow},\mathtt{S}}\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}, \mathtt{MM}^{\mathtt{Ow}}\\ \perp\end{pmatrix}.$$
2. For every $(\mathtt{Ow},\mathtt{C})$, initialize $K^{\mathtt{C}} = \perp$. For $1 \le i \le q^{\mathtt{Ow},\mathtt{C}}$, run
$$\begin{pmatrix}\perp\\ K^{\mathtt{C}}\end{pmatrix} \leftarrow \mathsf{Share}^{\mathtt{Ow},\mathtt{C}}\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}, \mathtt{Plcy}_i^{\mathtt{Ow},\mathtt{C}}\\ K^{\mathtt{C}}, \mathtt{Shr}_i^{\mathtt{Ow},\mathtt{C}}\end{pmatrix},$$
followed by
$$\begin{pmatrix}\perp\\ \mathsf{Res}_i^{\mathtt{Ow},\mathtt{C}}\end{pmatrix} \leftarrow \mathsf{Search}^{\mathtt{S},\mathtt{C}}\begin{pmatrix}\mathtt{EMM}^{\mathtt{Ow}}\\ K^{\mathtt{C}}, \mathsf{key}_i^{\mathtt{Ow},\mathtt{C}}\end{pmatrix}.$$

Output 1 iff, for every $(\mathtt{Ow},\mathtt{C})$ and every $1 \le i \le q^{\mathtt{Ow},\mathtt{C}}$,
$$\mathsf{Res}_i^{\mathtt{Ow},\mathtt{C}} = \begin{cases}\mathtt{MM}^{\mathtt{Ow}}[\mathsf{key}_i^{\mathtt{Ow},\mathtt{C}}] & \exists j \le i : \mathsf{key}_i^{\mathtt{Ow},\mathtt{C}} \in \mathtt{Shr}_j^{\mathtt{Ow},\mathtt{C}} \\ & \quad \wedge \mathtt{Plcy}_j^{\mathtt{Ow},\mathtt{C}}(\mathtt{Shr}_j^{\mathtt{Ow},\mathtt{C}}) = 1,\\ \perp & \text{otherwise.}\end{cases}$$

Figure 8: Correctness experiment for MKSE, where q bounds the number of share and search operations for each owner–client pair.

Discussion The definition differs from previous definitions. Unlike Hamlin et al. [HSWW18], we use simulation security. We consider arbitrary corruption of parties including owners, clients, and the server. We don't model the adversary providing improperly distributed keys to the algorithms but our algorithms work in this setting.

We provide an overview of how MKSE schemes are usually constructed. For input MM between key and $\vec{\mathsf{Doc}}$, one creates three maps that we call VolMap, ValMap and DocMap. VolMap maps each keyword key to the number of documents containing that keyword. ValMap maps each (key, i) to each document id (Id) that contain key. DocMap maps each Id to the Doc.

Construction 1 (MKSE from VolMap, ValMap, and DocMap)**.** *Let* $\mathtt{UOKVS}_{\mathtt{VolMap}}$, $\mathtt{UOKVS}_{\mathtt{ValMap}}$, *and* $\mathtt{UOKVS}_{\mathtt{DocMap}}$ *be* UOKVS *schemes for maps with leakage functions* $\mathcal{L}_{\mathtt{VolMap}}$, $\mathcal{L}_{\mathtt{ValMap}}$ *and* $\mathcal{L}_{\mathtt{DocMap}}$. *Then define* $\mathsf{MKSE}_{\mathtt{MM}}$ *for* MM *as in Figure 10.*

Upon calling MKSE.Insert(key, (Id, Doc)), depending on key already existing in the system or not, UOKVS.Update or UOKVS.Insert should be executed.

Theorem 7. *The construction of* $\mathsf{MKSE}_{\mathtt{MM}}$ *from Construction 1 is a good* MKSE *with leakage functions* $\mathcal{L}_{\mathtt{MM}} = \mathcal{L}_{\mathtt{VolMap}} \cup \mathcal{L}_{\mathtt{ValMap}} \cup \mathcal{L}_{\mathtt{DocMap}} \cup_{\mathsf{key}\in\mathsf{Query}} |\mathtt{MM}[\mathsf{key}]|$. *In the above, for each* Share, Search *invocation, there are* $\mathtt{Vol} = |\mathtt{MM}[\mathsf{key}]|$ *calls to the relevant algorithm for* ValMap *and* DocMap*, this reveals* Vol *in addition to leakage from* ValMap *and* DocMap.

The proof of Theorem 7 is straightforward.

The description in Figure 10 explains our focus on insertion for UOKVS. As mentioned in the Introduction, for the Enron dataset has 33K keywords so the size of VolMap, which requires updates, never exceeds 33K. It can be fully stored in the stash without impacting overall stash size. On the other hand, the larger maps of ValMap and DocMap require insertions. So our focus is on bounding the stash size of the larger maps.

$\underline{\mathsf{Exp}_{\mathsf{Sec},\mathcal{A},\mathsf{MKSE}}(1^\lambda, q)}$

$\mathcal{A}$ sends

$$\mathtt{Clients} = \{\mathtt{C}^j\}_{j=1}^m, \quad \mathtt{Owners} = \{\mathtt{Ow}^i\}_{i=1}^n,$$
$$\mathtt{Owners}_{\mathcal{A}} \subseteq \mathtt{Owners}, \quad \mathtt{Clients}_{\mathcal{A}} \subseteq \mathtt{Clients},$$
$$\{\mathsf{KOwn}^{\mathtt{Ow}}\}_{\mathtt{Ow}\in\mathtt{Owners}_{\mathcal{A}}}, \quad \{\mathtt{MM}^{\mathtt{Ow}}\}_{\mathtt{Ow}\in\mathtt{Owners}}, \quad \{q^{\mathtt{Ow},\mathtt{C}}\}_{\mathtt{Ow},\mathtt{C}},$$

where $q^{\mathtt{Ow},\mathtt{C}} \le q$.
$\mathcal{C}$ performs:

1. For each honest owner, sample $\mathsf{KOwn}^{\mathtt{Ow}} \leftarrow \mathsf{KeyGen}^{\mathtt{Ow}}(1^\lambda)$.
2. For every owner, run
$$\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}\\ \mathtt{EMM}^{\mathtt{Ow}}\end{pmatrix} \leftarrow \mathsf{Setup}^{\mathtt{Ow},\mathtt{S}}\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}, \mathtt{MM}^{\mathtt{Ow}}\\ \perp\end{pmatrix}.$$
Initialize every $K^{\mathtt{C}} = \perp$ and send the initial adversarial view View_0 to $\mathcal{A}$.
3. For $i = 1, \ldots, q$ and every $(\mathtt{Ow}, \mathtt{C})$ with $i \le q^{\mathtt{Ow},\mathtt{C}}$, $\mathcal{A}$ sends
$$(\mathsf{key}_i^{\mathtt{Ow},\mathtt{C}}, \mathtt{Plcy}_i^{\mathtt{Ow},\mathtt{C}}, \mathtt{Shr}_i^{\mathtt{Ow},\mathtt{C}}).$$
The challenger runs
$$\begin{pmatrix}\perp\\ K^{\mathtt{C}}\end{pmatrix} \leftarrow \mathsf{Share}^{\mathtt{Ow},\mathtt{C}}\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}, \mathtt{Plcy}_i^{\mathtt{Ow},\mathtt{C}}\\ K^{\mathtt{C}}, \mathtt{Shr}_i^{\mathtt{Ow},\mathtt{C}}\end{pmatrix}$$
and
$$\begin{pmatrix}\perp\\ \mathsf{Res}_i^{\mathtt{Ow},\mathtt{C}}\end{pmatrix} \leftarrow \mathsf{Search}^{\mathtt{S},\mathtt{C}}\begin{pmatrix}\mathtt{EMM}^{\mathtt{Ow}}\\ K^{\mathtt{C}}, \mathsf{key}_i^{\mathtt{Ow},\mathtt{C}}\end{pmatrix}.$$
It sends the resulting adversarial view View_i to $\mathcal{A}$.

$\mathcal{A}$ outputs a bit b.

$\underline{\mathsf{Exp}_{\mathcal{A},\mathsf{Sim},\mathsf{MKSE}}(1^\lambda, q)}$

$\mathcal{A}$ sends the same initial inputs as in the real experiment. The challenger defines $\vec{\mathsf{KOwn}}_{\mathcal{A}} = \{(\mathtt{Ow}, \mathsf{KOwn}^{\mathtt{Ow}}) : \mathtt{Ow} \in \mathtt{Owners}_{\mathcal{A}}\}, \vec{\mathtt{MM}} = \{(\mathtt{Ow}, \mathtt{MM}^{\mathtt{Ow}}) : \mathtt{Ow} \in \mathtt{Owners}\}$. It computes

$$\mathsf{View}_0 \leftarrow \mathsf{Sim}\left(\mathcal{L}^{\mathsf{Setup}}\begin{pmatrix}\mathtt{Owners}, \mathtt{Clients}, \mathtt{Owners}_{\mathcal{A}},\\ \mathtt{Clients}_{\mathcal{A}}, \vec{\mathsf{KOwn}}_{\mathcal{A}}, \vec{\mathtt{MM}}\end{pmatrix}\right)$$

and sends View_0 to $\mathcal{A}$.
For $i = 1, \ldots, q$ and every $(\mathtt{Ow}, \mathtt{C})$ with $i \le q^{\mathtt{Ow},\mathtt{C}}$, $\mathcal{A}$ sends

$$(\mathsf{key}_i^{\mathtt{Ow},\mathtt{C}}, \mathtt{Plcy}_i^{\mathtt{Ow},\mathtt{C}}, \mathtt{Shr}_i^{\mathtt{Ow},\mathtt{C}}).$$

Let

$$\vec{x}_i := \left\{(\mathtt{Ow}, \mathtt{C}, x_1^{\mathtt{Ow},\mathtt{C}}, \ldots, x_{\min\{i, q^{\mathtt{Ow},\mathtt{C}}\}}^{\mathtt{Ow},\mathtt{C}})\right\}_{\mathtt{Ow},\mathtt{C}}$$

for $x \in \{\mathsf{key}, \mathtt{Plcy}, \mathtt{Shr}\}$. The challenger computes

$$\mathsf{View}_i \leftarrow \mathsf{Sim}\left(\mathcal{L}^{\mathsf{Grant}}(\vec{\mathtt{Plcy}}_i, \vec{\mathtt{Shr}}_i), \mathcal{L}^{\mathsf{Search}}(\vec{\mathsf{key}}_i)\right)$$

and sends View_i to $\mathcal{A}$.
Finally, output the bit produced by $\mathcal{A}$.

Figure 9: Real and simulated security experiments for MKSE. Each View_i contains the views of the corrupted owners, corrupted clients, and the server through round i.

5 Instantiating Access Control Mechanism

Access control has been extensively studied in many different primitives, including identity-based encryption (IBE) [BGK08] and attribute-based encryption (ABE) [SSW12] including with attribute versioning [SSW12, YWRL10]. Later work studies direct and indirect revocation models to balance flexibility and scalability [AI09].

In this section, we present two instantiations of AcCtrl to illustrate how standard primitives fit our framework and how to analyze leakage. Logical key hierarchy [WGL00], attribute-based encryption [BSW07], and broadcast encryption [FN93] yield natural instantiations by treating each keyword as a participant in a key distribution system. We encourage the development of a variety of access control mechanisms. We briefly discuss two further candidates:

1. BlindSeer [FVK+15] evaluates a policy, query pair using garbled circuits between the owner and client.

2. One can use Labeled Private Set Intersection (LPSI) [CHLR18, CLR17, KKRT16, PSWW18, UCK+21] where Ow prepares a set of $(\mathsf{key}, K_{\mathsf{key}})$'s where K_{key} is the key for documents associated with key, and C holds key_q. Then for any key where $\mathsf{key} = \mathsf{key}_q$, C receives K_{key} which allows them to locate and decrypt the documents corresponding to key. LPSI is equivalent to batch keyword PIR [CHLR18], so this would correspond to a PIR interaction between the client and owner, albeit on an array whose size is proportional the number of keywords the owner is willing to grant to the client, which is of size at most |VolMap|. This approach also works for sending the identifiers to client.

$\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}, \mathsf{st}^{\mathtt{Ow}} \\ \mathtt{EMM}\end{pmatrix} \leftarrow \mathsf{Setup}^{\mathtt{Ow},\mathtt{S}} \begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}, \mathtt{MM} \\ \bot\end{pmatrix}$

Ow does:

1. Parse $\mathsf{KOwn}^{\mathtt{Ow}}$ as $\mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{VolMap}}, \mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{ValMap}}, \mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{DocMap}}$.
2. Define $\mathsf{M}_{\mathtt{VolMap}}, \mathsf{M}_{\mathtt{ValMap}}, \mathsf{M}_{\mathtt{DocMap}}$ as empty maps.
3. For each $\mathsf{key} \in \mathtt{MM}$,
   (a) Define $\mathsf{Arr}[\mathsf{key}]$ and $\mathsf{M}_{\mathtt{VolMap}}[\mathsf{key}] = |\mathtt{MM}[\mathsf{key}]|$.
   (b) For $i = 1$ to $|\mathsf{Arr}[\mathsf{key}]|$, define $\mathsf{Id}_i = \mathsf{Arr}[\mathsf{key}][i].\mathsf{Id}$
   set $\mathsf{M}_{\mathtt{ValMap}}[(\mathsf{key}, i)] = \mathsf{Id}_i$
   and $\mathsf{M}_{\mathtt{DocMap}}[\mathsf{Id}_i] = \mathsf{Arr}[\mathsf{key}][i].\mathsf{Doc}$
4. $\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{VolMap}} \\ \mathsf{EMap}_{\mathtt{VolMap}}, \mathsf{st}\end{pmatrix} \leftarrow \mathtt{UOKVS.Encode}^{\mathtt{Ow},\mathtt{S}} \begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}, \mathsf{M}_{\mathtt{VolMap}} \\ \bot\end{pmatrix}$
5. $\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{ValMap}} \\ \mathsf{EMap}_{\mathtt{ValMap}}, \mathsf{st}\end{pmatrix} \leftarrow \mathtt{UOKVS.Encode}^{\mathtt{Ow},\mathtt{S}} \begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}, \mathsf{M}_{\mathtt{ValMap}} \\ \bot\end{pmatrix}$
6. $\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{DocMap}} \\ \mathsf{EMap}_{\mathtt{DocMap}}, \mathsf{st}\end{pmatrix} \leftarrow \mathtt{UOKVS.Encode}^{\mathtt{Ow},\mathtt{S}} \begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}, \mathsf{M}_{\mathtt{DocMap}} \\ \bot\end{pmatrix}$

$\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}, \mathsf{st}^{\mathtt{Ow}} \\ \mathtt{EMM}\end{pmatrix} \leftarrow \mathsf{Insert}^{\mathtt{Ow},\mathtt{S}} \begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}, (\mathsf{key}, (\mathsf{Id}, \mathsf{Doc})), \mathsf{st}^{\mathtt{Ow}} \\ \mathtt{EMM}\end{pmatrix}$

Ow does:

1. Parse $\mathsf{KOwn}^{\mathtt{Ow}}$ as $\mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{VolMap}}, \mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{ValMap}}, \mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{DocMap}}$.
2. Parse $\mathtt{EMM}$ as $\mathsf{EMap}_{\mathtt{VolMap}}, \mathsf{EMap}_{\mathtt{ValMap}}$, and $\mathsf{EMap}_{\mathtt{DocMap}}$.
3. If $\mathsf{key} \in \mathsf{M}_{\mathtt{VolMap}}$: $\mathtt{Vol} = \mathsf{M}_{\mathtt{VolMap}}[\mathsf{key}] + 1$ and $\mathsf{op} = \mathsf{Update}$,
   Else: $\mathtt{Vol} = 1$ and $\mathsf{op} = \mathsf{Insert}$.
4. $\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{VolMap}} \\ \mathsf{EMap}_{\mathtt{VolMap}}\end{pmatrix} \leftarrow \mathtt{UOKVS.op}^{\mathtt{Ow},\mathtt{S}} \begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{VolMap}}, (\mathsf{key}, \mathtt{Vol}) \\ \mathsf{EMap}_{\mathtt{VolMap}}\end{pmatrix}$
5. $\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{ValMap}} \\ \mathsf{EMap}_{\mathtt{ValMap}}\end{pmatrix} \leftarrow \mathtt{UOKVS.op}^{\mathtt{Ow},\mathtt{S}} \begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{ValMap}}, ((\mathsf{key}, \mathtt{Vol}), \mathsf{Id}) \\ \mathsf{EMap}_{\mathtt{ValMap}}\end{pmatrix}$
6. If $\mathsf{Id} \in \mathsf{M}_{\mathtt{DocMap}}$: $\mathsf{op} = \mathsf{Update}$, Else: $\mathsf{op} = \mathsf{Insert}$.
7. $\begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{DocMap}} \\ \mathsf{EMap}_{\mathtt{DocMap}}\end{pmatrix} \leftarrow \mathtt{UOKVS.op}^{\mathtt{Ow},\mathtt{S}} \begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{DocMap}}, (\mathsf{Id}, \mathsf{Doc}) \\ \mathsf{EMap}_{\mathtt{DocMap}}\end{pmatrix}$

$\begin{pmatrix}\bot \\ K^{\mathtt{C}}\end{pmatrix} \leftarrow \mathsf{Share}^{\mathtt{Ow},\mathtt{C}} \begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}, \mathtt{Plcy} \\ K^{\mathtt{C}}, \mathsf{key}\end{pmatrix}$

1. Parse $K^{\mathtt{C}}$ as $K^{\mathtt{C}}_{\mathtt{VolMap}}, K^{\mathtt{C}}_{\mathtt{ValMap}}, K^{\mathtt{C}}_{\mathtt{DocMap}}$.
2. Parse $\mathtt{EMM}$ as $\mathsf{EMap}_{\mathtt{VolMap}}, \mathsf{EMap}_{\mathtt{ValMap}}, \mathsf{EMap}_{\mathtt{DocMap}}$.
3. Parse $\mathsf{KOwn}^{\mathtt{Ow}}$ as $\mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{VolMap}}, \mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{ValMap}}, \mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{DocMap}}$.
4. $\begin{pmatrix}\bot \\ K^{\mathtt{C}}_{\mathtt{VolMap}}\end{pmatrix} \leftarrow \mathtt{UOKVS.Grant}^{\mathtt{Ow},\mathtt{C}} \begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{VolMap}}, \mathtt{Plcy} \\ K^{\mathtt{C}}_{\mathtt{VolMap}}, \mathsf{key}\end{pmatrix}$
5. Until $\mathtt{Ow}$ sends $\bot$ starting at $i = 1$
   (a) $\begin{pmatrix}\bot \\ K^{\mathtt{C}}_{\mathtt{ValMap}}\end{pmatrix} \leftarrow \mathtt{UOKVS.Grant}^{\mathtt{Ow},\mathtt{C}} \begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{ValMap}}, \mathtt{Plcy} \\ K^{\mathtt{C}}_{\mathtt{ValMap}}, (\mathsf{key}, i)\end{pmatrix}$
   (b) $\begin{pmatrix}\bot \\ K^{\mathtt{C}}_{\mathtt{DocMap}}\end{pmatrix} \leftarrow \mathtt{UOKVS.Grant}^{\mathtt{Ow},\mathtt{C}} \begin{pmatrix}\mathsf{KOwn}^{\mathtt{Ow}}_{\mathtt{DocMap}}, \mathtt{Plcy} \\ K^{\mathtt{C}}_{\mathtt{DocMap}}, \mathsf{Id}_i\end{pmatrix}$
   (c) Set $i = i + 1$

$\begin{pmatrix}\bot \\ \mathsf{Res}\end{pmatrix} \leftarrow \mathsf{Search}^{\mathtt{S},\mathtt{C}} \begin{pmatrix}\mathtt{EMM} \\ K^{\mathtt{C}}, \mathsf{key}\end{pmatrix}$

1. Parse $K^{\mathtt{C}}$ as $K^{\mathtt{C}}_{\mathtt{VolMap}}, K^{\mathtt{C}}_{\mathtt{ValMap}}, K^{\mathtt{C}}_{\mathtt{DocMap}}$.
2. Parse $\mathtt{EMM}$ as $\mathsf{EMap}_{\mathtt{VolMap}}, \mathsf{EMap}_{\mathtt{ValMap}}, \mathsf{EMap}_{\mathtt{DocMap}}$.
3. Perform $\begin{pmatrix}\bot \\ \mathtt{Vol}\end{pmatrix} \leftarrow \mathtt{UOKVS.Decode}^{\mathtt{S},\mathtt{C}} \begin{pmatrix}\mathsf{EMap}_{\mathtt{VolMap}} \\ \mathsf{key}, K^{\mathtt{C}}_{\mathtt{VolMap}}\end{pmatrix}$
4. $\forall 1 \le i \le \mathtt{Vol}$
   (a) $\begin{pmatrix}\bot \\ \mathsf{Id}_i\end{pmatrix} \leftarrow \mathtt{UOKVS.Decode}^{\mathtt{S},\mathtt{C}} \begin{pmatrix}\mathsf{EMap}_{\mathtt{ValMap}} \\ (\mathsf{key}, i), K^{\mathtt{C}}_{\mathtt{ValMap}}\end{pmatrix}$
   (b) $\begin{pmatrix}\bot \\ \mathsf{Res}_i\end{pmatrix} \leftarrow \mathtt{UOKVS.Decode}^{\mathtt{S},\mathtt{C}} \begin{pmatrix}\mathsf{EMap}_{\mathtt{DocMap}} \\ \mathsf{Id}_i, K^{\mathtt{C}}_{\mathtt{DocMap}}\end{pmatrix}$
5. Output $\mathsf{Res} = \mathsf{Res}_1, ..., \mathsf{Res}_{\mathtt{Vol}}$.

Figure 10: MKSE for MM from UOKVS.

The first construction (Construction 2) uses symmetric key cryptography, while the second uses oblivious PRF or OPRF [CHL22, NR04, FIPR05] evaluation between owner and client. The latter lets the owner control how many keyword keys are issued per client without learning queries, and limits the number of decrypted keywords.

5.1 Keyword AcCtrl from PRFs

Construction 2. *Let* $F_{\mathsf{L}}, F_{\mathsf{D}} : \{0,1\}^\lambda \times \mathcal{W} \to \{0,1\}^\lambda, F_{\mathsf{Enc}} : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^{\eta+\lambda}$ *be PRFs. Define* (Setup, Update, Grant, Decrypt) *as in Figure 11.*

$\underline{\mathsf{Setup}(\mathsf{Arr}_\mathsf{M})}$

1. Sample independent PRF seeds $k_\mathsf{L}, k_\mathsf{D} \leftarrow \{0,1\}^\lambda$. Initialize empty maps $K^\mathsf{L}, K^\mathsf{D}$ and set $\mathsf{KOwn} := (k_\mathsf{L}, k_\mathsf{D}, K^\mathsf{L}, K^\mathsf{D})$.
2. Initialize aligned arrays $\vec{K}^\mathsf{L}$ and CArr of length ℓ.
3. For every $1 \le i \le \ell$, parse $\mathsf{Arr}_\mathsf{M}[i] = (\mathsf{key}_i, \mathsf{val}_i)$, run $(\mathsf{CArr}[i], \mathsf{KOwn}) \leftarrow \mathsf{Update}^{\mathtt{Ow}}(\mathsf{KOwn}, \mathsf{key}_i, \mathsf{val}_i)$, and set $\vec{K}^\mathsf{L}[i] := K^\mathsf{L}[\mathsf{key}_i]$. Output $(\mathsf{CArr}, \mathsf{KOwn})$.

$\underline{\mathsf{Update}^{\mathtt{Ow}}(\mathsf{KOwn}, \mathsf{key}, \mathsf{val})}$

1. Parse $\mathsf{KOwn} = (k_\mathsf{L}, k_\mathsf{D}, K^\mathsf{L}, K^\mathsf{D})$.
2. If $\mathsf{key} \notin K^\mathsf{L}$, set $K^\mathsf{L}[\mathsf{key}] := F_\mathsf{L}(k_\mathsf{L}, \mathsf{key})$ and $K^\mathsf{D}[\mathsf{key}] := F_\mathsf{D}(k_\mathsf{D}, \mathsf{key})$.
3. Sample $\mathtt{nonce} \leftarrow \{0,1\}^\lambda$ and set $c := F_{\mathsf{Enc}}(K^\mathsf{D}[\mathsf{key}], \mathtt{nonce}) \oplus (\mathsf{val} \,\|\, 0^\lambda)$. Output $((c, \mathtt{nonce}), \mathsf{KOwn})$.

$\underline{\mathsf{Grant}^{\mathtt{Ow},\mathtt{C}}(\mathsf{KOwn}, \mathtt{Plcy}; \mathsf{KClt}, \vec{\mathtt{Shr}})}$

Direct PRF version For every $\mathsf{key} \in \vec{\mathtt{Shr}}$ satisfying $\mathtt{Plcy}$, the owner reads $(K^\mathsf{L}[\mathsf{key}], K^\mathsf{D}[\mathsf{key}])$ from its maps and sends the pair to the client. We require direct grants to concern keywords already initialized by Setup or Update.

OPRF version For each authorized request, the parties run independent OPRF evaluations for F_L and F_D. The client receives $\big(F_\mathsf{L}(k_\mathsf{L}, \mathsf{key}), F_\mathsf{D}(k_\mathsf{D}, \mathsf{key})\big)$ without revealing key to the owner. These values equal the entries that Setup or Update stores in the owner's keyword-indexed maps. The owner does not test whether key is present in those maps.

$\underline{\mathsf{Decrypt}^{\mathtt{C}}(\vec{c}, \vec{K}^\mathsf{D})}$

1. Set $\mathsf{Res} = \perp$.
2. For every $(c, \mathtt{nonce}) \in \vec{c}$ and every $K^\mathsf{D} \in \vec{K}^\mathsf{D}$, compute $m = F_{\mathsf{Enc}}(K^\mathsf{D}, \mathtt{nonce}) \oplus c$. If m ends in λ zero bits, append $m_{1\ldots\eta}$ to Res.
3. Output Res.

Figure 11: Access control producing a lookup key and a decryption key for every keyword.

Theorem 8. *Let all variables be as in Construction 2 with the direct PRF version of* Grant. *Define* $\mathcal{L} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Op}})$ *as follows.*

1. $\mathcal{L}_{\mathsf{Setup}}$ *includes* ℓ, η.
2. *For every operation,* $\mathcal{L}_{\mathsf{Op}}$ *includes its type. For an* Update *it also includes the destination position and the equality class of its keyword relative to preceding Setup and Update occurrences.*
3. *If the owner is corrupted, the leakage additionally includes the initial array and the complete inputs to every* Update *and* Grant.
4. *If the client is corrupted, it selectively commits before* Setup *to* $\mathsf{Auth}^\star$. *For every* $\mathsf{key} \in \mathsf{Auth}^\star$, *the setup leakage includes* $\mathsf{Setup}_{\mathsf{key}} = \{(j, \mathsf{val}) : \mathsf{Arr}_0[j] = (\mathsf{key}, \mathsf{val})\}$. *An* Update *of such a keyword leaks* (j, val) *when it occurs.* Grant *leaks the requested keywords, the policy outcome, and the keys that are authorized.*
5. *If neither party is corrupted, no additional information is included.*

Then AcCtrl *is a secure access-control mechanism.*

Proof. Correctness follows from the persistent keyword-indexed maps K^L and K^D. Setup, Update and Grant therefore use the same lookup and decryption keys for every occurrence of a fixed keyword. Correct decryption recovers $\mathsf{val} \,\|\, 0^\lambda$.

For distinct keywords, collision freedom of the lookup keys follows from PRF security of F_{L} and a union bound over polynomially many keyword occurrences. After replacing F_{Enc} by a random function, decryption under an incorrect key passes the trailing-zero test with probability $2^{-\lambda}$. A union bound over all decryption attempts proves correctness except with negligible probability.

We next prove security through a sequence of hybrids. Let $\mathsf{Auth}^\star$ be the authorization group declared before Setup when the client is corrupted.

In the first hybrid, replace F_{L} and F_{D} by independent random functions. These replacements are indistinguishable by PRF security, including when the adversary chooses subsequent operations as a function of the preceding transcript. The functions are evaluated lazily, so every repeated keyword receives the same independently sampled pair $(K^{\mathsf{L}}[\mathsf{key}], K^{\mathsf{D}}[\mathsf{key}])$.

Consider first an honest owner and a corrupted client. For every $\mathsf{key} \in \mathsf{Auth}^\star$, the simulator samples and stores a pair $(K^{\mathsf{L}}[\mathsf{key}], K^{\mathsf{D}}[\mathsf{key}])$. Using the leaked initial values and the values leaked by later Update, it constructs the corresponding ciphertexts honestly. If a later Grant authorizes key, the simulator releases the previously stored key pair.

For every $\mathsf{key} \notin \mathsf{Auth}^\star$, the decryption key is never released. Replace the outputs of $F_{\mathsf{Enc}}(K^{\mathsf{D}}[\mathsf{key}], \cdot)$ at all fresh nonces by uniform strings. A polynomial-size multi-user hybrid and PRF security of F_{Enc} make this replacement indistinguishable. Nonce collisions occur with only negligible probability.

The simulator uses the leaked keyword-equality pattern to assign one uniform lookup key to each equivalence class of keyword occurrences. On every Update it installs that class's lookup key at the leaked position. It constructs ciphertexts honestly for classes in $\mathsf{Auth}^\star$ and uses independent uniform ciphertexts for all other classes. Thus it can process every operation online while preserving the consistency visible in the public lookup-key arrays.

If the owner is corrupted, the leakage contains the owner's inputs. The simulator samples the owner's PRF seeds and runs $\mathsf{Setup}, \mathsf{Update}$ and Grant honestly. If neither party is corrupted, no decryption key appears in the adversarial view, so the simulator samples consistent lookup keys and replaces every ciphertext by an independent uniform string.

The final simulated distribution is identical to the last hybrid. Each transition is indistinguishable by the security of F_{L}, F_{D}, or F_{Enc}. The argument permits Update operations and their destination positions to be selected from the preceding transcript; only the set of keywords whose decryption keys may eventually be released is fixed before Setup. □

5.1.1 Replacing PRFs with OPRFs

To hide requested keywords from the owner, replace the evaluations of F_{L} and F_{D} in Grant by independent OPRF evaluations. The owner uses the corresponding noninteractive evaluations in Setup and Update.

Definition 5 (Oblivious Pseudo Random Function (OPRF) [NR04, FIPR05])**.** *Let* OPRF *be a two-party protocol, such that,*

$$\begin{pmatrix} \\ F_{\mathsf{KOwn}}(x) \end{pmatrix} \leftarrow \mathsf{OPRF}^{\mathtt{Ow},\mathtt{C}} \begin{pmatrix} \mathsf{KOwn} \\ x \end{pmatrix}.$$

Let F_{KOwn} *be a non-interactive evaluation with the same outcome. We call* OPRF *a secure OPRF if*

1. *F is a PRF family,*
2. OPRF *outputs* $F_{\mathsf{KOwn}}(x)$ *with overwhelming probability,*
3. *For every PPT* $\mathcal{A}^{\mathtt{Ow}}_{\mathsf{OPRF}}$ *that specifies a vector of inputs* $\vec{x}, \mathsf{KOwn}$ *and observes* $\mathtt{Ow}$*'s view of* OPRF *on each input there exists a PPT* $\mathsf{Sim}^{\mathtt{Ow}}_{\mathsf{OPRF}}(|\vec{x}|)$ *that produces indistinguishable views, and*
4. *For every PPT* $\mathcal{A}^{\mathtt{C}}_{\mathsf{OPRF}}$ *that specifies a vector of inputs* $\vec{x}, \mathsf{KOwn}$ *and observes* $\mathtt{C}$*'s view of* OPRF *on each input with a uniform random* KOwn *there PPT* $\mathsf{Sim}^{\mathtt{C}}_{\mathsf{OPRF}}(\vec{x}, \vec{F_{\mathsf{KOwn}}}(x))$ *that produces indistinguishable views.*

Construction 3. *Let* $F_{\mathsf{L}}, F_{\mathsf{D}}, F_{\mathsf{Enc}}$ *be as in Construction 2. Let* $\mathsf{OPRF}_{\mathsf{L}}$ *and* $\mathsf{OPRF}_{\mathsf{D}}$ *securely realize the interactive evaluations of* F_{L} *and* F_{D}*, respectively. Use the OPRF version of* Grant *in Figure 11.*

Theorem 9. *For Construction 3, use the leakage of Theorem 8, except that when the owner is corrupted a* Grant *leaks only* $\mathtt{Plcy}_i$ *and* $|\vec{\mathsf{key}}_i|$. *Then* $\mathsf{AcCtrl}_{\mathsf{OPRF}}$ *is a secure access-control mechanism.*

Proof. Apply the proof of Theorem 8 and, for each grant, insert two additional hybrids replacing the real transcripts of $\mathsf{OPRF}_{\mathsf{L}}$ and $\mathsf{OPRF}_{\mathsf{D}}$ by their simulated transcripts. OPRF security makes each replacement indistinguishable. The simulated client outputs are the same consistent lookup/decryption-key pairs used in the PRF proof, so the remainder of that proof is unchanged. □

6 Implementation and Experimental Results

We present a proof-of-concept implementation of our `UOKVS` and MKSE in Rust available via our GitHub repository. We use Construction 2 for access control and FrodoPIR [DPC23] for the underlying PIR.

When comparing against prior MKSE schemes we only benchmark the combination of `VolMap` and `ValMap` as in common in searchable encryption, however we do implement the full retrieval using `DocMap`. We treat a plaintext map as requiring 40 bytes for each keyword document identifier pair, 32 bytes to store a hash of the keyword and 8 bytes for an identifier. Using the Enron dataset, the raw dataset is 2.6GB, associating each keyword with document identifiers requires 321MB.

The most secure version of state of the art MKSE is called FNU [WP21]. It uses an oblivious map, we consider a tree-based oblivious map from Wang et al. [WNL+14] of size 33132 to store `VolMap` with a branching factor of 8, requiring $\log_8(33132) + 1 = 6$ round-trips to `VolMap`. 33132 is number of keywords in the Enron dataset.

6.1 Implementation Choices

We utilize Blake3 and ChaCha20 as our cryptographic algorithms. We use FrodoPIR [DPC23] as our PIR due to its maturity and flexibility. We compared performance to the batch PIR by Mughees and Ren [MR23]. For a database of size 2^{20} a query where $|\mathsf{Docs}_{\mathsf{key}}| = 10000$ Mughees and Ren's implementation takes 85 seconds, which is slightly faster than the same query using FrodoPIR, which took 97 seconds. However, when $|\mathsf{Docs}_{\mathsf{key}}| = 700$ and one scales down the batch size for Mughees and Ren, their query took 57 seconds while FrodoPIR was 6 seconds. We chose FrodoPIR due to superior performance on representative queries, which we further optimized.

All evaluations and benchmarks are performed on a server with an AMD Ryzen Threadripper PRO 7995WX CPU with 96 physical cores and 768 GB of RAM, running Ubuntu 22.04. We utilize the Enron Corpus [KY04] to evaluate our scheme. For Enron, we generate subsets with keyword-value maps ranging in size from 2^{16} to 24M, which is $\approx 2^{24.5}$, where the upper bound corresponds to the size of the full Enron Corpus.

To evaluate our scheme, we set a configuration in which each document pointer is encrypted to 32 bytes, including a 12-byte nonce, with each bucket containing $p = 256$ encrypted pointers resulting in a total of $B = 8192$ bytes of data per bucket. That is, for a multimap of size ℓ DIP creates $\mu = 2 \cdot \ell/p$ total positions in $\mathsf{Buckets}$. We set $s = 4$ hash functions in the majority of our evaluation. We use a slight variants of the random walk and BFS hashing described earlier:

1. For random walk, we check if any of the s positions of the initial item are empty, if not we pick a random position to start the eviction chain.

2. For BFS Cuckoo hashing, at most two hashes are added to the queue for each item. There is also a maximum depth of five for chain length in addition to the α restriction.

We build an optimized fork of FrodoPIR which utilizes hardware intrinsics and supports updates on an array where each position is $B = 2^{13}$ bytes (8192 bytes). The resulting buckets are then transformed into a database that can be queried using FrodoPIR. We perform queries for each dataset, considering queries over key where $\mathsf{Docs}_{\mathsf{key}}$ is 100, 500, or 1000. The median keyword for Enron is associated with 21 documents.

	Setup, Query, Bandwidth, Stash, and Storage Statistics for Enron										
		$\|\texttt{MM}[\mathsf{key}]\| = 100$		$\|\texttt{MM}[\mathsf{key}]\| = 500$		$\|\texttt{MM}[\mathsf{key}]\| = 1000$		Stash			
Size	Setup(s)	Q(ms)	BW(MB)	Q(ms)	BW(MB)	Q(ms)	BW(MB)	RW	BFS	$\|\texttt{MM}[\mathsf{key}]\|$	Storage(GB)
2^{16}	2	42	0.26	43	0.38	43	0.62	0	0	2	0.05
2^{17}	7	46	0.27	44	0.40	47	0.65	0	0	4	0.06
2^{18}	19	48	0.29	44	0.43	48	0.71	0	0	8	0.09
2^{19}	33	49	0.32	50	0.49	51	0.84	0	0	15	0.14
2^{20}	53	50	0.38	51	0.62	53	1.09	0	0	32	0.25
2^{21}	87	53	0.51	52	0.87	57	1.59	0	0	63	0.48
2^{22}	136	56	0.76	55	1.37	60	2.59	0	0	126	0.92
2^{23}	186	63	1.26	60	2.37	71	4.59	0	0	253	1.81
2^{24}	246	74	2.26	67	4.37	87	8.59	3	0	506	3.59
24M	297	110	3.17	116	6.19	125	12.24	4	0	737	5.21

Table 3: Setup, Query, Bandwidth, Stash, and Storage Statistics for Enron Corpus $p = 256$. The numbers in column 'Storage' are reported in GB. $|\texttt{MM}[\mathsf{key}]|$ is the average number matching Docs across key. 24M is the full size of the Enron Corpus. Setup is reported in seconds, and queries in milliseconds. Bandwidth (BW) is reported in MB. Queries are over keywords linked to 100, 500, and 1000 documents. Query times are amortized due to our optimizations, including hardware intrinsics, which can make smaller result sets appear slower. However, query performance does degrade on large queries, a query with 50,000 matching documents took 2.42 seconds.

6.2 Experimental Results

Table 3 shows our primary results with bandwidth, computation, storage, and stash size. In this table all items are jointly inserted using the Encode procedure. We separately discuss Insert performance shortly. The first row represents a dataset of size 2^{16}, which takes 2 seconds to setup for a subset of the Enron Corpus. The setup is a one-time process. Following this, it takes 42 milliseconds for each query on keywords with 100 documents linked to them.

For the full Enron Corpus, setup completes in 297 seconds or about 5 minutes. We perform queries to keywords with 100, 500, and 1000 linked documents in $125, 116$, and 110 milliseconds, respectively, with a stash of size 4 when Encode uses RW eviction with $\alpha = 10$. No stash is present when using BFS eviction with $\alpha = 20$ and depth 5. Storage is 5.21GB for `VolMap` and `ValMap`, and 8.04GB for all three maps. The bandwidth is $3.17, 6.19$, and 12.24MB for $|\mathsf{Doc}_{\mathsf{key}}| = 100, 500$, and 1000 respectively. We only report the bandwidth of the PIR protocol; the stash is small enough to not impact performance.

Each query corresponds to two steps: first, retrieving volume from `VolMap`, which involves four PIR queries, one to each hash position; second, retrieving the documents from `ValMap`, which involves $4 \cdot |\mathsf{Docs}_{\mathsf{key}}|/256$ array positions in the PIR. As such, computational time (and bandwidth) scales linearly with the number of matching documents. However, our optimizations to FrodoPIR utilize SIMD instructions and multiple threads, allowing us to amortize our query times. For example, querying a keyword associated with $100, 500$, and 1000 documents in the entire Enron corpus takes < 125 ms for both `VolMap` and `ValMap`. A query with 50,000 matching documents took 2.42 seconds.

Insertions and stash size In all experiments where the full Enron corpus was encoded simultaneously, the stash never exceeded 4 entries. Our cryptographic implementation uses BFS with $s = 4$ as it provided the best balance of stash size and query efficiency. We were also able to prove stronger stash size bounds on BFS eviction. Figure 12 shows a comparison of RW and BFS hashing, both the bounds and our experimental results. In these graphs s hash functions are used with a RW of length α. As mentioned above in Figure 1, using BFS based eviction reduces stash size by several orders of magnitude. Based on our experiments, as long as the load ρ remains under .75, BFS eviction can safely handle insertions with a stash size fraction of 10^{-5}.

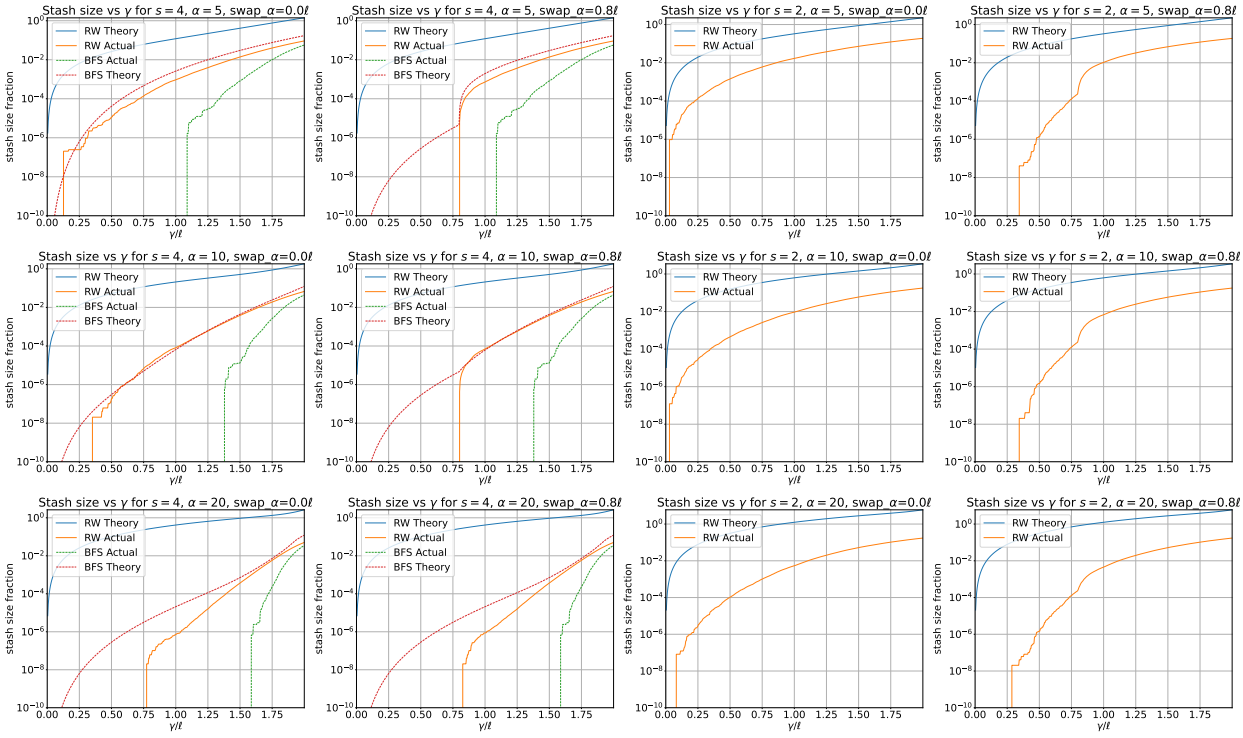

Figure 12: Summary of stash size for Cuckoo hashing computed using Theorems 2 and 5 with $s = 4$, $\ell = 24$M (size of Enron dataset), $\mu = 2.0\ell$, *swap_*α represents the switch from $\alpha = 100$ to the one denoted in the graph.

Insertion time + Bandwidth Inserting a single item using random walk in FrodoPIR takes 2ms on Enron with 80% already built using `Encode` and $\alpha = 10$. This requires a bandwidth of 82KB. BFS takes 4ms with a bandwidth of 164KB under the same parameters while allowing at most $\alpha = 20$ and depth 5. Additionally, one must download the updated PIR hint, which is 291KB for random walk and 582KB for BFS. We note that size of the hint update is the same as the size of the hint so clients can download the current hint before the query. Our improvements to hint updates improves the computational efficiency for the `S`. See Appendix B.

`DocMap` retrieval We additionally support a third round in which the user queries `DocMap` to retrieve documents directly, following the two retrieval rounds against `VolMap` and `ValMap`. To reduce bandwidth, we compress documents using Brotli. Under this scheme, retrieving 100 documents takes approximately 8 seconds.

Sharding to Parallelize `Search` To improve throughput on larger datasets, one could shard the dataset across multiple servers, each holding a smaller partition. Querying all shards in parallel yields a nearly linear speedup, with the tradeoff that bandwidth scales proportionally with the number of shards. This approach also dramatically reduces setup time, since each server operates on a smaller matrix independently. However, this approach does increase bandwidth as one sends a response for each shard.

7 Further Prior Work

OKVS Oblivious Key-Value Store was first formalized by Garimella et al. [GPR+21] and has a major application in the Private Set Intersection (PSI) [PRTY20, RR22, BPSY23, HLP+24]. PaXoS [PRTY20] uses Cuckoo Hashing with two hash functions achieving the rate of 0.4. Later, Garimella et al. [GPR+21]

extended PaXoS to use three hash functions, resulting in an increase in the rate of encoding up to 0.81. Raghuraman and Rindal [RR22] optimized OKVS construction by triangulation of the matrix. They also define the notion of doubly obliviousness for OKVS. Bienstock et al. [BPSY23] achieve a rate of .97 Hao et al. [HLP+24] introduced the notion of Oblivious Key-Value Retrieval (OKVR) where they apply a PIR on top of OKVS. As discussed in the previous section, we support variable load ρ which is also the rate of our OKVS.

Multi-Key Searchable Encryption The first generation of multi-key schemes [PZ13,PSV+14,KOR+16] have joint access pattern leakage: identifiers for a set of documents that match each client's query are visible to the server. These identifiers are persistent across queries and clients. By colluding with an adversarial `C` or `Ow` that can issue their own queries, this leakage can be devastating [GMN+16,VRMÖ17,HSWW18]. This is why current approaches for securing `DocMap` such as DDR-SSE [GMPP26] provide little security in MKSE applications, they protect against a server trying to recover queries [IKK12, CGPR15, KKNO16, WLD+17, GSB+17,GLMP18,MT19,KE19,KPT20,GPP23] but don't help against an owner that can match this access pattern to plaintext queries.

The second generation of solutions [HSWW18, WP21, WP23] addresses this issue by duplicating data when it is shared to each `C` decoupling different clients' access patterns. This approach causes storage to grow; server storage for a single owner may reach $O(|\texttt{MM}|m)$ for m clients.[1]

The current state of the art is due to Wang and Papadopoulos [WP21, WP23]. They present two main replication approaches for `VolMap` and `ValMap`, they do not propose an approach to secure `DocMap`. They recommend using an oblivious map using tree based ORAM to secure `VolMap` and symmetric key techniques to `ValMap`. The symmetric key solution for `ValMap` we call MUSSE leaks:

1. When a keyword is searched, the server learns the timestamp of its last `Shr` and sees the accessed location.
2. For each shared keyword, the server learns the set of timestamps this keyword was shared to the client.
3. An adversarial owner learns $\vec{\texttt{Shr}}$.
4. An adversarial client learns the subset of the map that has been shared with them.

In our solution, an adversarial client does learn when items have been inserted and moved by the `Ow`. The only techniques we are aware of to remove leakage of shared state updates while retaining efficiency is multi-client ORAM [MMRS17].

Acknowledgments

The authors thank the anonymous reviewers for their help in improving the manuscript. The work of A.S., M.R., and B.F. has been supported by NSF grants #2141033 and #2232813.

References

[ADMT22] Aydin Abadi, Changyu Dong, Steven J Murdoch, and Sotirios Terzis. Multi-party updatable delegated private set intersection. In *Financial Crypto*, pages 100–119. Springer, 2022.

[AG22] Megumi Ando and Marilyn George. On the cost of suppressing volume for encrypted multi-maps. *PoPETS*, 4:44–65, 2022.

[AGY+22] Nitish Andola, Raghav Gahlot, Vijay Kumar Yadav, S Venkatesan, and Shekhar Verma. Searchable encryption on the cloud: a survey. *The Journal of Supercomputing*, 78(7):9952–9984, 2022.

[1] Hamlin et al. [HSWW18] present a second construction where the size of the sharing is independent of the number of shared values that relies on public-coin different inputs obfuscation [IPS15]. We restrict our attention to standard cryptographic tools that can be implemented on benchmark datasets.

[AI09] Nuttapong Attrapadung and Hideki Imai. Attribute-based encryption supporting direct/indirect revocation modes. In *IMA International Conference, Cryptography and Coding*, pages 278–300. Springer, 2009.

[ALP+21] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication–Computation trade-offs in PIR. In *USENIX Security*, pages 1811–1828, 2021.

[ALZ+25] Ali Arastehfard, Weiran Liu, Qixian Zhou, Zinan Shen, Liqiang Peng, Lin Qu, Shuya Feng, and Yuan Hong. Kpir-c: Keyword pir with arbitrary server-side computation. *Cryptology ePrint Archive*, 2025.

[APP+23] Ghous Amjad, Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. Dynamic volume-hiding encrypted multi-maps with applications to searchable encryption. *PoPETS*, 1:417–436, 2023.

[BBF+21] Angèle Bossuat, Raphael Bost, Pierre-Alain Fouque, Brice Minaud, and Michael Reichle. SSE and SSD: page-efficient searchable symmetric encryption. In *CRYPTO 2021*, pages 157–184. Springer, 2021.

[BF24] Tolson Bell and Alan Frieze. O (1) insertion for random walk d-ary cuckoo hashing up to the load threshold. In *FOCS*, pages 106–119. IEEE, 2024.

[BGK08] Alexandra Boldyreva, Vipul Goyal, and Virendra Kumar. Identity-based encryption with efficient revocation. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM CCS*, pages 417–426. ACM, 2008.

[BHJP14] Christoph Bösch, Pieter Hartel, Willem Jonker, and Andreas Peter. A survey of provably secure searchable encryption. *ACM Computing Surveys (CSUR)*, 47(2):1–51, 2014.

[BMS+24] Saikrishna Badrinarayanan, Peihan Miao, Xinyi Shi, Max Tromanhauser, and Ruida Zeng. Updatable private set intersection revisited: Extended functionalities, deletion, and worst-case complexity. In *Asiacrypt*, pages 200–233. Springer, 2024.

[BMX22] Saikrishna Badrinarayanan, Peihan Miao, and Tiancheng Xie. Updatable private set intersection. *PoPETS*, 2022(2), 2022.

[BPSY23] Alexander Bienstock, Sarvar Patel, Joon Young Seo, and Kevin Yeo. Near-Optimal oblivious Key-Value stores for efficient PSI, PSU and Volume-Hiding Multi-Maps. In *USENIX Security*, pages 301–318, 2023.

[BSW07] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *IEEE S&P*, pages 321–334. IEEE, 2007.

[CD24] Sofía Celi and Alex Davidson. Call me by my name: Simple, practical private information retrieval for keyword queries. In *CCS*, 2024.

[CGN97] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. *TR CS0917, Dept. of Computer Science, Technion, 1997*, 1997.

[CGPR15] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, pages 668–679, 2015.

[CHL22] Sílvia Casacuberta, Julia Hesse, and Anja Lehmann. Sok: Oblivious pseudorandom functions. In *EuroS&P*, pages 625–646, 2022.

[CHLR18] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled psi from fully homomorphic encryption with malicious security. In *CCS*, pages 1223–1237, 2018.

[CLR17] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *CCS*, pages 1243–1255, 2017.

[DPC23] Alex Davidson, Gonçalo Pestana, and Sofía Celi. FrodoPIR: Simple, scalable, single-server private information retrieval. *PoPETS*, 2023.

[DPPS20] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. SEAL: Attack mitigation for encrypted databases via adjustable leakage. In *USENIX Security*, pages 2433–2450, 2020.

[FIPR05] Michael J Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*, pages 303–324. Springer, 2005.

[FN93] Amos Fiat and Moni Naor. Broadcast encryption. In *CRYPTO*, pages 480–491. Springer, 1993.

[FPSS05] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems*, 38(2):229–248, 2005.

[Fre75] David A Freedman. On tail probabilities for martingales. *the Annals of Probability*, pages 100–118, 1975.

[FVK+15] Ben A Fisch, Binh Vo, Fernando Krell, Abishek Kumarasubramanian, Vladimir Kolesnikov, Tal Malkin, and Steven M Bellovin. Malicious-client security in blind seer: a scalable private DBMS. In *IEEE S&P*, pages 395–410. IEEE, 2015.

[FVY+17] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K Cunningham. SoK: Cryptographically protected database search. In *IEEE S&P*, pages 172–191. IEEE, 2017.

[GKL+20] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. In *USENIX Security*, pages 2451–2468, 2020.

[GLMP18] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *CCS*, pages 315–331, 2018.

[GMN+16] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *CCS*, pages 1353–1364, 2016.

[GMPP26] Zichen Gui, Simon-Philipp Merz, Kenneth G Paterson, and Sikhar Patranabis. Ddr-sse: Duplicated retrieval of documents for system-wide secure searchable symmetric encryption. In *USENIX Security*, 2026.

[GPP23] Zichen Gui, Kenneth G. Paterson, and Sikhar Patranabis. Rethinking searchable symmetric encryption. In *IEEE S&P*, 2023.

[GPPW24] Zichen Gui, Kenneth G Paterson, Sikhar Patranabis, and Bogdan Warinschi. Swissse: System-wide security for searchable symmetric encryption. *PoPETS*, 2024.

[GPR+21] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Oblivious key-value stores and amplification for private set intersection. In *CRYPTO*, pages 395–425. Springer, 2021.

[GSB+17] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *IEEE S&P*, pages 655–672. IEEE, 2017.

[HKLS24] Kyoohyung Han, Seongkwang Kim, Byeonghak Lee, and Yongha Son. Revisiting okvs-based oprf and psi: Cryptanalysis and better construction. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT*, 2024.

[HLP+24] Meng Hao, Weiran Liu, Liqiang Peng, Hongwei Li, Cong Zhang, Hanxiao Chen, and Tianwei Zhang. Unbalanced Circuit-PSI from oblivious Key-Value retrieval. In *USENIX Security*, pages 6435–6451, 2024.

[HPPY25] Alexander Hoover, Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Plinko: single-server pir with efficient updates via invertible prfs. In *EUROCRYPT*, pages 3–33. Springer, 2025.

[HSWW18] Ariel Hamlin, Abhi Shelat, Mor Weiss, and Daniel Wichs. Multi-key searchable encryption, revisited. In *PKC*, pages 95–124. Springer, 2018.

[IKK12] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In *NDSS*, volume 20, page 12. Citeseer, 2012.

[IPS15] Yuval Ishai, Omkant Pandey, and Amit Sahai. Public-coin differing-inputs obfuscation and its applications. In *TCC*, pages 668–697. Springer, 2015.

[KE19] Evgenios M Kornaropoulos and Petros Efstathopoulos. The case of adversarial inputs for secure similarity approximation protocols. In *EuroS&P*, pages 247–262. IEEE, 2019.

[KKM+22] Seny Kamara, Abdelkarim Kati, Tarik Moataz, Thomas Schneider, Amos Treiber, and Michael Yonli. Sok: Cryptanalysis of encrypted search with leaker - a framework for leakage attack evaluation on real-world data. In *Euro S&P*, 2022.

[KKNO16] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *CCS*, pages 1329–1340, 2016.

[KKRT16] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In *CCS*, pages 818–829, 2016.

[KM18] Seny Kamara and Tarik Moataz. Encrypted multi-maps with computationally-secure leakage. Cryptology ePrint Archive, Paper 2018/978, 2018.

[KMO18] Seny Kamara, Tarik Moataz, and Olya Ohrimenko. Structured encryption and leakage suppression. In *CRYPTO*, pages 339–370. Springer, 2018.

[KOR+16] Aggelos Kiayias, Ozgur Oksuz, Alexander Russell, Qiang Tang, and Bing Wang. Efficient encrypted keyword search for multi-user data sharing. In *ESORICS*, pages 173–195. Springer, 2016.

[KPT20] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. The state of the uniform: attacks on encrypted databases beyond the uniform query distribution. In *IEEE S&P*, pages 1223–1240. IEEE, 2020.

[KY04] Bryan Klimt and Yiming Yang. The enron corpus: A new dataset for email classification research. In *European conference on machine learning*, pages 217–226. Springer, 2004.

[LMM+23] Feng Li, Jianfeng Ma, Yinbin Miao, Ximeng Liu, Jianting Ning, and Robert H Deng. A survey on searchable symmetric encryption. *ACM Computing Surveys*, 56(5):1–42, 2023.

[LMW23] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. Doubly efficient private information retrieval and fully homomorphic RAM computation from ring LWE. In *STOC*, page 595–608, 2023.

[LTT+25] Guowei Ling, Peng Tang, Fei Tang, Shifeng Sun, Shouling Ji, and Weidong Qiu. Ultra-fast private set intersection from efficient oblivious key-value stores. *IEEE Transactions on Dependable and Secure Computing*, 2025.

[MK22] Rasoul Akhavan Mahdavi and Florian Kerschbaum. Constant-weight PIR: Single-round keyword PIR via constant-weight equality operators. In *USENIX Security*, pages 1723–1740, 2022.

[MMRS17] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. Maliciously secure multi-client oram. In *ACNS*, pages 645–664. Springer, 2017.

[MR23] Muhammad Haris Mughees and Ling Ren. Vectorized batch private information retrieval. In *IEEE S&P*, pages 437–452. IEEE, 2023.

[MT19] Evangelia Anna Markatou and Roberto Tamassia. Full database reconstruction with access and search pattern leakage. In *ISC*, pages 25–43. Springer, 2019.

[MVA+23] Sujaya Maiyya, Sharath Chandra Vemula, Divyakant Agrawal, Amr El Abbadi, and Florian Kerschbaum. Waffle: An online oblivious datastore for protecting data access patterns. *Proceedings of the ACM on Management of Data*, 1(4):1–25, 2023.

[NR04] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. *J. ACM*, 51(2):231–262, March 2004.

[PR01] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *European Symposium on Algorithms*, pages 121–133. Springer, 2001.

[PRTY20] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Psi from paxos: Fast, malicious private set intersection. In *EUROCRYPT*, pages 739–767. Springer, 2020.

[PSV+14] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. Building web applications on top of encrypted data using mylar. In *NSDI*, pages 157–172, 2014.

[PSWW18] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based PSI via cuckoo hashing. In *Eurocrypt*, pages 125–157. Springer, 2018.

[PSY23] Sarvar Patel, Joon Young Seo, and Kevin Yeo. Don't be dense: Efficient keyword PIR for sparse databases. In *USENIX Security*, pages 3853–3870, 2023.

[PT12] Mihai Pătraşcu and Mikkel Thorup. The power of simple tabulation hashing. *Journal of the ACM (JACM)*, 59(3):1–50, 2012.

[PZ13] Raluca Ada Popa and Nickolai Zeldovich. Multi-key searchable encryption. *Cryptology ePrint Archive*, 2013.

[RR22] Srinivasan Raghuraman and Peter Rindal. Blazing fast psi from improved okvs and subfield vole. In *ACM CCS*, pages 2505–2517, 2022.

[SSW12] Amit Sahai, Hakan Seyalioglu, and Brent Waters. Dynamic credentials and ciphertext delegation for attribute-based encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 199–217. Springer, 2012.

[SWP00] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *IEEE S&P*, pages 44–55. IEEE, 2000.

[Tho17] Mikkel Thorup. Fast and powerful hashing using tabulation. *Communications of the ACM*, 60(7):94–101, 2017.

[UCK+21] Erkam Uzun, Simon P Chung, Vladimir Kolesnikov, Alexandra Boldyreva, and Wenke Lee. Fuzzy labeled private set intersection with applications to private real-time biometric search. In *USENIX Security*, pages 911–928, 2021.

[VRMÖ17] Cédric Van Rompay, Refik Molva, and Melek Önen. A leakage-abuse attack against multi-user searchable encryption. *PoPETS*, 2017.

[WGL00] Chung Kei Wong, Mohamed Gouda, and Simon S Lam. Secure group communications using key graphs. *IEEE/ACM transactions on networking*, 8(1):16–30, 2000.

[Wil91] David Williams. *Probability with martingales*. Cambridge university press, 1991.

[WLD+17] Guofeng Wang, Chuanyi Liu, Yingfei Dong, Hezhong Pan, Peiyi Han, and Binxing Fang. Query recovery attacks on searchable encryption based on partial knowledge. In *SecureComm*, pages 530–549. Springer, 2017.

[WNL+14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *ACM CCS*, pages 215–226, 2014.

[WP21] Yun Wang and Dimitrios Papadopoulos. Multi-user collusion-resistant searchable encryption with optimal search time. In *AsiaCCS*, pages 252–264, 2021.

[WP23] Yun Wang and Dimitrios Papadopoulos. Multi-user collusion-resistant searchable encryption for cloud storage. *IEEE Transactions on Cloud Computing*, 2023.

[WZC+24] Ruochen Wang, Jun Zhou, Zhenfu Cao, Xiaolei Dong, and Kim-Kwang Raymond Choo. Updatable private set intersection with forward privacy. *IEEE Transactions on Information Forensics and Security*, 19:8573–8586, 2024.

[XLW25] Fei Xiao, Chunyang Lv, and Jianfeng Wang. Fair server-aided multiparty private set intersection from okvs and oprf. In *AsiaCCS*, pages 136–148, 2025.

[Yeo23] Kevin Yeo. Cuckoo hashing in cryptography: Optimal parameters, robustness and applications. In *CRYPTO*, pages 197–230. Springer, 2023.

[YWRL10] Shucheng Yu, Cong Wang, Kui Ren, and Wenjing Lou. Attribute based data sharing with attribute revocation. In Dengguo Feng, David A. Basin, and Peng Liu, editors, *ASIACCS 2010*, pages 261–270. ACM, 2010.

[ZCL+24] Cong Zhang, Yu Chen, Weiran Liu, Liqiang Peng, Meng Hao, Anyu Wang, and Xiaoyun Wang. Unbalanced private set union with reduced computation and communication. In *ACM CCS*, pages 1434–1447, 2024.

A Tabulation Hashing for static multimaps

In this appendix we present a mechanism for saving on bandwidth in the case of a static dataset. For a key that matches ℓ documents, the solution presented in Section 4 requires $\ell \cdot s$ calls to the underlying PIR in $\texttt{ValMap}$ (and s calls to $\texttt{VolMap}$ and ℓ calls to $\texttt{DocMap}$).

Recall, for key and $\texttt{Vol}$, the $\mathsf{MKSE.Search}$ calls $\texttt{UOKVS}.\mathsf{Decode}(\mathsf{key}, 1)$, ..., $\texttt{UOKVS}.\mathsf{Decode}(\mathsf{key}, \texttt{Vol})$ (abusing notation and only showing the client input). The client uses the hashes, $H_1, ...H_s$ to find the positions for each $1 \leq i \leq \texttt{Vol}$, this set of positions is

$$\mathsf{PIR\ Locations} = \left\{ \begin{array}{c} H_1(F(\mathsf{KOwn}_{\mathsf{PRF},\mathsf{DIP}}, (\mathsf{key}, 1))), ..., \\ H_1(F(\mathsf{KOwn}_{\mathsf{PRF},\mathsf{DIP}}, (\mathsf{key}, \texttt{Vol}))), ..., \\ H_s(F(\mathsf{KOwn}_{\mathsf{PRF},\mathsf{DIP}}, (\mathsf{key}, 1)), ..., \\ H_s(F(\mathsf{KOwn}_{\mathsf{PRF},\mathsf{DIP}}, (\mathsf{key}, \texttt{Vol}))) \end{array} \right\}.$$

The client inputs these positions into $\mathsf{PIR.Query}$ (and requests the entire stash).

Review of Tabulation Hashing For an input (x_L, x_R), the tabulation hashing [PT12, Tho17] hashes each component separately and adds the result: $H(x_L, x_R) = H_L(x_L) + H_R(x_L)$ where the above addition is modulo the number of possible positions.

Tabulation Hashing with Shift Friendly PIR

Shift Friendly We additionally say that a PIR is shift friendly if there is an additional pair of algorithms $\mathsf{Shift}, \mathsf{Rec}_\mathsf{S}$ such that $\forall i, j \in [1, \mu]$, define $k := i + j \mod \mu$,

$$\Pr\left[\mathcal{DB}' = \mathcal{DB}[k] \middle| \begin{array}{r} \tilde{\mathcal{DB}} \leftarrow \mathsf{Setup}(\mathcal{DB}) \\ (\mathsf{q}, \mathsf{st}) \leftarrow \mathsf{Query}(i) \\ (\mathsf{q}', \mathsf{st}) \leftarrow \mathsf{Shift}(j, \mathsf{q}) \\ \mathsf{ans} \leftarrow \mathsf{Resp}(\tilde{\mathcal{DB}}, \mathsf{q}') \\ \mathcal{DB}' \leftarrow \mathsf{Rec}_\mathsf{S}(\mathsf{st}, \mathsf{q}, j, \mathsf{ans}) \end{array}\right] > 1 - \mathtt{ngl}(\lambda).$$

Suppose PIR is *shift friendly* and we replace the set of PIR positions with the following:

$$\mathsf{PIR\ Locations} = \left\{ \begin{matrix} H_{1,L}(F(\mathsf{KOwn}_{\mathsf{PRF},\mathsf{DIP}}, \mathsf{key})) + H_{1,R}(i), ..., \\ H_{s,L}(F(\mathsf{KOwn}_{\mathsf{PRF},\mathsf{DIP}}, \mathsf{key})) + H_{s,R}(i) \end{matrix} \right\}_{i=1}^{\mathtt{Vol}}.$$

Since the S learns the Vol as a result of MUSSE.Search, the client can send queries created as:

$$(\mathsf{q}_{1,L}, \mathsf{st}) \leftarrow \mathsf{PIR.Query}(H_{1,L}(F(\mathsf{KOwn}_{\mathsf{PRF},\mathsf{DIP}}, \mathsf{key}))), ...,$$
$$(\mathsf{q}_{s,L}, \mathsf{st}) \leftarrow \mathsf{PIR.Query}(H_{s,L}(F(\mathsf{KOwn}_{\mathsf{PRF},\mathsf{DIP}}, \mathsf{key}))).$$

S then creates

$$\mathsf{PIR\ Locations} = \left\{\mathsf{Query}_{1,i} \leftarrow \mathsf{Shift}(\mathsf{Query}_{j,L}, H_{j,R}(i))\right\}_{i=1,j=1}^{\mathtt{Vol},s}.$$

This whole set of created positions along with the Vol is returned to C. In the above, public hash functions are crucial.

Unfortunately, this technique does not work with moving data around the PIR array. The S can check if accessed positions ever differ by $H_{j,R}(j)$, if so it is likely that the moved items correspond to the same key. However, we present the technique for use in static multimaps. Below we show that common PIRs are shift friendly.

Summary of Performance In the worst case, tabulation hashing can increase the stash size in Cuckoo hashing. We show this does not happen, we perform similar experiments to Figure 1 inserting items up to full load, pretending that tabulation based insertion is secure. Figure 13 shows that tabulation hashing has little effect on stash size for RW and a slight increase for BFS.

A.1 Index Permutation in DEPIR

Definition 6. *Let $\boldsymbol{v} = (v_0, v_1, \cdots, v_{n-1}) \in \mathbb{R}^\psi$ be a vector over a ring $\mathbb{R}$. The circular right shift (also called a cyclic right rotation) of $\boldsymbol{v}$ by k positions, denoted by $Rot_k(\boldsymbol{v})$, is defined as the vector $\boldsymbol{v}' = (v'_0, v'_1, \cdots, c'_{\psi-1})$, where:*

$$v'_j = v_{(j-k) \mod \psi}, \text{ for all } j \in \{0, 1, \cdots, \psi - 1\}$$

When vectors are represented as polynomials in the ring $\mathbb{R}[x]/(x^\psi - 1)$, this operation corresponds to multiplying the polynomial by $x^{-k} \mod (x^\psi - 1)$. That is, if:

$$\boldsymbol{v}(x) = \textstyle\sum_{i=0}^{\psi-1} v_i x^i,$$

then the circular right shift of $\boldsymbol{v}$ by k positions yields:

$$\boldsymbol{v}'(x) = x^{-k} \cdot \boldsymbol{v}(x) \mod (x^\psi - 1).$$

In the DEPIR protocol, circular right shift plays a central role in making the protocol shift friendly. Specifically, during the query generation phase:

1. The client takes the desired query index $i \in [N]$ written in base-d representation:

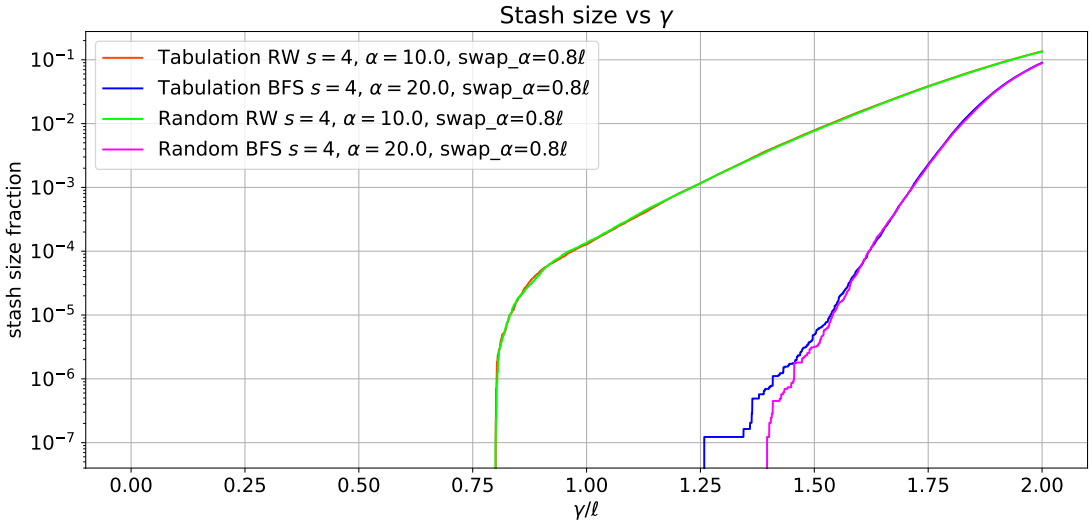

Figure 13: Experimental stash size growth comparison for Random Hashing and Tabulation hashing-based scheme.

$$i = (i_1, \cdots, i_m) \text{ for } m = log_d N$$

2. Each digit i is now encrypted into individual ciphertexts: $\{ct_1, \cdots, ct_m \in R\}$. For all $i = (i_1, \cdots, i_m) \in [N]$, there is a unique polynomial $f_{DB}(X_1, \cdots, X_m)$ such that: $f_{DB}(i_1, \cdots, i_m) = DB[i]$. The resulting ciphertexts encode the index $i' = Rot_k(i)$, that is, the base-d digits of i' are cyclic permutations of those in i.

3. The transformation enables the PIR protocol to be evaluated over the shifted ciphertexts. The server computes:
$$f_{DB}(ct'_1, \cdots, ct'_m) = Enc(DB[i']),$$
where f_{DB} is a polynomial that evaluates to $DB[i']$ at the shifted index i'.

A.2 Index Permutation in FrodoPIR

Similar to DEPIR, circular right shift enables query permutation of the client without revealing the target index in FrodoPIR. Specifically, during the query generation phase:

1. The client selects an index $i \in [\mu]$ that it wishes to query. A query vector $f_i = (0, \cdots, 0, \frac{q}{p}, 0, \cdots, 0)$ is formed such that the only non-zero entry is at position i, which is scaled by $\frac{q}{p}$, as per the scheme parameters.

2. To permute the i^{th} position, the client applies a circular right-shift by k positions to the vector f_i, resulting in $f' = Rot_k(f_i)$, such that, the non-zero entry originally at position i now appears at position $(i + k) \mod N$.

3. The shifted vector f' is then used in the protocol in place of f_i, ensuring that the target index is not revealed.

To overcome this, we consider PIRs that are *shift-friendly*, meaning the server can obliviously change the searched position. **For such PIRs, we show how to combine tabulation hashing and shift-friendly PIR, and sharding to reduce overall search time from 32s to 1.2s.** We replace the hashes in our Cuckoo hashing scheme with tabulation hashes [PT12] where instead of $H(\mathsf{Id}_{\mathsf{key},i})$, the relevant positions are $H(\mathsf{Id}_{\mathsf{key}}) + H'(i)$ for $1 \leq i \leq \mathtt{Vol}$ for hash functions H, H'.

We show that FrodoPIR [DPC23] and DEPIR [LMW23] are shift friendly. This allows the server to obliviously shift $\mathsf{Query}(H(\mathsf{Id}_{\mathsf{key}}))$ into $\mathsf{Query}(H(\mathsf{Id}_{\mathsf{key}}) + H'(i))$. This allows the server to simultaneously craft a response for all $(\mathsf{Id}_{\mathsf{key}}, i)$ from just $\mathsf{Query}(H(\mathsf{Id}_{\mathsf{key}}))$. There are negative results on the collision probability of tabulation hashing in comparison to random hashing [Tho17]. However, in our testing, this change does not harm the quality of packing or the number of documents that cannot be packed. (Many packing-based techniques have a small stash; in our experiments, the stash size never exceeds 200.)

B Multi Owner and Hint Updates in FrodoPIR

Multi Owner Support in FrodoPIR Here we describe how to issue a single query to multiple owners in FrodoPIR and to efficiently create hint updates. The server Setup and Preprocessing steps are as per the original protocol. The only exception is that $\mathbf{A} \in \mathbb{Z}_q^{\psi \times \mu}$ generated by a PRG, is jointly shared by owners. In more detail:

1. For each owner $\mathtt{Ow} \in \mathtt{Owners}$, an encoded database $\mathbf{D}^{\mathtt{Ow}} \in \mathbb{Z}_p^{\mu \times \rho}$ and a vector $\mathbf{M}^{\mathtt{Ow}} \leftarrow \mathbf{A} \cdot \mathbf{D}^{\mathtt{Ow}} \in \mathbb{Z}_q^{\psi \times \rho}$ are created. We use $\mathbf{M}$ to refer to the concatenation of such $\mathbf{M}^{\mathtt{Ow}} \in \mathbb{Z}_q^{n \times \psi \times \rho}$.

2. In the preprocessing phase, the client downloads all $\mathbf{M} \in \mathbb{Z}_q^{n \times \psi \times \rho}$. For each query, the client sample a vector $\mathbf{s} \leftarrow (\chi)^{\psi}$ and $\mathbf{e} \leftarrow (\chi)^{\mu}$. It computes $\mathbf{b} \in \mathbb{Z}_q^{\mu}$ and a matrix $\mathbf{c} \in \mathbb{Z}_q^{n \times \rho}$, where
$$\mathbf{b} \leftarrow \mathbf{s}^T \cdot \mathbf{A} + \mathbf{e}^T,$$
$$\mathbf{c}^{\mathtt{Ow}} \leftarrow \mathbf{s}^T \cdot \mathbf{M}^{\mathtt{Ow}} \in \mathbb{Z}_q^{\rho}.$$

3. In the query phase, for each intended query index $j \in [\mu]$, the client generates a query vector $\mathbf{f} \in \mathbb{Z}_q^{\mu}$, where $\mathbf{f} = (0, \cdots, 0, q/p, 0 \cdots, 0)$. Next, it computes the final query as $\tilde{\mathbf{b}} \leftarrow \mathbf{b} + \mathbf{f} \in \mathbb{Z}_q^{\mu}$, and sends $\tilde{\mathbf{b}}$ to server.

4. As a response, the server computes a matrix $\tilde{\mathbf{c}}$, where each row $\tilde{\mathbf{c}}^{\mathtt{Ow}}$ is the dot product of $\tilde{\mathbf{b}} \cdot \mathbf{D}^{\mathtt{Ow}}$.

5. In post-processing, the client receives matrix $\tilde{\mathbf{c}}$, and outputs $\mathbf{x} \leftarrow \lfloor \tilde{\mathbf{c}} - \mathbf{c}^{\mathtt{Ow}} \rceil \in \mathbb{Z}_p^{n \times \rho}$.

The client can store $\mathbf{s}$ and receive $\mathbf{M}$ as part of the response from the query and use this to compute $\mathbf{c}^{\mathtt{Ow}}$. We believe the tradeoff between storing $\mathbf{M}$ between queries and sending it is application dependent.

Efficient Hint Updates in FrodoPIR We now instantiate $\mathsf{UpdateDB}$ and $\mathsf{UpdateHint}$ from Definition 2 for FrodoPIR. Recall that during preprocessing, the client downloads the compressed database representation $\mathbf{M} = \mathbf{A} \cdot \mathbf{D} \in \mathbb{Z}_q^{\psi \times \rho}$. The matrix $\mathbf{M}$ serves as the client hint.

Suppose the database is updated from $\mathbf{D}$ to $\mathbf{D}'$. Let $\Delta = \mathbf{D}' - \mathbf{D}$ denote the database update. Since $\mathbf{M} = \mathbf{A} \cdot \mathbf{D}$, the updated compressed database representation is

$$\mathbf{M}' = \mathbf{A} \cdot \mathbf{D}' = \mathbf{A}(\mathbf{D} + \Delta) = \mathbf{M} + \mathbf{A}\Delta.$$

The server executes $\mathsf{UpdateDB}$ by computing the update token $\delta = \mathbf{A}\Delta$. The update token ($\delta$) is transmitted to clients holding the corresponding hint. Upon receiving δ, the client executes $\mathsf{UpdateHint}$ and updates its stored hint as $\mathbf{M}' \leftarrow \mathbf{M} + \delta$, since

$$\mathbf{M} + \mathbf{A}\Delta = \mathbf{A}(\mathbf{D} + \Delta) = \mathbf{A}\mathbf{D}' = \mathbf{M}'.$$

The cost of computing this update for α updated positions is $\alpha \cdot \psi \cdot \rho$ total multiplications in $\mathbb{F}_q$ this is faster than naive recomputation by a factor of μ/α. The size of the hint update is $\psi \rho \log q$.