

Incremental Keyword Private Information Retrieval from d -ary Segmented Cuckoo Filters [★]

Bao Ninh^(✉) 

Orochi Network

`bao.ninh@orochi.network`

Abstract. We introduce RisePIR, the first preprocessing keyword private information retrieval (PIR) scheme that absorbs `insert`, `update`, and `delete` on its key-value store at a cost proportional to the number of mutations alone. Where the static state-of-the-art schemes ChalametPIR and KPIR^{index} re-run their preprocessing from scratch on every change, RisePIR patches its preprocessing in place and stays as practical as both in the online phase. We give a generic construction of incremental preprocessing keyword PIR from any updatable preprocessing index PIR, an index PIR that efficiently supports in-place `update`. The construction rests on our d -ary Segmented Cuckoo Filter, a cuckoo filter that confines the j -th candidate bucket of every key to the j -th of d segments and stays dynamic under partial-key cuckoo hashing.

Instantiating the construction over FrodoPIR and SimplePIR, each equipped with the entry-level hint patch of iSimplePIR, gives RisePIR-F and RisePIR-S, both secure under decisional LWE. On a store of about one million keys, both variants patch their hint in place in 0.06–4.7 ms per mutation, five to six orders of magnitude faster than the 56–686 s full re-run of the preprocessing phase. Their online phase stays competitive with ChalametPIR and KPIR^{index} in both asymptotic and concrete cost. We deploy RisePIR-S as a privacy-preserving `eth_getBalance` service over the 2.05×10^8 accounts of Ethereum mainnet, so that a wallet reads a balance without revealing to its RPC provider which account it holds. The running service answers a private query in 0.69 s and absorbs each block’s changes in 5.6 ms, well inside the 12 s block interval. The filter and RisePIR ship as an open-source Rust implementation, together with the benchmark harness behind every number in this paper.

Keywords: Private Information Retrieval · Keyword PIR · Cuckoo filters · Incremental preprocessing · Lattice-based cryptography

1 Introduction

Private Information Retrieval (PIR) [13] allows a client to download an item from a public database held by an untrusted server while hiding *which* item it

[★] This paper is a full version of the paper accepted at the 25th International Conference on Cryptology and Network Security (CANS 2026). Compared with the conference version, this version includes full proofs of the stated theorems and lemmas and provides additional details on parameter selection.

retrieves. It has grown into a practical tool for privacy-preserving systems such as anonymous communication [29,23], Certificate Transparency auditing [22], and private Safe Browsing [14].

In most PIR schemes, the database is treated as an N -entry array and the client fetches the i -th entry without revealing the index i to the server, hence this setting is called *index PIR*. However, most databases are keyed rather than positional, so the client knows the key of the value it wants but not its position in the server’s database. For example, a breach-checking service is queried by a password hash, and a contact-discovery service by a phone number. *Keyword PIR*, introduced by Chor et al. [12], targets this scenario, allowing the client to retrieve the value bound to a key of its choice while hiding which key was queried.

The two state-of-the-art single-server keyword PIR schemes today, **ChalametPIR** [10] and $\text{KPIR}^{\text{index}}$ [21], answer a query against a database of one million keys in well under a second, an order of magnitude more efficient than the prior keyword PIR schemes [32,26,3]. At that speed, both are practical for real applications. However, this practicality holds only as long as the database never changes. Neither scheme can efficiently handle even a single **insert**, **update**, or **delete** of a key-value pair, and since the databases seen in real use are rarely static, these schemes cannot in practice be deployed where they are most needed. We aim to build a keyword PIR scheme that not only answers queries as efficiently as **ChalametPIR** and $\text{KPIR}^{\text{index}}$ but also handles database changes at low cost — an incremental keyword PIR scheme — and so bring keyword PIR to real-world applications.

The root cause. **ChalametPIR** and $\text{KPIR}^{\text{index}}$ owe their efficiency to the same design, and that design is also what keeps them from handling database changes. Neither works directly on the original key-value store. Instead, both operate on a *relaxed database* that flattens the key-value store into an indexable database, one in which the client can determine on its own the positions it must retrieve to reconstruct the value of a key. This flattening is what lets a keyword PIR scheme reduce seamlessly to an index PIR, and a low-overhead reduction over an efficient index PIR yields an efficient keyword PIR, exactly what **ChalametPIR** and $\text{KPIR}^{\text{index}}$ achieve. However, in both schemes the relaxed database is fixed once built. **ChalametPIR** builds it from a Binary Fuse Filter [20] in one pass over the whole key set, with no way to add or remove a key afterward. $\text{KPIR}^{\text{index}}$ builds it from an approximate key-to-index mapping [18] that requires the database to be sorted by key. An **insert**, **update**, or **delete** therefore forces the server to rebuild the relaxed database from scratch.

Worse, the rebuild does not stop at the relaxed database. Both schemes stand on a preprocessing index PIR based on Learning With Errors (LWE), **ChalametPIR** on FrodoPIR [14] and $\text{KPIR}^{\text{index}}$ on SimplePIR [22]. Such an index PIR runs a heavy one-time preprocessing phase that moves the bulk of the computation offline, and that is what keeps the online phase practical. The preprocessing phase derives a hint from the relaxed database, and that hint is bound to the database it came from. In **ChalametPIR** and $\text{KPIR}^{\text{index}}$, every

change to the key-value store rebuilds the relaxed database, and the hint no longer matches. The server must then re-run the whole preprocessing phase, the heavy one-time cost now paid all over again, and every client must download the fresh hint. This is why neither scheme can handle database changes efficiently.

Toward a solution. By contrast, the LWE-based index PIR schemes beneath them already support an evolving database. Their hint is a linear image of the database, so changing a few entries moves the hint by the image of that change alone, and the hint is patched in place at a cost proportional to the number of changed entries rather than to the database size [22,25]. A keyword PIR scheme should be able to inherit this incrementality from the index PIR beneath it. In *ChalametPIR* and *KPIR^{index}* it cannot, because their relaxed database stops incrementality from reaching the keyword layer. Every change to the key-value store rebuilds the relaxed database from scratch, so the number of changed entries is the entire relaxed database, and the cheap hint patch collapses into a full re-run of the preprocessing phase.

Looked at from either side, the problem is the same. Going down from the keyword PIR to the index PIR, a change is costly because the relaxed database must be rebuilt from scratch. Going up from the index PIR to the keyword PIR, its cheap hint patch never reaches the keyword layer, because the relaxed database is rebuilt in full. Both directions meet at one place, the static relaxed database, and that is where the solution belongs. Since the index PIR already patches its hint in place, the keyword PIR will inherit this incrementality as soon as the reduction beneath it stops inflating small changes into large ones. The missing piece is therefore a reduction from keyword PIR to index PIR that keeps the number of changed entries in the relaxed database proportional to the number of mutations in the key-value store, rather than to the store’s total size. With such a reduction, a handful of mutations in the store produces only a handful of changed entries in the relaxed database, the index PIR patches its hint at that same small cost, and the preprocessing phase is no longer paid in full on every change. This paper builds that reduction.

1.1 Our contributions

d -ary Segmented Cuckoo Filter. In Section 4 we introduce the d -ary Segmented Cuckoo Filter (SCF), a cuckoo filter [16,37] whose table is cut into d equal segments with the j -th candidate bucket of every key confined to segment j . The bucket layout itself is not new, going back to d -left hashing [34,8]. What is new is the *partial-key* cuckoo hashing that runs inside it, in which an eviction moves a fingerprint without needing the key that owns it, so the filter stays dynamic under the segmentation constraint. The layout cheapens the reduction from keyword PIR to index PIR, and partial-key eviction keeps every **insert**, **update**, and **delete** to $O(1)$ bucket overwrites at positions fixed in advance, exactly the property the missing piece above calls for. The SCF is built for this incremental keyword PIR construction, though the filter itself may find broader use. Set beside the standard cuckoo filter, it has two advantages.

- Built on the SCF, the generic reduction from keyword PIR to index PIR is a factor d cheaper than the same reduction over a standard cuckoo filter [3], since each index PIR query now touches only its own segment instead of scanning the whole table (see Section 2).
- As a standalone filter, the SCF reaches a slightly higher load factor than the standard filter in every configuration we test (Table 2), and runs $1.3\text{--}1.8\times$ faster at arity 3 while staying within about 10% of the standard filter at arities 2 and 4.

Incremental keyword PIR. In Section 5 we give a generic construction of incremental preprocessing keyword PIR from any updatable preprocessing index PIR, using the key-value form of the SCF, the KV-SCF of Section 5.1, as its relaxed database. To our knowledge this is the first (single-server) incremental keyword PIR construction. The KV-SCF flattens the key-value store into d index PIR databases, one per segment, so a keyword query becomes d index PIR queries, one per segment, and a mutation becomes a sparse, position-predictable batch of index PIR updates. The construction therefore needs only an index PIR that efficiently supports `update`, which we call an updatable preprocessing index PIR, a weaker requirement than the full `insert`, `update`, and `delete` support of an incremental index PIR. The keyword scheme talks only to the index PIR interface, and Theorem 1 shows that it inherits the correctness and security of its index PIR and becomes incremental from that index PIR’s updatability.

RisePIR realizes the construction in two variants, RisePIR-F over FrodoPIR and RisePIR-S over SimplePIR. In Section 6 we first equip FrodoPIR and SimplePIR with the entry-level hint patch of iSimplePIR [25], making each an updatable preprocessing index PIR (Lemma 5), then feed them to the generic construction. Both variants are provably secure under decisional LWE. Section 7 evaluates them against ChalametPIR and KPIR^{index}.

- Asymptotically, RisePIR matches the offline and online computation of the state-of-the-art (non-incremental) keyword PIR schemes ChalametPIR and KPIR^{index}, while paying a mutation cost proportional to the number of mutations alone (Table 1).
- Concretely, on a store of about one million keys, RisePIR absorbs a mutation in 0.06–4.7ms, five to six orders of magnitude faster than the 56-to-686-second full rebuild, while its online cost stays competitive with both ChalametPIR and KPIR^{index} (Tables 3 and 4).
- RisePIR also reduces to either FrodoPIR or SimplePIR, whereas ChalametPIR is tied to FrodoPIR and KPIR^{index} to SimplePIR.

In Section 8 we deploy RisePIR-S as a privacy-preserving `eth_getBalance` service covering the 2.05×10^8 Ethereum mainnet accounts with a nonzero balance. The live service answers a private query in 0.69s and absorbs a block’s 433 mutations in 5.6ms, publishing a 10.8kB delta for clients to sync.

The SCF and RisePIR ship as an open-source Rust implementation, together with the benchmark harness behind every number reported in this paper.

Table 1. Communication ($\mathbb{Z}_q$ elements) and computation ($\mathbb{Z}_q$ ops) cost of RisePIR against ChalametPIR and $\text{KPIR}^{\text{index}}$ across the Offline, Online, and Mutation phases, over relaxed databases of equal size with N slots, each slot w elements wide. n is the LWE dimension, τ the number of mutation operations, and (d, b) the KV-SCF arity and bucket size. A $\mathbb{Z}_q$ op is one addition and one multiplication in $\mathbb{Z}_q$. Rebuild means re-running the Offline phase.

Scheme	Offline phase		Online phase			Mutation phase
	Hint ($\mathbb{Z}_q$ el.)	Comp. ($\mathbb{Z}_q$ ops.)	Query ($\mathbb{Z}_q$ el.)	Resp. ($\mathbb{Z}_q$ el.)	Comp. ($\mathbb{Z}_q$ ops.)	
ChalametPIR	nw	nNw	N	w	Nw	Rebuild
$\text{KPIR}^{\text{index}}$	$n\sqrt{Nw}$		$\sqrt{Nw}$	$\sqrt{Nw}$		
RisePIR-F (<i>ours</i>)	$ndbw$		N/b	dbw		Hint-patch $\Theta(\tau w)$ $\mathbb{Z}_q$ el. $\Theta(n\tau w)$ $\mathbb{Z}_q$ ops.
RisePIR-S (<i>ours</i>)	$n\sqrt{Nwd}$		$\sqrt{Nwd}$	$\sqrt{Nwd}$		

2 Technical Overview

One spirit runs through this work: we care about the concrete cost of the resulting keyword PIR scheme as much as its asymptotic cost, since our aim is a scheme practical enough for real-world applications. That spirit is what rules out several natural-looking alternatives, and what makes the design of the d -ary SCF, together with the other choices we make below, non-trivial.

Most keyword PIR schemes, including the state-of-the-art ChalametPIR and $\text{KPIR}^{\text{index}}$, are built on top of index PIR. As index PIR grew more efficient [14,22], keyword PIR inherited that efficiency and became practical [10,21]. We follow the same approach for our incremental keyword PIR scheme, aiming to match the online efficiency of the state of the art. Other approaches exist that do not build on an index PIR, such as the constant-weight PIR [26], but they run well behind the index-PIR schemes in practice, so we leave them aside.

Our goal is a keyword PIR scheme that answers each query as efficiently as ChalametPIR and $\text{KPIR}^{\text{index}}$, but absorbs database evolution at far lower cost. On a key-value store of N pairs, ChalametPIR and $\text{KPIR}^{\text{index}}$ both rebuild the relaxed database and re-run the preprocessing at cost $\Theta(N)$ on every single mutation, no matter how small. What we aim at instead is a scheme that absorbs τ mutations at cost $O(\tau)$, independent of N .

The LWE-based index PIR beneath, FrodoPIR and SimplePIR, already supports a hint update whose cost scales with the number of changed database entries alone, at concrete cost small enough for practical deployment [25]. The missing piece is a relaxed database whose number of changed entries is proportional to the number of mutations on the store, with a small blow-up factor. The rest of this section walks the intuition that leads to such a relaxed database, and arrives at the d -ary SCF, a non-trivial construction that meets the requirement.

We start from Celi and Davidson’s framework for building the relaxed database, which they call the *key-value filter*, from a fingerprint-based filter. The framework simply extends the fingerprint-based filter to store the fingerprint of the key concatenated with its associated value, $\text{fp}(k) \parallel v$, instead of $\text{fp}(k)$ alone, while keeping the probing mechanism unchanged. As a consequence, the key-value filter inherits every property of the underlying fingerprint-based filter. Therefore instantiating the framework with a dynamic fingerprint-based filter, which can absorb the mutation in $O(1)$ cells, gives us a relaxed database that meets our requirement.

Instantiating the framework with an existing dynamic fingerprint-based filter [16,37,36,9,7,31], however, makes a keyword query expensive. These filters are OR-probing, so a key’s item sits whole in one of its d candidate buckets and only the fingerprint test on the retrieved bucket reveals which one. The client therefore cannot target the bucket that holds the item, and must retrieve all d candidates privately and run the test itself. Since the candidate buckets lie scattered anywhere in the table, one keyword query ends up costing d full-database index PIR queries instead of one.

It is instructive to see why **ChalametPIR** escapes this cost. In a Binary Fuse Filter no cell holds an item whole. Instead, the filter follows the XOR-probing mechanism, where each key is hashed to a small fixed set of cells, every one of those cells contributes a share, and a filter lookup reconstructs the stored entry by XOR-ing the shares together. **ChalametPIR** deliberately encodes this filter over $\mathbb{Z}_p$, replacing the XOR with addition, so that the reconstruction becomes a linear function over $\mathbb{Z}_p$, which the additively homomorphic **FrodoPIR** can evaluate in encrypted form. The whole keyword query therefore collapses into a single full-database query. No such shortcut exists for OR-probing, since selecting the one matching bucket is not a linear operation, and summing the d candidate buckets would only bury the true entry under the unrelated occupants of the others. Therefore, the d individual retrievals are unavoidable for a dynamic filter, and the only lever left is the cost of each.

The key-value filter we build on the d -ary Segmented Cuckoo Filter, the KV-SCF, keeps the d index PIR queries that a keyword query needs but makes each one cheap. We cut the table into d equal segments and confine the j -th candidate of every key to the j -th segment, so segment j becomes a self-contained index PIR sub-database holding $1/d$ of the buckets. A keyword query still issues d index PIR queries, one per segment, but query j now reads only segment j , and the d of them together touch each table entry exactly once, the work of a single full-database index PIR. Figure 1 illustrates the construction for $d = 2$, where a query for key k_9 becomes two index PIR queries, one per segment.

Segmentation cheapens the keyword query, and because the KV-SCF stays dynamic it cheapens change as well. The filter is position-predictable, so an **insert**, **update**, or **delete** on the key-value store rewrites only $O(1)$ buckets at positions fixed in advance, and a mutation becomes a sparse, position-predictable batch of index PIR updates. Treating each segment as one index PIR database, we build incremental keyword PIR generically from any updatable index PIR

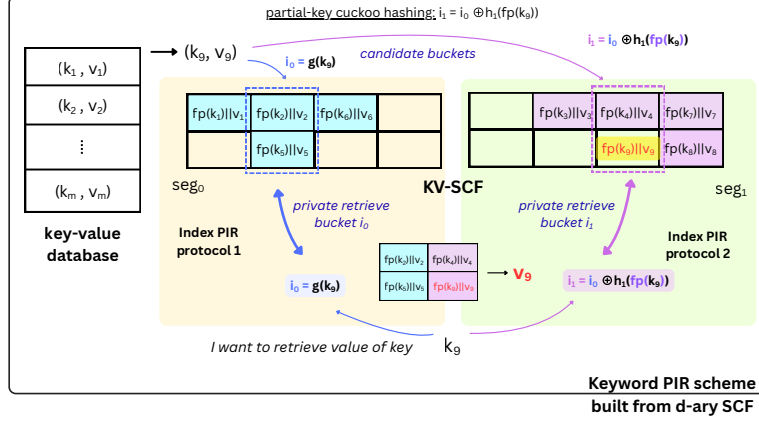


Fig. 1. Overview of a keyword PIR scheme built from the d -ary SCF and an index PIR, shown for $d = 2$. The KV-SCF flattens the key-value store into d segments, each becoming an index PIR database. A keyword PIR query for a key becomes d index PIR queries to get the candidate buckets of that key, one against each segment.

(Section 5). The keyword scheme talks only to the index PIR interface and inherits its correctness, security, and efficiency. Instantiating that interface with FrodoPIR and with SimplePIR, each equipped with the entry-level hint patch of iSimplePIR, yields RisePIR-F and RisePIR-S, both secure under decisional LWE.

3 Preliminaries

Notation. We write $[n]$ for the index set $\{0, 1, \dots, n-1\}$, λ for the security parameter, and $\perp$ for a distinguished symbol signalling that no value was found; $\lfloor x \rfloor$, $\lceil x \rceil$, and $\text{round}(x)$ round x down, up, and to the nearest integer. For $\mathbf{x} \in \mathbb{Z}_q^m$, we write $\lfloor \mathbf{x} \rfloor_p$ to denote the computation of $\lfloor p/q \cdot \mathbf{x} \rfloor$, where the rounding is applied to each entry of $\mathbf{x}$ individually. We write $x \leftarrow y$ for an assignment. We write $x \xleftarrow{\$} S$ for uniform sampling from a finite set or a distribution, and write $(x_j)_{j \in [d]}$ for the ordered tuple $(x_0, \dots, x_{d-1})$.

3.1 Updatable Preprocessing Index PIR

Our incremental keyword PIR scheme is built on an *Updatable preprocessing Index PIR* (UIPIR), which extends an ordinary preprocessing index PIR with the ability to efficiently handle database **update** operations. We adapt this notion from the Incremental single-server preprocessing Index PIR of iSimplePIR [25] (which supports **insert**, **update**, and **delete** operations) and give its formal definition in Figure 2, organized into three separate phases. The **Offline phase** and **Online phase** comprise the four algorithms (Setup, Query, Answer, Recover),

which together depict an ordinary preprocessing index PIR scheme. The **Mutation phase** adds two algorithms (**DBMutation**, **HintUpdate**), where **DBMutation** evolves the server database under a batch of τ **update** operations and **HintUpdate** patches the hint to match.

This primitive is defined over a database $\text{DB} \in \mathcal{V}^N$, which holds N records indexed by $[N]$ and drawn from a value space $\mathcal{V}$. An update operation is a pair $\text{op} = (i, v)$ that overwrites the record at position $i \in [N]$ with the value $v \in \mathcal{V}$, leaving the length N and every other record unchanged. The *mutation space* $\mathcal{M}_{\text{idx}} := [N] \times \mathcal{V}$ collects all such operations, from which **DBMutation** consumes a batch $\text{op}_1, \dots, \text{op}_\tau$.

Updatable preprocessing Index PIR (UIPIR)

Incremental preprocessing Keyword PIR (IKPIR)

Parameters. A database $\text{DB} \in \mathcal{V}^N$ of N records in a value space $\mathcal{V}$, with mutation space $\mathcal{M}_{\text{idx}} := [N] \times \mathcal{V}$.

Parameters. A key-value store $\text{DB} \in (\mathcal{K} \times \mathcal{V})^m$ of m pairs with distinct keys, with mutation space $\mathcal{M}_{\text{kw}} := \{\text{insert}, \text{update}, \text{delete}\} \times \mathcal{K} \times \mathcal{V}$.

All algorithms take 1^λ , and those after **Setup** the hint's public parameters, as implicit inputs.

Offline phase.

- **Setup**(DB) $\rightarrow (\text{D}, \text{hint})$: preprocess the database DB into a processed database D and a public hint.

Online phase.

- **Query**(i) $\rightarrow (\text{qu}, \text{st})$: encode the queried index $i \in [N]$ into a query qu and a client state st .
- **Query**(k) $\rightarrow (\text{qu}, \text{st})$: the same for a queried key $k \in \mathcal{K}$.
- **Answer**(qu, D) $\rightarrow \text{ans}$: evaluate the query qu against the processed database D .
- **Recover**($\text{hint}, \text{st}, \text{ans}$) $\rightarrow v$: decode the answer into the value $v \in \mathcal{V}$ ($v \in \mathcal{V} \cup \{\perp\}$ for keyword PIR, $\perp$ if k is absent).

Mutation phase.

- **DBMutation**($\text{D}, (\text{op}_1, \dots, \text{op}_\tau)$) $\rightarrow (\text{D}', \text{trans})$: apply the batch of mutation operations $\text{op}_1, \dots, \text{op}_\tau$ to D , yielding D' and a transcript trans of the changed cells.
- **HintUpdate**($\text{hint}, \text{trans}$) $\rightarrow \text{hint}'$: patch the hint to match D' by replaying trans .

Fig. 2. Syntax of UIPIR and IKPIR; the shaded parts are specific to IKPIR.

A UIPIR scheme must satisfy correctness, security, and updatability, which we formalize below.

Definition 1 (Correctness). Fix a database $\text{DB} \in \mathcal{V}^N$ and a batch of updates $\text{op}_1, \dots, \text{op}_\tau \in \mathcal{M}_{\text{idx}}$, and write $\text{DB}^{(\tau)}$ for the array left after applying the batch to DB in order. A UIPIR scheme is δ -correct if for every such DB and batch and

every index $i \in [N]$,

$$\Pr \left[\tilde{v} \neq \text{DB}_i^{(\tau)} : \begin{array}{l} (\text{D}, \text{hint}) \leftarrow \text{Setup}(\text{DB}) \\ (\text{D}', \text{trans}) \leftarrow \text{DBMutation}(\text{D}, (\text{op}_1, \dots, \text{op}_\tau)) \\ \text{hint}' \leftarrow \text{HintUpdate}(\text{hint}, \text{trans}); \quad (\text{qu}, \text{st}) \leftarrow \text{Query}(i) \\ \text{ans} \leftarrow \text{Answer}(\text{qu}, \text{D}'); \quad \tilde{v} \leftarrow \text{Recover}(\text{hint}', \text{st}, \text{ans}) \end{array} \right] \leq \delta.$$

The honest-but-curious server can choose the initial database, execute every mutation, track the hint, and observe all client queries. Figure 3 grants the adversary $\mathcal{A}$ these exact capabilities in the game $\text{QIND}_{\text{UIPIR}}^{\mathcal{A}}$ through two oracles that share one evolving state and may be interleaved arbitrarily: an update oracle $\mathcal{O}_{\text{upd}}$ that applies a batch of update operations of its choice and returns the mutated database, the transcript, and the refreshed hint, and a challenge oracle $\mathcal{O}_{\text{chal}}$ that, on a pair of indices (i_0, i_1) , returns a query for i_b .

This is the standard publisher deployment of preprocessing PIR [14,25]: the data owner performs every mutation and publishes the resulting transcript, which each client replays on its own hint copy, so a mutation is a public event and the sole secrecy goal is query privacy. The transcript **trans** accordingly need not conceal what it changes, and the game hands it to $\mathcal{A}$ in full; security demands only that a challenge query reveal nothing about the queried index i_b . The keyword game defined next is identical, protecting the queried key while leaving mutations public.

Definition 2 (Security). A UIPIR scheme is (T, ε) -secure if, for all adversaries $\mathcal{A}$ running in time at most T ,

$$\left| \Pr[\text{QIND}_{\text{UIPIR}}^{\mathcal{A}}(1^\lambda) = 1] - \frac{1}{2} \right| \leq \varepsilon.$$

Definition 3 (Updatability). A UIPIR scheme is updatable if, for a batch of τ update operations, the communication and computation costs of the mutation phase are proportional to τ and asymptotically smaller than the cost of a full rebuild via **Setup**.

Security game for UIPIR (IKPIR)

$\text{QIND}_{\text{UIPIR IKPIR}}^{\mathcal{A}}(1^\lambda)$:

1. $\text{DB} \leftarrow \mathcal{A}(1^\lambda); \quad b \xleftarrow{\$} \{0, 1\}$
2. $(\text{D}, \text{hint}) \leftarrow \text{Setup}(\text{DB})$
3. $b' \leftarrow \mathcal{A}^{\mathcal{O}_{\text{upd}}, \mathcal{O}_{\text{chal}}}(\text{D}, \text{hint})$
4. **return** 1 if $b' = b$, else 0

The two oracles share the one evolving state (D, hint) and may be interleaved in any order and called any number of times.

$\mathcal{O}_{\text{upd}}(\text{op}_1, \dots, \text{op}_\tau)$, each $\text{op}_i \in \mathcal{M}_{\text{idk}}$ (resp. $\mathcal{M}_{\text{kw}}$):

- $(\text{D}', \text{trans}) \leftarrow \text{DBMutation}(\text{D}, (\text{op}_1, \dots, \text{op}_\tau))$

```

- hint' ← HintUpdate(hint, trans)
- (D, hint) ← (D', hint') // commit the new shared state
- return (D', trans, hint')

 $\mathcal{O}_{\text{chal}}(i_0, i_1)$ , with  $i_0, i_1 \in [N]$ :
- (qu, st) ← Query( $i_b$ ); return qu

 $\mathcal{O}_{\text{chal}}(k_0, k_1)$ , with  $k_0, k_1 \in \mathcal{K}$ :
- (qu, st) ← Query( $k_b$ ); return qu

```

Fig. 3. Security game (query indistinguishability under mutations) for UIPIR and IKPIR; the shaded parts are specific to IKPIR.

Without the asymptotic clause, the trivial scheme that reruns **Setup** on every mutation and ships the fresh hint would already qualify at no benefit over a static scheme; the clause is what rules it out. The definition constrains the mutation phase alone, the incremental contribution of the primitive, and leaves the online phase at the standard preprocessing query cost.

3.2 Incremental Preprocessing Keyword PIR

An *incremental preprocessing keyword PIR* (IKPIR) extends an ordinary preprocessing keyword PIR with the ability to efficiently handle database **insert**, **update**, and **delete** operations. We give its formal definition in Figure 2 read with the shaded text included, organized into three separate phases. The offline and online phases comprise the four algorithms (**Setup**, **Query**, **Answer**, **Recover**), which together depict an ordinary preprocessing keyword PIR scheme, where **Query** addresses a key $k \in \mathcal{K}$ and **Recover** returns the bound value or $\perp$ when the key is absent. The mutation phase adds two algorithms (**DBMutation**, **HintUpdate**), where **DBMutation** evolves the server database under a batch of τ mutation operations and **HintUpdate** patches the hint to match.

This primitive is defined over a key-value store $\text{DB} \in (\mathcal{K} \times \mathcal{V})^m$, which holds m pairs with distinct keys drawn from a key space $\mathcal{K}$ and a value space $\mathcal{V}$. A mutation operation is a triple $\text{op} = (\text{type}, k, v)$ whose tag $\text{type} \in \{\text{insert}, \text{update}, \text{delete}\}$ fixes the roles of $k \in \mathcal{K}$ and $v \in \mathcal{V}$, where **insert** binds a fresh key k to v , **update** rebinds an existing k to a new v , and **delete** removes k and ignores the value field. The mutation space $\mathcal{M}_{\text{kw}} := \{\text{insert}, \text{update}, \text{delete}\} \times \mathcal{K} \times \mathcal{V}$ collects all such operations, from which **DBMutation** consumes a batch $\text{op}_1, \dots, \text{op}_\tau$.

We accept only *well-formed* batches: an **insert** must target a key that is not yet in the store, while an **update** or **delete** must target a key that already is. This is how a key-value store normally behaves, and the analysis in Section 5.3 relies on it.

An IKPIR scheme must satisfy the same three properties, now measured in the number of pairs m rather than the record count N , which we formalize below.

Definition 4 (Correctness). Fix a store $\text{DB} \in (\mathcal{K} \times \mathcal{V})^m$ and a batch of mutations $\text{op}_1, \dots, \text{op}_\tau \in \mathcal{M}_{\text{kw}}$, and write $\text{DB}^{(\tau)} = \{(k_i, v_i)\}_i$ for the store left after

applying them to DB in order. For a queried key $k \in \mathcal{K}$, let $\tilde{v}$ be the value produced by the experiment of Definition 1 with $\text{Query}(k)$ in place of $\text{Query}(i)$. An IKPIR scheme is (δ, ξ) -correct if for every such DB, batch, and key k ,

$$\begin{aligned} \Pr[\tilde{v} \neq v_i] &\leq \delta && \text{if } k = k_i \text{ for some } i, \\ \Pr[\tilde{v} \neq \perp] &\leq \xi && \text{if } k \notin \{k_i\}_i. \end{aligned}$$

Correctness is two-sided here, bounding a wrong value for a present key by δ and a false hit for an absent one by ξ .

The chosen-keyword game $\text{QIND}_{\text{IKPIR}}^A$ is that same game over a key-value store, its update oracle taking a batch from $\mathcal{M}_{\text{kw}}$ and its challenge oracle taking two keys (k_0, k_1) and returning $\text{Query}(k_b)$. It protects the queried *key* and leaves mutations public, exactly as the index game does. Figure 3 writes both games as one, the shaded parts specific to IKPIR.

Reading Figure 3 with the shaded text included gives the chosen-keyword game $\text{QIND}_{\text{IKPIR}}^A$, the analogue of Definition 2 in which the update oracle takes a batch from $\mathcal{M}_{\text{kw}}$ and the challenge oracle takes two keys (k_0, k_1) and returns $\text{Query}(k_b)$. It protects the queried *key* and leaves mutations public, exactly as the index game does.

Definition 5 (Security). An IKPIR scheme is (T, ε) -secure if, for all adversaries $\mathcal{A}$ running in time at most T ,

$$\left| \Pr[\text{QIND}_{\text{IKPIR}}^A(1^\lambda) = 1] - \frac{1}{2} \right| \leq \varepsilon.$$

Definition 6 (Incrementality). An IKPIR scheme is incremental if, for a batch of τ mutation operations, the communication and computation costs of the mutation phase are proportional to τ and asymptotically smaller than the cost of a full rebuild via Setup.

Without the asymptotic clause, the trivial scheme that reruns Setup on every mutation and ships the fresh hint would already qualify at no benefit over a static scheme; the clause is what rules it out.

This last definition constrains the mutation phase alone, the incremental contribution of the primitive, and leaves the online phase at the standard pre-processing query cost.

Naming convention. With both primitives now defined, we adopt lighter names for the remainder of the paper: we drop the qualifier *preprocessing* and write simply *updatable index PIR* (UIPIR) and *incremental keyword PIR* (IKPIR), consistent with their acronyms, leaving the one-time Setup hint understood.

4 The d -ary Segmented Cuckoo Filter

This section presents the d -ary Segmented Cuckoo Filter (SCF), the primitive we use to build incremental keyword PIR.

4.1 Bucket layout and partial-key cuckoo hashing

In the standard (2-ary) cuckoo filter [16] each item x has two candidate buckets, and its fingerprint $\text{fp}(x)$ is stored in whichever of the two has a free slot. The two candidate buckets are computed by partial-key cuckoo hashing, which sets $i_1 = g(x)$ and $i_2 = i_1 \oplus h(\text{fp}(x))$, so the second bucket is the first XOR-ed with a hash of the fingerprint. Because $\oplus$ is its own inverse, either bucket determines the other from the fingerprint alone, $i_1 = i_2 \oplus h(\text{fp}(x))$, with no need for the original item. This is what makes the filter dynamic, since an insertion that must relocate a stored fingerprint knows only that fingerprint yet can still recover its alternate bucket. The d -ary cuckoo filter [37] extends this to $d > 2$ candidate buckets through a base- d digit-wise XOR.

The SCF keeps partial-key cuckoo hashing but confines each candidate to a fixed region of the table. We partition the n_b buckets into d equal segments of $s = n_b/d$ buckets each and tie the j -th candidate bucket of every item to the j -th segment, so a key occupies exactly one bucket per segment. This one-candidate-per-segment layout comes from d -left hashing [34,8], which the SCF reuses for a fingerprint-based filter.

The SCF's layout gives a two-way relation between a bucket's global index and its position inside a segment. Forwards, a global index i belongs to segment $\lfloor i/s \rfloor$ with local offset $i \bmod s$. Backwards, a local offset $a \in [s]$ in segment j has global index $j \cdot s + a$. Based on this relation, our partial-key cuckoo hashing in the SCF works on the local offsets of the candidate buckets only, because all candidates' offsets now fall in the same range $[s]$, which is convenient for the XOR operation. We take s to be a power of two, so that the XOR of two offsets again lies in $[s]$ and needs no reduction modulo s . Working instead on the global indices would place the candidates in different ranges, where XOR no longer applies seamlessly.

Concretely, letting $a_j \in [s]$ be the offset of the j -th candidate of x inside segment j , the partial-key cuckoo hashing sets:

$$a_0 = g(x), \quad a_j = a_{j-1} \oplus h_j(\text{fp}(x)) \quad (1 \leq j < d),$$

where g maps the item and each h_j maps the fingerprint into $[s]$.

Given any one offset a_j and the fingerprint, the neighbors follow as $a_{j-1} = a_j \oplus h_j(\text{fp}(x))$ and $a_{j+1} = a_j \oplus h_{j+1}(\text{fp}(x))$, and unrolling rebuilds the whole chain. We capture the two directions as a pair of routines, $\text{Candidates}(x) = (a_j)_{j \in [d]}$ returning the candidate offsets of an item and $\text{AltBuckets}(i, \varphi)$ rebuilding the same set from a single bucket i and a fingerprint φ . Both are deterministic, so the client can compute a key's d candidate buckets in advance from the item, while the server, during an eviction, can relocate a displaced fingerprint from its slot alone without ever recovering the key.

All-or-nothing collisions. From the partial-key cuckoo hashing above, the d -ary SCF enjoys a property that makes the later analysis cleaner. Any two items with the same fingerprint either agree on all d candidate buckets or agree on

none, with no partial overlap. We prove this now and use it in Section 5.3 to obtain tighter bounds.

Lemma 1 (Candidate alignment). *Let $x \neq y$ with $\text{fp}(x) = \text{fp}(y)$. Then x and y have the same d candidate buckets, which happens exactly when $g(x) = g(y)$, or they share no candidate bucket at all.*

Proof. Expanding the definition step by step gives $a_j(x) = g(x) \oplus H_j(\text{fp}(x))$, where $H_j := h_1 \oplus \dots \oplus h_j$ and $H_0 := 0$. A bucket index determines its segment, so x and y share a bucket only within one segment j , that is only if $a_j(x) = a_j(y)$. Writing $\varphi := \text{fp}(x) = \text{fp}(y)$, the two offsets are $g(x) \oplus H_j(\varphi)$ and $g(y) \oplus H_j(\varphi)$, so $a_j(x) = a_j(y)$ holds if and only if $g(x) = g(y)$, a condition free of j . The d offsets therefore agree for every j or for none. $\square$

This property is special to the SCF. The standard d -ary filter spreads its candidates across the whole table, so two items can share some candidate buckets without sharing all of them. This all-or-nothing structure is part of what makes the SCF a good fit for reducing keyword PIR to index PIR, since it keeps the collision behavior simple to control. The analysis in Section 5.3 uses Lemma 1 both to bound how ambiguous a lookup can be and to guarantee that an update or deletion can only touch keys sharing the target’s fingerprint, never an unrelated key.

4.2 The filter algorithms

The four algorithms of the SCF appear in Figure 4, read with the shaded text ignored. `Initialize` allocates an empty d -segment table and samples the hash functions $g, h_1, \dots, h_{d-1}$ that determine each item’s candidate buckets. `F.insert` writes a new fingerprint into a free slot of one candidate bucket, or, when every candidate bucket is full, falls back to partial-key cuckoo eviction, relocating a displaced fingerprint. `F.lookup` scans the candidate buckets and reports whether the queried fingerprint is present, and `F.delete` clears the slot that holds it.

The d -ary Segmented Cuckoo Filter (SCF)

The d -ary Segmented Cuckoo Key-value Filter (KV-SCF)

Parameters. The filter F is a table of n_b buckets, partitioned into d equal segments of $s = n_b/d$ buckets (with s a power of two), each bucket holding b slots. A slot is either free or holds an f -bit fingerprint and an ℓ -bit value, and MaxKicks bounds the insertion random walk.

The mutating operations of the KV-SCF (`insert`, `delete`, `update`) additionally output a write log W that records the change of each modified slot.

In this description, we use x for key and v for value.

`KV-SCF.Initialize( $d, b, n_b$ )` $\rightarrow (F, \text{pp})$:

- Allocate an empty table F of n_b buckets, each with b slots of $f + \ell$ bits.

– Sample the fingerprint function $\text{fp} : \mathcal{K} \rightarrow \{0,1\}^f$ and the hash functions $g, h_1, \dots, h_{d-1} : \{0,1\}^* \rightarrow [s]$, all independently, where $s = n_b/d$.
 – Return F and the public parameters $\text{pp} := (d, b, n_b, \text{fp}, g, h_1, \dots, h_{d-1})$.
 $F.\text{insert}(x) \rightarrow \{0,1\}$ $F.\text{insert}(x, v) \rightarrow (\{0,1\}, W)$:
 – Compute the d candidate buckets via $\text{Candidates}(x)$. If one has a free slot, write $\text{fp}(x) \parallel v$ into it, recording the change in W , and return 1.
 – Otherwise evict the fingerprint φ and value ϑ in a random occupied slot, put $\text{fp}(x) \parallel v$ in its place, and reinsert $\varphi \parallel \vartheta$ into a free slot of its own candidate buckets $\text{AltBuckets}(i, \varphi)$, repeating as a random walk of up to MaxKicks steps, recording each rewritten slot's change in W .
 – Return 1 once a free slot is found, or, if the budget is spent, undo every eviction and return 0 with $W = \emptyset$.
 $F.\text{lookup}(x) \rightarrow \{0,1\}$ $F.\text{lookup}(x) \rightarrow v$:
 – Scan the b slots of each candidate bucket $\text{Candidates}(x)$ for $\text{fp}(x)$, and return 1 if some slot matches, else 0 (the matching value, else $\perp$).
 $F.\text{delete}(x) \rightarrow \{0,1\}$ $F.\text{delete}(x) \rightarrow (\{0,1\}, W)$:
 – Find a slot of $\text{Candidates}(x)$ holding $\text{fp}(x)$, clear it, and return 1, or 0 if none matches, recording the cleared slot's change in W .
 $F.\text{update}(x, v) \rightarrow (\{0,1\}, W)$:
 – Find the slot of $\text{Candidates}(x)$ holding $\text{fp}(x)$, overwrite its value field with v , recording the change in W , and return 1; if x is absent, return $(0, \emptyset)$.

Fig. 4. The d -ary Segmented Cuckoo Filter (SCF) and the d -ary Segmented Cuckoo Key-value Filter (KV-SCF); the shaded parts are specific to the KV-SCF.

False-positive rate. For an item x absent from the filter, $F.\text{lookup}(x)$ compares $\text{fp}(x)$ against the at most $d \cdot b$ fingerprints stored in its candidate buckets, each the image of some other item under the fp sampled at `Initialize` and hence an independent uniform f -bit string that matches with probability 2^{-f} . A union bound over the comparisons gives

$$\xi = \Pr[F.\text{lookup}(x) = 1 \text{ for } x \notin F] \leq d \cdot b \cdot 2^{-f}. \quad (1)$$

Segmentation changes where a lookup reads and not how much, so this is the false-positive rate of the standard d -ary cuckoo filter [16,37].

False-negative rate. The SCF has none, for the reason the standard cuckoo filter gives [16]. $F.\text{insert}$ writes one copy of $\text{fp}(x)$ for every present item, duplicates included, and $F.\text{delete}(x)$ removes one copy from x 's candidate buckets, which by Lemma 1 are also the candidate buckets of every item that copy could belong to. Which copy goes is thus immaterial, and as long as only present items are deleted, each item still present keeps a copy where its lookup will find it. The copies stop being interchangeable once a slot also carries a value, and there deletion does become a hazard, one Section 5.3 bounds for the key-value form.

5 Generic Construction of Incremental Keyword PIR

This section constructs IKPIR generically from any UIPIR. The relaxed database is the KV-SCF, the key-value filter that the framework of Celi and Davidson [10] builds from the SCF, and it flattens the key-value store into d index PIR databases, one per segment. The keyword scheme raised over it talks only to the index PIR interface, so its correctness, security, and incrementality reduce to the correctness, security, and updatability of the UIPIR beneath.

5.1 d -ary Segmented Cuckoo Key-value Filter (KV-SCF)

The relaxed database we use in the IKPIR construction is the d -ary Segmented Cuckoo Key-value Filter (KV-SCF), a key-value filter built on the framework of Celi and Davidson [10]. The framework turns any fingerprint-based filter into a key-value store by placing $\text{fp}(k)\|v$ in the slot that would hold $\text{fp}(k)$ alone, leaving the probing mechanism untouched, so the result inherits whatever the underlying filter offers. Applying it to the dynamic SCF yields a dynamic relaxed database. Each segment of the KV-SCF becomes one index PIR database in the IKPIR construction.

Figure 4, read with the shaded text included, gives the algorithms of the KV-SCF. Its `Initialize`, `F.insert`, `F.lookup`, and `F.delete` have the same shape as in the SCF, differing only in that they additionally carry the value alongside the fingerprint. The KV-SCF also provides `F.update`, which rebinds a present key with new value.

Every mutating operation of the KV-SCF also hands back a *write log* W describing the edit it made, one record per slot it disturbed, identifying that slot by its (bucket, slot) position and giving the signed difference between the slot's new and old contents. A successful `update` or `delete` disturbs one slot and logs one record, while an `insert` may walk up to MaxKicks evictions before it settles, for at most $\text{MaxKicks} + 1$ records. MaxKicks is a constant independent of the database size, so $|W| = O(1)$ whichever operation ran, the property Lemma 4 rests on.

5.2 The generic construction of IKPIR

Figure 5 builds IKPIR from any UIPIR, reducing one IKPIR protocol to d UIPIR protocols over databases d times smaller. It uses the KV-SCF as its relaxed key-value database and gives segment j its own UIPIR instance with state (D_j, hint_j) . `Setup` builds those d instances offline and the later phases drive them through the single UIPIR interface, no instance reading or writing another's state. Theorem 1 states what the construction inherits from its index PIR, and Section 5.3 gives the analysis behind it, with the two longer proofs deferred to Appendix A.

Generic construction of IKPIR from UIPIR

Parameters:

- **Database.** A key-value store $\text{DB} = \{(k_i, v_i)\}_{i \in [m]} \in (\mathcal{K} \times \mathcal{V})^m$ of m pairs with distinct keys, over $\mathcal{V} = \{0, 1\}^\ell$.
- **Filter.** Arity d , bucket size b , a fingerprint $\text{fp} : \mathcal{K} \rightarrow \{0, 1\}^f$, and lf , the maximum load factor of the d -ary SCF at this bucket size.
- **Index PIR.** A UIPIR scheme used as a black box and instantiated once per segment.

Offline phase.

$\text{Setup}(\text{DB}) \rightarrow (\text{D}, \text{hint})$:

1. Choose the bucket count n_b with $n_b \cdot b \cdot \text{lf} \geq m$, so the KV-SCF can hold all of DB. Run $(F, \text{pp}) \leftarrow \text{KV-SCF.Initialize}(d, b, n_b)$ and insert every $(k_i, v_i) \in \text{DB}$ via $F.\text{insert}(k_i, v_i)$, aborting with $\perp$ if any insertion returns 0. Let F_j denote segment j of F .
2. For each $j \in [d]$, run $(\text{D}_j, \text{hint}_j) \leftarrow \text{UIPIR.Setup}(F_j)$.
3. Output $\text{D} := (F, \text{pp}, (\text{D}_j)_{j \in [d]})$ and $\text{hint} := (\text{pp}, (\text{hint}_j)_{j \in [d]})$.

From here on, pp is an implicit parameter that both client and server hold throughout the online and mutation phases.

Online phase.

$\text{Query}(k) \rightarrow (\text{qu}, \text{st})$:

1. Compute $(r_0, \dots, r_{d-1}) \leftarrow \text{Candidates}(k)$, the candidate bucket position of k in each segment.
2. For each $j \in [d]$, run $(\text{qu}_j, \text{st}_j) \leftarrow \text{UIPIR.Query}(r_j)$.
3. Output $\text{qu} := (\text{qu}_j)_{j \in [d]}$ and $\text{st} := ((\text{st}_j)_{j \in [d]}, \text{fp}(k))$.

$\text{Answer}(\text{qu}, \text{D}) \rightarrow \text{ans}$:

1. For each $j \in [d]$, run $\text{ans}_j \leftarrow \text{UIPIR.Answer}(\text{qu}_j, \text{D}_j)$.
2. Output $\text{ans} := (\text{ans}_j)_{j \in [d]}$.

$\text{Recover}(\text{hint}, \text{st}, \text{ans}) \rightarrow v$:

1. For each $j \in [d]$, run $\text{bk}_j \leftarrow \text{UIPIR.Recover}(\text{hint}_j, \text{st}_j, \text{ans}_j)$ to get the candidate bucket of segment j .
2. Scan the b slots of the d recovered buckets $\text{bk}_0, \dots, \text{bk}_{d-1}$ for one whose fp field equals $\text{fp}(k)$, and return its value field, or $\perp$ if none matches.

Mutation phase.

$\text{DBMutation}(\text{D}, (\text{op}_1, \dots, \text{op}_\tau)) \rightarrow (\text{D}', \text{trans})$:

1. Apply each $\text{op} = (\text{type}, k, v) \in \mathcal{M}_{\text{kw}}$ to the key-value filter F through the matching $F.\text{insert}$, $F.\text{update}$, or $F.\text{delete}$, collecting their write logs into the batch's change set W .
2. For each segment $j \in [d]$, take the buckets that W touched in that segment and turn each one into a single record overwrite in $\mathcal{M}_{\text{idx}}$ that rewrites the bucket's record with its new contents. Collect these overwrites into the batch ops_j and run $(\text{D}'_j, S_j) \leftarrow \text{UIPIR.DBMutation}(\text{D}_j, \text{ops}_j)$, updating segment j to D'_j and returning its transcript S_j .
3. Output $\text{D}' := (F, \text{pp}, (\text{D}'_j)_{j \in [d]})$ and $\text{trans} := (S_j)_{j \in [d]}$.

$\text{HintUpdate}(\text{hint}, \text{trans}) \rightarrow \text{hint}'$:

1. For each segment $j \in [d]$, patch the hint by $\text{hint}'_j \leftarrow \text{UIPIR.HintUpdate}(\text{hint}_j, S_j)$ with S_j read from $\text{trans} = (S_j)_{j \in [d]}$.

2. Output $\text{hint}' := (\text{pp}, (\text{hint}'_j)_{j \in [d]}).$

Fig. 5. Construct IKPIR from any UIPIR, bridged by the KV-SCF; without its mutation phase the construction is a keyword PIR from any preprocessing index PIR.

Theorem 1 (IKPIR from UIPIR). *The construction of Figure 5, built on any UIPIR, is a correct, secure, and incremental IKPIR, with the precise bounds of Lemmas 2–4.*

In the online phase, a query for k fetches its candidate bucket in each segment with one index PIR query, so a keyword query becomes d index PIR queries, one per segment. The client runs the filter’s lookup rule over the $d \cdot b$ slots it recovers, returning the value beside $\text{fp}(k)$, or $\perp$.

Our central contribution is to make IKPIR handle an evolving key-value store efficiently, by pairing the KV-SCF with a UIPIR. We require only an efficient in-place **update** from the index PIR, since the KV-SCF turns every keyword mutation into **updates** on the indexable database beneath it: an **update** rewrites the one slot holding the key and a **delete** clears it, both in one bucket, while a successful **insert** rewrites at most $\text{MaxKicks} + 1$ slots along its eviction walk, a constant independent of the database size. Every mutation touches $O(1)$ buckets, one index PIR **update** each, so a mutation becomes a sparse, position-predictable batch of index PIR updates. An efficient UIPIR **update** makes IKPIR’s mutation phase efficient, which is precisely its incrementality.

Scope of the incremental guarantee. A KV-SCF that grows too full can fail an **insert**, and the only fix is a one-time rebuild at larger capacity, re-running **Setup** at full preprocessing cost. It amortizes over mutations between rebuilds (Section 7.3), acceptable in practice, not a clean solution in theory. The limitation comes from using only the *updatability* of the index PIR; removing it needs another relaxed database and the *incrementality* of the index PIR, a direction we leave to future work. We henceforth assume every **insert** succeeds.

5.3 Correctness, security, and incrementality

In this section, we establish three lemmas that separately prove the correctness, security, and incrementality of the IKPIR of Figure 5 from the corresponding properties of the UIPIR underneath.

Consider a batch of τ mutations. We write $\text{DB}^{(t)}$ and $F^{(t)}$ for the store and the filter after the first t operations of the batch. We define the key history $\mathcal{H}$ as the set of all keys that either belong to the initial store or are inserted by an **insert** during the batch:

$$\mathcal{H} := \{k_i\}_{i \in [m]} \cup \{k : \text{op}_t = (\text{insert}, k, \cdot) \text{ for some } t \leq \tau\}, \quad (2)$$

$|\mathcal{H}| \leq m + \tau_{\text{ins}}$, where τ_{ins} is the number of **inserts** in the batch.

Lemma 2 (Correctness reduction). *Built on a δ_{idx} -correct UIPIR, the construction of Figure 5 is (δ, ξ) -correct for every well-formed batch of τ mutations in which every *insert* succeeds, with*

$$\delta \leq d \cdot \delta_{\text{idx}} + |\mathcal{H}| \cdot \frac{2^{-f}}{s} \leq d \cdot \delta_{\text{idx}} + \left(1 + \frac{\tau_{\text{ins}}}{m}\right) d \cdot b \cdot 2^{-f}, \quad \xi \leq d \cdot \delta_{\text{idx}} + d \cdot b \cdot 2^{-f},$$

where $\mathcal{H}$ is the key history of Eq. (2). The second bound of δ assumes $m \geq 1$.

We now analyze the events that break correctness.

The collision event. Let F be a KV-SCF holding the key set X . A lookup on k reads $d \cdot b$ fingerprints and so cannot separate k from another key k' that shares its fingerprint and its bucket, and the lookup can return the value bound to k' rather than to k . By Lemma 1, that sharing means $\text{fp}(k) = \text{fp}(k')$ and $g(k) = g(k')$. We call this the collision event,

$$\text{Coll}(k, X) := [\exists k' \in X \setminus \{k\} : \text{fp}(k') = \text{fp}(k) \wedge g(k') = g(k)]. \quad (3)$$

Initialize draws fp and g independently and uniformly, so for any $k' \neq k$

$$\begin{aligned} \Pr[\text{fp}(k') = \text{fp}(k) \wedge g(k') = g(k)] &= \Pr[\text{fp}(k') = \text{fp}(k)] \cdot \Pr[g(k') = g(k)] \\ &= 2^{-f} \cdot \frac{1}{s}. \end{aligned}$$

The filter holds at most $d \cdot s \cdot b$ keys, so $|X| \leq d \cdot s \cdot b$, and a union bound over $k' \in X \setminus \{k\}$ gives

$$\Pr[\text{Coll}(k, X)] \leq |X| \cdot \frac{2^{-f}}{s} \leq d \cdot b \cdot 2^{-f}. \quad (4)$$

The collision event under mutations. A mutating filter can drift from its store. $F.\text{update}$ and $F.\text{delete}$ locate the slot for k by scanning $\text{Candidates}(k)$ for $\text{fp}(k)$. Since the filter stores no keys, that slot may in fact belong to another present key k' that shares both a bucket and a fingerprint with k . On the SCF of Section 4.2 this mismatch is harmless, since any two slots with the same fingerprint are interchangeable there. In the KV-SCF each slot also carries its own *value*, so such an operation rewrites or removes the value bound to another key. A *delete* on k' can clear the slot of k and leave the slot of k' intact, and a later query for k reads the value at that slot long after the store has forgotten k' , a failure the collision event over X alone does not catch. The same happens for *update*. We must bound the probability of this failure.

The same collision event still covers this failure. A *delete* or *update* on k' scans $\text{Candidates}(k')$ and touches only slots carrying $\text{fp}(k')$. By Lemma 1, every such slot belongs to a key that shares both a bucket and a fingerprint with k' , so a *delete* or *update* on k' can affect the slot of k only when k collides with k' in the sense of Eq. (3).

A lookup on k therefore returns the wrong value in one of two cases. In the first, a key k' currently in the store collides with k , the case already covered by Eq. (3) over X . In the second, a key k' once inserted but since targeted by a

`delete` or `update` collides with k , and that mutation landed on the slot of k . Both cases are instances of the collision event, and its index set widens from X to the history $\mathcal{H}$ of keys the store has ever held. Every key that ever owns a slot lies in $\mathcal{H}$, since only `insert` writes a fingerprint, and so does every key a mutation targets, by well-formedness.

If $\text{Coll}(k, \mathcal{H})$ does not occur, no other key in $\mathcal{H}$ shares both a bucket and a fingerprint with k , so by Lemma 1 every slot in $\text{Candidates}(k)$ carrying $\text{fp}(k)$ belongs to k itself. A lookup on k is therefore unambiguous. When k is currently in the store, exactly one such slot exists and holds the value bound to k . When k is not, no such slot exists.

Proof (of Lemma 2). Fix the queried key $k \in \mathcal{K}$. Let E be the event that, for some of the d index PIR reads, `Recover` returns a value different from the current bucket at the queried index (Definition 1), and let $A := \text{Coll}(k, \mathcal{H})$ be the collision event of Eq. (3) over the key history. A union bound over the d reads gives $\Pr[E] \leq d \cdot \delta_{\text{idx}}$.

Present key. By the isolation argument above, a lookup on k returns the wrong value only when E or A occurs, so a union bound gives

$$\delta \leq \Pr[E \cup A] \leq \Pr[E] + \Pr[A] \leq d \cdot \delta_{\text{idx}} + |\mathcal{H}| \cdot \frac{2^{-f}}{s},$$

using Eq. (4) at $X = \mathcal{H}$ for $\Pr[A]$. With $|\mathcal{H}| \leq m + \tau_{\text{ins}}$ and $m \leq d \cdot b \cdot s$, this rewrites as the second bound of the lemma.

Absent key. We count E as a failure whatever the read returns. If E does not occur, a lookup on k returns a non- $\perp$ value only when a slot of $\text{Candidates}(k)$ carries $\text{fp}(k)$. Such slots belong to keys aligned with k (Lemma 1) and number exactly the aligned keys the store holds, so that event is $\text{Coll}(k, \text{DB}^{(\tau)})$. A union bound with Eq. (4) at $X = \text{DB}^{(\tau)}$ gives

$$\xi \leq \Pr[E] + \Pr[\text{Coll}(k, \text{DB}^{(\tau)})] \leq d \cdot \delta_{\text{idx}} + d \cdot b \cdot 2^{-f}. \square$$

Lemma 3 (Security reduction). *Built on a $(T, \varepsilon_{\text{idx}})$ -secure UIPIR, the construction of Figure 5 is (T', ε) -secure with $\varepsilon \leq d \cdot \varepsilon_{\text{idx}}$ and $T' \approx T$.*

Proof (sketch). We give the outline here, which Appendix A.1 turns into a full proof by building the hybrid over all challenges at once and spelling out what the reduction simulates and what it forwards to its oracles. The candidate map $k \mapsto \text{Candidates}(k)$ is public and deterministic, so the only secret in $\text{Query}(k_b)$ is the randomness of its d UIPIR queries on public target records. For $0 \leq i \leq d$ let the hybrid H_i answer every challenge with segments $j < i$ encoding the candidate of k_1 and segments $j \geq i$ that of k_0 , so that H_0 and H_d are the two worlds of $\text{QIND}_{\text{IPIR}}$. Adjacent hybrids differ in one segment, and a distinguisher for them is a UIPIR adversary for that segment, which the reduction plants as its challenge instance while it simulates the other $d-1$ segments from the public candidate map and forwards that segment's share of every mutation batch to its update oracle. The multi-challenge oracle of Figure 3 folds all challenges into one UIPIR game per segment, so the telescoping sum costs $d \cdot \varepsilon_{\text{idx}}$ and no factor in their number, at $T' \approx T$. $\square$

Lemma 4 (Incrementality reduction). *Built on an updatable UIPIR, the construction of Figure 5 is incremental.*

Proof (sketch). Everything rests on one fact about the KV-SCF, that a single mutation disturbs a constant number of buckets. We follow that fact through to Definition 6 here, and Appendix A.2 does the same with the communication clause separated from the computation clause and the constants tracked through both. An **update** or **delete** rewrites the one bucket whose slot matches $\text{fp}(k)$ and a successful **insert** at most $\text{MaxKicks} + 1$ buckets along its eviction walk (Figure 4), so every operation touches $O(1)$ buckets, a constant independent of m , and a batch of τ operations costs $\text{DBMutation } O(\tau)$ filter work and yields transcripts $(S_j)_{j \in [d]}$ of $O(\tau)$ cells. **HintUpdate** hands each S_j to the **HintUpdate** of the UIPIR, whose cost is proportional to $|S_j|$ and asymptotically below a segment rebuild by Definition 3, so a mutation becomes a sparse, position-predictable batch of index PIR updates whose cost the index PIR sets, which is Definition 6. $\square$

Together, Lemmas 2–4 establish Theorem 1.

6 The IKPIR instantiation: RisePIR

This section instantiates the construction of Figure 5 and obtains RisePIR. Section 6.1 equips FrodoPIR and SimplePIR [14,22] with a mutation phase, making each an updatable index PIR. Section 6.2 drops them in, yielding RisePIR-F and RisePIR-S. Section 6.3 fixes their parameters for the evaluation.

6.1 Instantiating UIPIR from FrodoPIR and SimplePIR

Figure 6 gives FrodoPIR and SimplePIR, the two index PIR schemes we build on. Both take a raw database $\text{DB} \in \mathbb{Z}_p^{N \times w}$ of N records, each a row of w cells over $\mathbb{Z}_p$, so the record space is $\mathcal{V} = \mathbb{Z}_p^w$, and both answer a query from the matrix $\text{D} \in \mathbb{Z}_p^{\rho \times \omega}$ with $\rho \cdot \omega = N \cdot w$ into which **Setup** reshapes DB . The two differ only in that reshape. FrodoPIR keeps the database as is, $\text{D} = \text{DB}$ at $(\rho, \omega) = (N, w)$, so a record is a row. SimplePIR packs several records into one row to make D near-square, $\rho \approx \omega$.

FrodoPIR and SimplePIR

Parameters: LWE dimension n , modulus q , plaintext modulus $p < q$, scaling $\Delta = \lfloor q/p \rfloor$, secret distribution χ_s , and error distribution χ_e . The raw database is a matrix $\text{DB} \in \mathbb{Z}_p^{N \times w}$ of N records, each a row of w cells in $\mathbb{Z}_p$.

Offline phase.

Setup(DB) $\rightarrow$ (D , hint):

- Reshape DB into the database $\text{D} \in \mathbb{Z}_p^{\rho \times \omega}$ with $\rho \cdot \omega = N \cdot w$, the identity $\text{D} = \text{DB}$ for FrodoPIR and near-square for SimplePIR.
- Sample a seed $\beta \xleftarrow{\$} \{0, 1\}^\lambda$ and expand $\mathbf{A} \in \mathbb{Z}_q^{n \times \rho}$ deterministically from it.

```

– Compute the hint  $\mathbf{H} := \mathbf{A} \cdot \mathbf{D} \in \mathbb{Z}_q^{n \times \omega}$ 
– Output the server processed database  $\mathbf{D}$  and the client hint  $\text{hint} := (\mathbf{H}, \beta)$ .
// Client when receive hint, they derive  $\mathbf{A}$  from the seed  $\beta$ 
Online phase.
Query( $i$ )  $\rightarrow$  (qu, st): of  $\mathbf{D}$ :
– Sample  $\mathbf{s} \xleftarrow{\$} \chi_s^n$ ,  $\mathbf{e} \xleftarrow{\$} \chi_e^\rho$ , and set  $\mathbf{b} := \mathbf{s} \cdot \mathbf{A} + \mathbf{e} + \Delta \cdot \mathbf{u}_i \in \mathbb{Z}_q^\rho$ , where  $\mathbf{u}_i$  is
  the  $i$ -th standard basis vector of  $\mathbb{Z}_q^\rho$ .
– Output qu :=  $\mathbf{b}$  and st :=  $\mathbf{s}$ .
Answer(qu,  $\mathbf{D}$ )  $\rightarrow$  ans:
– Output ans :=  $\mathbf{a} = \mathbf{b} \cdot \mathbf{D} \in \mathbb{Z}_q^\omega$ .
Recover(hint, st, ans)  $\rightarrow v$ :
– Output the row  $v := \lfloor \mathbf{a} - \mathbf{s} \cdot \mathbf{H} \rfloor_p \in \mathbb{Z}_p^\omega$ , rounding each coordinate to the nearest
  multiple of  $\Delta$  and dividing by  $\Delta$ .

```

Fig. 6. The FrodoPIR and SimplePIR scheme

Entry-level hint patching. The hint of Figure 6 is linear in the database. Setup forms it as the product $\mathbf{H} = \mathbf{A} \cdot \mathbf{D}$ of the public matrix $\mathbf{A}$, expanded from the seed β , with the database $\mathbf{D}$, and iSimplePIR [25] exploits that linearity to patch $\mathbf{H}$ in place, at a cost proportional to the cells of $\mathbf{D}$ that changed rather than to $\mathbf{D}$ itself. We have

$$\mathbf{H} = \mathbf{A} \cdot \mathbf{D} \in \mathbb{Z}_q^{n \times \omega}, \quad \text{that is,} \quad \mathbf{H}[k'][c] = \sum_{r \in [\rho]} \mathbf{A}[k'][r] \cdot \mathbf{D}[r][c],$$

so a cell (r, c) of $\mathbf{D}$ touches only column c of $\mathbf{H}$, entering all n of its entries through the one term $\mathbf{A}[k'][r] \cdot \mathbf{D}[r][c]$, and adding γ to that cell moves that column by

$$\mathbf{H}[k'][c] \leftarrow \mathbf{H}[k'][c] + \mathbf{A}[k'][r] \cdot \gamma \quad \text{for all } k' \in [n]. \quad (5)$$

With Eq. (5), for each cell of $\mathbf{D}$ that changes by γ , we need only n additions and n multiplications in $\mathbb{Z}_q$ to update the hint $\mathbf{H}$.

The mutation phase. Figure 7 builds DBMutation and HintUpdate for FrodoPIR and SimplePIR on entry-level hint patching. DBMutation applies each update (i, v') to the w cells that the reshape assigns record i and logs every rewritten cell as a triple (r, c, γ) , the transcript S , which HintUpdate replays into $\mathbf{H}$ by Eq. (5).

Mutation phase

```

DBMutation( $\mathbf{D}, (\text{op}_1, \dots, \text{op}_\tau)$ )  $\rightarrow (\mathbf{D}', S)$ , each  $\text{op}_t = (i, v') \in \mathcal{M}_{\text{idx}}$  with  $v' \in \mathbb{Z}_p^w$ :
– Start from  $S := \emptyset$ . For each  $(i, v')$  in order, overwrite the  $w$  cells of  $\mathbf{D}$  that the
  reshape assigns record  $i$  with  $v'$ , and for every cell  $(r, c)$  whose value changed
  append  $(r, c, \gamma)$  to  $S$ , with  $\gamma$  the signed difference between its new and old value.

```

- Output the rewritten database D' and the transcript S .
- HintUpdate**(hint, S) $\rightarrow$ hint', with hint = $(\mathbf{H}, \beta)$:
- For each $(r, c, \gamma) \in S$ in order, read column r of $\mathbf{A}$, cached or re-expanded from β , and set $\mathbf{H}[k'][c] \leftarrow \mathbf{H}[k'][c] + \mathbf{A}[k'][r] \cdot \gamma$ for all $k' \in [n]$, as in Eq. (5).
 - Output hint' := $(\mathbf{H}, \beta)$.

Fig. 7. The mutation phase for UIPIR built on FrodoPIR and SimplePIR

The cost of the two algorithms follows from Eq. (5). An **update** rewrites the w cells of one record, so a batch of τ operations costs **DBMutation** at most τw cell writes and yields a transcript of $|S| \leq \tau w$ triples, the only data a client downloads in the mutation phase. **HintUpdate** pays n additions and n multiplications in $\mathbb{Z}_q$ per triple, so its total cost is at most $n\tau w$ of each. Both costs grow with τ and not with N .

Lemma 5 (FrodoPIR and SimplePIR as UIPIR). *Equipped with the mutation phase of Figure 7, FrodoPIR and SimplePIR of Figure 6 are correct, secure, and updatable UIPIR schemes, each with the correctness error and the security of the scheme it extends, hence secure under decisional LWE [33].*

Proof. It is easy to see that the D' output by **DBMutation** is exactly the processed database **Setup** would produce from the raw database with the batch already applied, since the reshape is fixed and an **update** only overwrites records. Likewise, by Eq. (5) the patched hint equals $\mathbf{A} \cdot D'$, the hint **Setup** would compute on that database from the same seed β . The mutation phase thus moves the scheme from one valid state to another, and every query afterwards is a plain FrodoPIR or SimplePIR query on D' , with the same correctness error. Privacy is also untouched, since a query depends only on the fixed $\mathbf{A}$ and fresh randomness, never on the database or the hint. Finally, the mutation phase costs at most τw writes, $|S| \leq \tau w$ triples, and $n\tau w$ operations of each kind, all proportional to τ and far below the nNw operations and $n\omega$ elements of a rebuild, which is updatability. $\square$

6.2 RisePIR

Instantiating the generic construction of Figure 5 with the two UIPIR schemes of Lemma 5 gives our IKPIR scheme, which we name RisePIR. It comes in two variants, RisePIR-F over FrodoPIR and RisePIR-S over SimplePIR, and by Theorem 1 both are correct, secure, and incremental. Table 1 gives the cost of each variant in the **offline**, **online**, and **mutation** phases.

The $\Theta(n\tau w)$ that Table 1 charges the mutation phase hides a small constant. By Lemma 5, patching one rewritten slot of w cells costs at most $n\omega$ $\mathbb{Z}_q$ operations, and the KV-SCF rewrites one slot per **update** or **delete** and at most $\text{MaxKicks} + 1$ slots per **insert**, along its eviction walk (Section 5.1). A batch of τ mutations therefore costs at most $(\text{MaxKicks} + 1)n\tau w$ operations, and exactly $n\tau w$ when it holds no **insert**, against the nNw that ChalametPIR and KPIR^{index} pay to rebuild the hint on any change.

6.3 Parameter selection

We fix parameters against two criteria, one for security and one for correctness, and we state each criterion before the numbers that meet it.

For security we target NIST Category 1, the level at which an attack costs at least the 2^{143} classical gates of an AES-128 key search, and the level the reference lattice KEMs Kyber-512 and FrodoKEM-640 adopt [6,30]. That is the right anchor because the security of RisePIR reduces entirely to an LWE instance, so the question we ask is the one a lattice KEM asks. We add a stricter floor of our own, at least 128 bits under the core-SVP model of ADPS16 [4], which charges a single sieving call, ignores everything else, and so understates the true cost by design. An instance that clears 128 bits there clears at least as much under any sharper accounting. Each LWE dimension is then pinned to that floor rather than chosen freely, so it follows from the criterion rather than from convention.

For correctness we target $\delta, \xi \leq 2^{-40}$ on both sides of Definition 4. This number is a convention in lattice PIR rather than a derived requirement. **Spiral** fixes a correctness error of 2^{-40} in its parameter search [28] and **SimplePIR** fixes the same value when it instantiates its scheme [22], both in a parameter-selection step and neither by derivation, and much of the work since has followed them. It is not even universal in this line, since **FrodoPIR** states its correctness only qualitatively and fixes no failure probability at all [14]. What justifies keeping it is the kind of event it bounds. A correctness failure is an honest-execution event and not a security event. The bound holds against a worst-case database, so the server’s control of the data is already charged for, and no known attack turns a failure into a break. That is why the target is set far looser than the 2^{-128} a security parameter would ask for. Nor is it forced by cost at our shapes: tightening it to 2^{-128} would leave **RisePIR-F**’s plaintext modulus unchanged at every configuration we report and cost **RisePIR-S** one bit of modulus at three of the six. We therefore adopt 2^{-40} as the shared convention it is, and we recommend that later work name it that way rather than present it as derived.

The LWE instance. Both variants use the modulus $q = 2^{32}$. **RisePIR-F** keeps the secret and error distributions of **FrodoPIR**, which are uniform over $\{-1, 0, 1\}$, the ternary LWE of its Assumption 1 [14]. **RisePIR-S** keeps the secret and error distributions of **SimplePIR**, a uniform secret over $\mathbb{Z}_q$ and a discrete-Gaussian error of width $\sigma = 6.4$ [22]. We pin each dimension n to the 128-bit classical core-SVP floor on the lattice estimator [2], giving $n = 1566$ for **RisePIR-F** and $n = 1275$ for **RisePIR-S**. Both settings also meet NIST Category 1.

In this parameter selection, we do not consider the factor d lost in security from Lemma 3. That factor comes from the reduction, not from an attack. The lemma bounds the advantage against **RisePIR** by d times the advantage against its index PIR, which costs two bits at $d = 4$, but no known attack on the LWE instance becomes d times cheaper because the instance sits inside **RisePIR**. **FrodoPIR** and **SimplePIR** follow the same convention [14,22].

The fingerprint width and the plaintext modulus. By Lemma 2, the target $\delta, \xi \leq 2^{-40}$ holds once $d \cdot \delta_{\text{idx}} \leq 2^{-41}$ and $(1 + \tau_{\text{ins}}/m) db 2^{-f} \leq 2^{-41}$, the second written in the generic form that bounds both sides. We choose $f = 64$, which puts the filter term near 2^{-61} at every configuration we run and leaves the whole budget to the index PIR. For the plaintext modulus p , the index half of the budget becomes a per-cell budget $\delta_{\text{cell}} \leq 2^{-41}/(d \cdot w)$ over the w cells one read decodes, and we take the largest p meeting it. SimplePIR states its correctness at a chosen failure probability, so we retarget that bound at δ_{cell} , while FrodoPIR states its own only qualitatively, so we replace it with an explicit tail bound on the same quantity [14,22]. This gives $p = 2^9$ at every configuration we report, except RisePIR-S (2, 4) at $\ell = 1024$ B, where $p = 2^8$.

Appendix B carries out both selections in full, with the estimator output behind the two LWE instances and the explicit tail bounds behind the plaintext modulus.

7 Evaluation

7.1 Asymptotic Evaluation

Table 1 sets RisePIR against ChalametPIR and KPIR^{index}, the two state-of-the-art static keyword PIR, in all three phases, over relaxed databases of N slots each w wide. All four schemes reduce to FrodoPIR or SimplePIR over the same Nw cells, so their computation in the offline and online phases agrees. They differ only in the layout of that reduction. ChalametPIR reduces to one FrodoPIR and KPIR^{index} to one SimplePIR over all N slots, whereas RisePIR-F and RisePIR-S reduce to d instances of FrodoPIR or SimplePIR, one per segment, over N/d slots each. This is what sets the hint, query, and response sizes apart in Table 1. Concretely, RisePIR-F carries a hint and a response db times larger than ChalametPIR's, since a query retrieves a bucket of b slots from each of the d segments, but a query b times smaller, one element per bucket rather than per slot. RisePIR-S pays a factor $\sqrt{d}$ over KPIR^{index} on all three, the price of segmentation on a square layout.

In return, RisePIR-F and RisePIR-S handle key-value store mutations far better than ChalametPIR and KPIR^{index}. Any change, however small, makes ChalametPIR and KPIR^{index} re-run the offline phase and send the new hint to every client, nNw operations of computation and a download of nw elements for ChalametPIR and $n\sqrt{Nw}$ for KPIR^{index}. RisePIR instead spends $\Theta(n\tau w)$ operations and ships a delta of $\Theta(\tau w)$ cells, with a constant of at most $\text{MaxKicks} + 1$ (Section 6.2). That bound is a worst case. An **update** or **delete** rewrites exactly one slot, and a successful **insert** into a KV-SCF that is not close to full rewrites only a few, so the constant is small in practice, which is the efficiency the next section demonstrates.

7.2 SCF implementation and evaluation

Implementation. We implement the SCF in Rust, in the same open-source repository as RisePIR¹ beside a standard d -ary cuckoo filter [16,17,37] that serves as the reference. The numbers in Table 2 are measured on an AWS `r7a.xlarge` instance with a 64-bit fingerprint and a kick budget of $MaxKicks = 2500$. Each load factor is the mean of 20 fills of an empty filter until its first `insert` fails. Each throughput is the mean of 10 timed trials after 3 untimed warm-up trials, where `insert` fills the filter to capacity, `lookup` queries it at capacity with a 50% hit rate, and `delete` empties it again. The standard deviation over trials stays below 0.06 percentage points on load factor and below 8% of the mean on throughput, most cells far tighter.

The choice of d and b . Two criteria pick the configurations of Table 2. First, we want the filter to run at a fill of 90 to 95%, so that its storage is well used. A filter close to its maximum load factor, however, makes each `insert` walk a long eviction chain, and the mutation phase of RisePIR pays for every slot on that chain. So we ask for a maximum load factor of 97% or more. Walzer’s tabulation [35] gives that maximum for each (d, b) , which rules out $(2, 1)$ at 50%, $(2, 2)$ at 89.7%, $(3, 1)$ at 91.8%, and $(2, 3)$ at 95.9%, and admits $(2, 4)$, $(3, 2)$, $(3, 3)$, $(4, 1)$, and $(4, 2)$, from 97.7% to 99.8%. Second, communication of RisePIR-F and RisePIR-S grows with d and b . By Table 1, RisePIR-S pays a factor $\sqrt{d}$ over KPIR^{index} and RisePIR-F a factor db over ChalametPIR on hint and response, so we stop at $d = 4$ and at the smallest b that clears the first criterion, plus one more slot at $d = 3$ and $d = 4$, where it lifts the maximum past 99.7% at a modest extra response.

Results. Table 2 compares the SCF with a standard cuckoo filter of the same (d, b) on two things, the maximum load factor and the throughput of the three operations, in tables of 1.0 to 1.6 million buckets. On load factor one would expect the SCF to lose. Confining the j -th candidate of every key to segment j takes placement freedom away from cuckoo hashing, so the segmented table should fill less before an `insert` fails. The measurement says otherwise. The SCF reaches a slightly higher load factor than the standard filter in all five configurations, both stay within 0.7 percentage points of the theoretical maximum (the Theory column, Walzer’s tabulation [35] of prior work [19,15]), and the SCF peaks at 99.60% at $(4, 2)$. Segmentation costs nothing in storage.

On throughput the outcome depends on the arity. A standard d -ary filter computes a key’s candidates by digit-wise XOR in base d , so its bucket count must be a power of d , and at $d = 3$ that means base-3 arithmetic through a lookup table, which is slow. The SCF needs only each segment to be a power of two, $n_b = d \cdot 2^z$, so a candidate is one binary XOR whatever d is. At arity 3 the SCF is therefore 1.3 to 1.8 $\times$ faster on `insert`, `lookup`, and `delete`. At arities 2

¹ <https://github.com/orochi-network/IKPIR>

Table 2. Maximum load factor and operation throughput (million ops/s) of the SCF and a standard cuckoo filter across the five configurations RisePIR uses; the better filter on each metric is in **bold**.

Arity	b	Scheme	n_b	Load factor (%)		Throughput (Mops)		
				Achieved	Theory [35]	Insert	Lookup	Delete
2	4	Segmented	1 048 576	97.40	98.04	4.02	15.18	13.85
		Standard	1 048 576	97.34		3.79	15.15	13.90
3	2	Segmented	1 572 864	98.33	98.82	4.48	20.85	17.40
		Standard	1 594 323	98.26		3.01	11.36	10.80
	3	Segmented	1 572 864	99.42	99.73	3.55	11.80	10.76
		Standard	1 594 323	99.39		2.65	7.32	7.70
4	1	Segmented	1 048 576	97.13	97.68	5.92	42.47	28.58
		Standard	1 048 576	97.09		6.48	41.39	28.14
	2	Segmented	1 048 576	99.60	99.82	6.16	23.36	18.76
		Standard	1 048 576	99.56		6.42	21.90	18.76

and 4 the standard table is a power of two as well, both filters use binary XOR, and they run within about 10% of each other, neither consistently ahead.

7.3 Keyword PIR implementation and evaluation

Implementation. We implement RisePIR-F and RisePIR-S in Rust² and compare RisePIR-F against ChalametPIR and RisePIR-S against KPIR^{index} under the same setting. ChalametPIR runs from the authors’ code [11]. KPIR^{index} we run from our own Rust reimplementation,³ since the authors’ reference implementation in mpc4j [24] is written in Java and would not compare fairly with the two Rust implementations. All four schemes run single-threaded without SIMD. RisePIR uses the parameters present in Section 6.3, and ChalametPIR and KPIR^{index} inherit those same parameters, ChalametPIR matching RisePIR-F on FrodoPIR and KPIR^{index} matching RisePIR-S on SimplePIR.

Every number in Tables 3 and 4 is measured on AWS `r7a.xlarge` instances. The online phase is benchmarked on a key-value store of $m = 10^6$ pairs at value sizes $\ell = 256$ B and 1024 B, with RisePIR in the three configurations (2, 4), (4, 1), and (4, 2). At arity 3 the KV-SCF table has $3 \cdot 2^z$ slots and cannot hold 10^6 keys at a comparable fill, so we defer those configurations to the full version. The mutation phase uses the same three configurations with a KV-SCF table of $N = 2^{20}$ slots. For each operation type we fill the KV-SCF to 90%, apply a batch of $\tau = 10485$ mutations, and time HintUpdate over the whole batch. The rebuild column times a full Setup on the same store.

² <https://github.com/orochi-network/IKPIR>

³ <https://github.com/bao-ninh-orochi/KPIR-index>

Table 3. Per-query online cost at $m = 10^6$ keys: query size (Qry.), response size (Rsp.) and server answer latency (Ans.); and database expansion rate (Exp.) per value size ℓ . (d, b) is the KV-SCF arity and bucket size.

Scheme	$\ell = 256$ B				$\ell = 1024$ B			
	Qry. (kB)	Rsp. (kB)	Ans. (ms)	Exp. (N/m)	Qry. (kB)	Rsp. (kB)	Ans. (ms)	Exp. (N/m)
ChalamePIR	4 522	0.92	49.2	1.130	4 522	3.65	197	1.130
RisePIR-F (2, 4)	1 049	7.53	36.5	1.049	1 049	29.4	207	1.049
RisePIR-F (4, 1)	4 194	3.77	41.6	1.049	4 194	14.7	145	1.049
RisePIR-F (4, 2)	2 097	7.53	37.4	1.049	2 097	29.4	147	1.049
KPIR ^{index}	61.3	72.4	45.8	1.180	128.5	177.5	225	1.381
RisePIR-S (2, 4)	87.4	90.3	45.3	1.049	174.8	198.2	187	1.049
RisePIR-S (4, 1)	127.1	124.1	43.6	1.049	246.7	249.7	166	1.049
RisePIR-S (4, 2)	123.4	127.9	43.1	1.049	262.2	235.0	166	1.049

Online phase. Every answer in Table 3 takes under a second, and RisePIR answers ahead of its own baseline in every configuration but one. At $\ell = 256$ B, RisePIR-F answers in 36.5 to 41.6 ms against ChalamePIR’s 49.2 ms, and RisePIR-S in 43.1 to 45.3 ms against KPIR^{index}’s 45.8 ms. At $\ell = 1024$ B the two shapes at $d = 4$ answer about a quarter faster than their baselines, while RisePIR-F (2, 4) lands about 5% behind ChalamePIR. These answer times do not follow Table 1 as closely as the communication does, and the reason is almost entirely implementation.

The communication cost follows Table 1. RisePIR-F uploads a query b times smaller than ChalamePIR’s and downloads a response about db times larger, and RisePIR-S moves about $\sqrt{d}$ times what KPIR^{index} moves in both directions. The small deviations from these factors come from the database sizes. Table 1 compares the schemes over the same N slots, whereas here each scheme expands the same 10^6 keys into its own N , again the expansion column of Table 3.

Mutation phase. RisePIR patches its hint in place in 0.06–4.7 ms per mutation, whereas a static scheme rebuilds it in 55.9–686.0 s, five to six orders of magnitude apart. A mutation costs a database write by DBMutation and a hint patch by HintUpdate, and a static scheme pays the same write before it rebuilds from scratch, so Table 4 sets one local patch against one full recomputation. In the time of one rebuild at $\ell = 256$ B, 55.9–65.3 s on FrodoPIR and 127.9–155.7 s on SimplePIR, RisePIR-F absorbs $1.1\text{--}2.8 \times 10^5$ insert or $0.6\text{--}1.1 \times 10^6$ update operations, and RisePIR-S $0.7\text{--}2.1 \times 10^5$ and $4.6\text{--}6.3 \times 10^5$. At $\ell = 1024$ B the hint row is four times wider and both sides grow with it, the patch by $2.4\text{--}7\times$ and the rebuild by $3.5\text{--}5.4\times$, and only the rebuild grows with N . Every throughput cell repeats within 8% across runs except RisePIR-F (2, 4) and (4, 2) at $\ell = 1024$ B, the two largest hints we patch, whose runs split into clusters 11–24% apart.

Table 4. Mutation throughput and rebuild cost of RisePIR at $N = 2^{20}$ slots ($m \approx 0.94 \times 10^6$ pairs), fill 0.90. Ins., Upd., Del.: client **HintUpdate** ops/s per operation type. Rbld.: the end-to-end **Setup** a static scheme pays on every mutation, in seconds; **RisePIR** patches in place instead. The **RisePIR-S** (4, 1) rebuild at $\ell = 256$ B is the mean of five runs split into modes 115–118 and 144–146 s.

Scheme	$\ell = 256$ B				$\ell = 1024$ B			
	Ins. (ops/s)	Upd. (ops/s)	Del. (ops/s)	Rbld. (s)	Ins. (ops/s)	Upd. (ops/s)	Del. (ops/s)	Rbld. (s)
RisePIR-F (2, 4)	1 958	10 990	10 565	55.9	317	1 693	1 764	219.9
RisePIR-F (4, 1)	2 488	16 912	16 357	65.3	694	4 458	4 520	231.1
RisePIR-F (4, 2)	4 521	11 653	11 326	61.6	629	1 807	1 842	214.8
RisePIR-S (2, 4)	691	4 043	3 696	155.4	245	1 299	1 236	541.8
RisePIR-S (4, 1)	525	3 573	3 450	127.9	211	1 362	1 377	686.0
RisePIR-S (4, 2)	1 356	3 475	3 521	155.7	566	1 416	1 428	591.2

Two effects set the cells of Table 4 apart. The eviction chain of the KV-SCF separates **insert** from **update** and **delete**, and the memory hierarchy separates the shapes within a column. An **insert** rewrites every slot on its eviction chain, and each rewritten slot costs one patch, so the ratio of **update** to **insert** throughput is the chain length plus one. It is 2.5–2.9 at (4, 2) and 5.3–6.8 at (2, 4) and (4, 1), for both variants and both value sizes, the constant of Section 7.1 bounded by $MaxKicks + 1$ and small in practice. Table 2 explains the split. (4, 2) saturates at 99.60%, so at fill 0.90 an **insert** finds a free slot after 1–2 evictions on average, whereas (2, 4) and (4, 1) saturate near 97%, so the same fill is over 92% of their capacity and an **insert** evicts 4–6 slots.

The memory hierarchy sets the shapes apart within a column. An **update** or **delete** does the same arithmetic in every shape, so its speed is set by whether the hint fits in cache. At $\ell = 256$ B RisePIR-F holds 5.6 MiB of hint at (4, 1) and 11.2 MiB at (2, 4) and (4, 2), four times as much at $\ell = 1024$ B, on a 16 MiB share of last-level cache. So (4, 1) patches mostly in cache and runs 1.4 – $2.6\times$ faster per slot than the other two, which spill to DRAM. RisePIR-S holds 110–304 MiB of hint in every shape and patches from DRAM throughout, so its columns are flat across shapes and 1.3 – $4.7\times$ behind RisePIR-F on no more arithmetic per slot. The gap is smallest, $1.3\times$, at (2, 4) and (4, 2) for $\ell = 1024$ B, exactly where RisePIR-F spills to DRAM as well.

8 Application: Private `eth_getBalance` at Ethereum scale

A standard `eth_getBalance` query hands the RPC provider the address it asks about, and the provider can log it together with the requester’s IP address and query pattern. Over time this metadata links a real-world identity to an Ethereum address, and a set of addresses queried together lays out a portfolio and its trading strategy. The chain state is public, so only the query needs hiding.

The lookup is keyed by an address in a store of balances that changes with every 12 s block, so a static keyword PIR would rebuild its hint and re-ship it to every client every 12 s. This is the setting incremental keyword PIR is built for.

We deploy RisePIR-S over Ethereum mainnet, serving the 2.05×10^8 accounts with a nonzero balance. Each slot holds the fingerprint of the account’s address together with a 22-byte record: an 18-byte value carrying the 96-bit balance, plus a key tag and a checksum that catch false matches. The KV-SCF runs at $(2, 4)$ with $N = 2^{28}$ slots, which the 2.05×10^8 keys fill to 76%, and the store spans 23.6 GB. RisePIR-S keeps the LWE parameters of Section 6.3, with $p = 2^8$ and $f = 64$ for correctness.

The service is public at <https://risepir.org/>. The client runs in the browser, compiled to WebAssembly, so the queried address never leaves the user’s machine. The server is a GCP `c3d-highmem-16`, an 8-core AMD EPYC 9B14 (Zen 4) with 125 GiB of memory. It runs the optimized build of our implementation on all 16 hardware threads, rather than the single-threaded build of Section 7.3.

A new client downloads the 554 MB hint once. A private `eth_getBalance` query then completes in 0.69 s end to end, with a 435 kB query and a 435 kB response, measured from an AWS `r7a.xlarge` client. Each block changes 433 accounts on average, which the server absorbs in 5.6 ms and publishes as a 10.8 kB transcript, against the 29.2 s a full hint rebuild takes on the same 16 threads, 5 200× cheaper and a 0.05% duty cycle of the 12 s block.

9 Discussion and Conclusion

Choosing between RisePIR-F and RisePIR-S. The generic construction leaves the index PIR to the deployment, and our two variants pay the cost of a large store on opposite sides of the link. Table 1 sets the difference out in one place, the growth of each cost in the number of slots N . RisePIR-F carries a hint of $ndbw$ elements and a response of dbw , both fixed by the filter shape and the value width, so neither grows with the store, and it pays for that with a query of N/b elements that grows linearly with it. RisePIR-S spreads the cost evenly instead, a hint of $n\sqrt{Nwd}$ against a query and a response of $\sqrt{Nwd}$, so its query stays small at any scale while its hint grows as $\sqrt{N}$, the one download a client must take before it can query at all.

The scarcest resource picks the variant, and at scale the store decides which resource that is. At the 10^6 keys of Section 7.3, RisePIR-F uploads 1.0–4.2 MB per query and downloads 3.8–29.4 kB, whereas RisePIR-S moves 87–262 kB up and 90–250 kB down (Table 3). RisePIR-F is the variant for a scarce downlink and RisePIR-S for the case where both directions are scarce. The hint parts them furthest. RisePIR-F holds 5.6–11.2 MiB of it at $\ell = 256$ B and 22.4–44.8 MiB at $\ell = 1024$ B, RisePIR-S 110–304 MiB across both. That footprint carries straight into the mutation phase, since only RisePIR-F’s smallest hint, 5.6 MiB at $\ell = 256$ B, still fits the last-level cache, while RisePIR-S patches from DRAM in every

shape at both value sizes. RisePIR-F therefore absorbs an `update` or `delete` 1.3–4.7× faster (Table 4).

Over the $N = 2^{28}$ slots of Section 8, RisePIR-S queries with 435 kB against a one-time 554 MB hint, whereas RisePIR-F at the same $(2, 4)$ would upload 2^{26} elements, 268 MB at $q = 2^{32}$, on every query for a hint that never grows. That is why we deploy RisePIR-S there. ChalametPIR and KPIR^{index} offer no such choice, each tied to one index PIR. RisePIR composes with both through the same interface, and with whatever updatable index PIR later replaces them.

Choosing the filter shape. Inside a variant, the arity d and the bucket size b of the KV-SCF are the remaining levers, and a wider bucket does not move every cost the same way. The query of RisePIR-F shrinks as N/b , so widening the bucket cuts the upload. Its hint and response grow as $d \cdot b \cdot w$, so widening the bucket or raising the arity enlarges both the download and the footprint every client patches. And the maximum load factor rises with b at fixed arity, from 97.13% at $(4, 1)$ to 99.60% at $(4, 2)$ (Table 2), so the same widening also leaves more room before an `insert` fails. On RisePIR-S the first two levers fall away, since query, response, and hint all scale as $\sqrt{Nwd}$ and only the arity enters, so there the filter’s own behavior sets the shape almost by itself.

The three shapes we measure sit at the corners of that space. $(4, 1)$ gives the smallest response, 3.8 kB at $\ell = 256$ B, and the smallest hint, 5.6 MiB, which keeps its `update` and `delete` 1.4–2.6× ahead of the other two shapes on RisePIR-F. It pays with the largest query, 4.2 MB, and the lowest ceiling of the three at 97.13%. $(2, 4)$ takes the opposite corner, the smallest query at 1.0 MB over a hint that spills to DRAM. $(4, 2)$ buys insertion headroom instead, its 99.60% ceiling leaving an `insert` at fill 0.90 only 1–2 evictions to walk against the 4–6 that $(2, 4)$ and $(4, 1)$ walk. It absorbs 4 521 `inserts` per second on RisePIR-F and 1 356 on RisePIR-S at $\ell = 256$ B, while the other two shapes manage 1 958–2 488 and 525–691 (Table 4). A deployment picks the shape from its scarcest resource, $(4, 1)$ when the downlink or the client’s memory is tight, $(2, 4)$ when the uplink dominates, and $(4, 2)$ when `inserts` are frequent enough to be worth the headroom.

Arity 3 fills in between. Its ceilings, 98.33% at $(3, 2)$ and 99.42% at $(3, 3)$, fall between $(4, 1)$ and $(4, 2)$, and the segmented layout runs 1.3–1.8× faster than the standard filter there rather than paying the usual base-3 addressing penalty (Section 7.2). These two shapes are a first-class option and not a fallback. We keep them out of the keyword-level tables only because a $3 \cdot 2^z$ table cannot hold 10^6 keys at a fill comparable to the other shapes.

Update leakage. The data owner performs every mutation and publishes it, so a mutation is a public event that carries no secret, and we state plainly what its transcript shows. A mutation publishes one transcript of triples (r, c, γ) per segment (Figure 7), so an observer reads off which cells of the processed database changed and by how much, and through the fixed reshape which slots of the KV-SCF moved. Since the candidate map is public and deterministic, an observer

that can enumerate the key space then matches a changed bucket to the key that moved it. An **insert** shows more still, since it logs every slot on its eviction chain, so the length of its transcript publishes how long that chain ran and with it how close to full the KV-SCF sits. All of this is a function of public data alone. Nothing a client sends ever enters a transcript, which is why the game of Figure 3 can hand the adversary every transcript in full and still protect the queried key (Lemma 3). This is the leakage model of iSimplePIR [25], which RisePIR inherits together with the entry-level hint patch.

The model does not give deletion in the strong sense. After a **delete**, a query for the removed key returns $\perp$, and a client that joins later learns nothing of it. A client that kept the transcript, however, keeps the value, since the triple recording the cleared slot carries the signed difference in the clear. Deletion is therefore *weak* in the sense standard for preprocessing PIR, removing the key from every future answer without erasing it from the history a client has already downloaded. The limit is the model’s and not our construction’s, since any scheme that publishes its mutations lets a client keep what it has already seen.

Conclusion. RisePIR makes keyword PIR incremental by changing one component, the relaxed database between the keyword layer and the index PIR. The index PIR beneath ChalametPIR and KPIR^{index} can already patch its hint at a cost proportional to the changed entries, but their static relaxed database turns a single mutation into a full rebuild. The d -ary Segmented Cuckoo Filter gives a relaxed database that is dynamic and position-predictable. Its key-value form flattens the key-value store into d index PIR databases, one per segment, so that a keyword query becomes d index PIR queries and a mutation becomes a sparse, position-predictable batch of index PIR updates. Built on it, incremental keyword PIR follows generically from any updatable index PIR, the first such construction to our knowledge, and it needs from that index PIR only an efficient in-place **update**.

RisePIR instantiates the construction over FrodoPIR and SimplePIR under decisional LWE. On a store of about one million keys, it patches its hint in 0.06–4.7 ms per mutation where re-running the preprocessing takes 56–686 s, at an online cost competitive with ChalametPIR and KPIR^{index}. Over 2.05×10^8 Ethereum accounts, RisePIR-S absorbs a block’s changes in 5.6 ms where a rebuild of the same hint takes 29 s, longer than the 12 s block. A scheme that rebuilds on every change cannot keep pace with the chain. Here incrementality is what makes keyword PIR deployable, not only cheaper.

Limitation. A KV-SCF that grows too full, close to its maximum load factor, can cause an **insert** to fail, and the only fix is to rebuild the KV-SCF at a larger capacity and re-run the preprocessing phase. The cost amortizes when we reserve enough capacity for the application’s needs, so **inserts** rarely fail. This is acceptable in practice, though not a clean solution in theory. The limitation comes from using only the *updatability* of the index PIR; removing it requires

another relaxed database and the *incrementality* of the index PIR, a direction we leave to future work.

References

1. Albrecht, M.R., Gheorghiu, V., Postlethwaite, E.W., Schanck, J.M.: Estimating quantum speedups for lattice sieves. In: *Advances in Cryptology – ASIACRYPT 2020*. pp. 583–613. Springer (2020)
2. Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of learning with errors. *J. Math. Cryptol.* **9**(3), 169–203 (2015)
3. Ali, A., Lepoint, T., Patel, S., Raykova, M., Schoppmann, P., Seth, K., Yeo, K.: {Communication–Computation} trade-offs in {PIR}. In: *30th USENIX security symposium (USENIX Security 21)*. pp. 1811–1828 (2021)
4. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key {ExchangeA} new hope. In: *25th USENIX security symposium (USENIX Security 16)*. pp. 327–343 (2016)
5. Arora, S., Ge, R.: New algorithms for learning in presence of errors. In: *Automata, Languages and Programming – ICALP 2011*. pp. 403–415. Springer (2011)
6. Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Kyber: Algorithm specifications and supporting documentation. NIST Post-Quantum Cryptography Standardization, Round 3 submission (2021)
7. Bender, M.A., Farach-Colton, M., Johnson, R., Kuszmaul, B.C., Medjedovic, D., Montes, P., Shetty, P., Spillane, R.P., Zadok, E.: Don’t thrash: How to cache your hash on flash. In: *3rd Workshop on Hot Topics in Storage and File Systems (Hot-Storage 11)* (2011)
8. Bonomi, F., Mitzenmacher, M., Panigrahy, R., Singh, S., Varghese, G.: An improved construction for counting bloom filters. In: *European Symposium on algorithms*. pp. 684–695. Springer (2006)
9. Breslow, A.D., Jayasena, N.S.: Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proceedings of the VLDB Endowment* **11**(9), 1041–1055 (2018)
10. Celi, S., Davidson, A.: Call me by my name: Simple, practical private information retrieval for keyword queries. In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. pp. 4107–4121 (2024)
11. Celi, S., Davidson, A.: ChalametPIR: reference Rust implementation. <https://github.com/claucece/chalamet> (2024)
12. Chor, B., Gilboa, N., Naor, M.: Private information retrieval by keywords. Technion-IIT, Department of Computer Science (1997)
13. Chor, B., Kushilevitz, E., Goldreich, O., Sudan, M.: Private information retrieval. *Journal of the ACM (JACM)* **45**(6), 965–981 (1998)
14. Davidson, A., Pestana, G., Celi, S.: FrodoPIR: Simple, scalable, single-server private information retrieval. *Proceedings on Privacy Enhancing Technologies* (2023)
15. Dietzfelbinger, M., Goerdt, A., Mitzenmacher, M., Montanari, A., Pagh, R., Rink, M.: Tight thresholds for cuckoo hashing via xorsat. In: *International Colloquium on Automata, Languages, and Programming*. pp. 213–225. Springer (2010)
16. Fan, B., Andersen, D.G., Kaminsky, M., Mitzenmacher, M.D.: Cuckoo filter: Practically better than bloom. In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. pp. 75–88 (2014)

17. Fan, B., Andersen, D.G., Kaminsky, M.: Cuckoo filter: reference implementation. <https://github.com/efficient/cuckoofilter> (2014)
18. Ferragina, P., Vinciguerra, G., et al.: The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment* **13**(8), 1162–1175 (2020)
19. Fountoulakis, N., Panagiotou, K.: Sharp load thresholds for cuckoo hashing. *Random Structures & Algorithms* **41**(3), 306–333 (2012)
20. Graf, T.M., Lemire, D.: Binary fuse filters: Fast and smaller than xor filters. *Journal of Experimental Algorithmics (JEA)* **27**(1), 1–15 (2022)
21. Hao, M., Liu, W., Peng, L., Zhang, C., Wu, P., Zhang, L., Li, H., Deng, R.H.: Practical keyword private information retrieval from {Key-to-Index} mappings. In: 34th USENIX Security Symposium (USENIX Security 25). pp. 3397–3416 (2025)
22. Henzinger, A., Hong, M.M., Corrigan-Gibbs, H., Meiklejohn, S., Vaikuntanathan, V.: One server for the price of two: Simple and fast {Single-Server} private information retrieval. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 3889–3905 (2023)
23. Kwon, A.H., Lazar, D., Devadas, S., Ford, B.: Riffle: An efficient communication system with strong anonymity (2015)
24. Liu, W., et al.: mpc4j: a multi-party computation and private information retrieval library. <https://github.com/alibaba-edu/mpc4j> (2025)
25. Lu, P., Xu, G., Li, Z., Wang, M., Cui, H.: Incremental single-server private information retrieval. *Cryptology ePrint Archive*, Paper 2026/030 (2026)
26. Mahdavi, R.A., Kerschbaum, F.: Constant-weight {PIR}: single-round keyword {PIR} via constant-weight equality operators. In: 31st USENIX security symposium (USENIX Security 22). pp. 1723–1740 (2022)
27. MATZOV: Report on the security of LWE: Improved dual lattice attack. Zenodo (2022), <https://doi.org/10.5281/zenodo.6412486>
28. Menon, S.J., Wu, D.J.: Spiral: Fast, high-rate single-server PIR via FHE composition. In: 2022 IEEE Symposium on Security and Privacy (S&P). pp. 930–947. IEEE (2022)
29. Mittal, P., Olumofin, F., Troncoso, C., Borisov, N., Goldberg, I.: {PIR-Tor}: Scalable anonymous communication using private information retrieval. In: 20th USENIX security symposium (USENIX security 11) (2011)
30. Naehrig, M., Alkim, E., Bos, J., Ducas, L., Easterbrook, K., LaMacchia, B., Longa, P., Mironov, I., Nikolaenko, V., Peikert, C., Raghunathan, A., Stebila, D.: FrodoKEM: Learning with errors key encapsulation. algorithm specifications and supporting documentation. NIST Post-Quantum Cryptography Standardization, Round 3 submission (2021)
31. Pandey, P., Bender, M.A., Johnson, R., Patro, R.: A general-purpose counting filter: Making every bit count. In: Proceedings of the 2017 ACM international conference on Management of Data. pp. 775–787 (2017)
32. Patel, S., Seo, J.Y., Yeo, K.: {Dont} be dense: efficient keyword {PIR} for sparse databases. In: 32nd USENIX security symposium (USENIX Security 23). pp. 3853–3870 (2023)
33. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)* **56**(6), 1–40 (2009)
34. Vöcking, B.: How asymmetry helps load balancing. *Journal of the ACM (JACM)* **50**(4), 568–589 (2003)
35. Walzer, S.: Load thresholds for cuckoo hashing with overlapping blocks. *ACM Transactions on Algorithms* **19**(3), 1–22 (2023)

36. Wang, M., Zhou, M.: Vacuum filters: more space-efficient and faster replacement for bloom and cuckoo filters. *Proceedings of the VLDB Endowment* (2019)
37. Xie, Z., Ding, W., Wang, H., Xiao, Y., Liu, Z.: D-ary cuckoo filter: A space efficient data structure for set membership lookup. In: 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS). pp. 190–197. IEEE (2017)

A Proof for Lemma 3 and Lemma 4

Section 5.3 proves the correctness reduction in full and gives the idea behind the other two. This appendix completes them. We restate each lemma, then prove it along the lines the body sketched.

A.1 Proof of the security reduction

Lemma 3 (Security reduction). *Built on a $(T, \varepsilon_{\text{idx}})$ -secure UIPIR, the construction of Figure 5 is (T', ε) -secure with $\varepsilon \leq d \cdot \varepsilon_{\text{idx}}$ and $T' \approx T$.*

Proof (of Lemma 3). Recall what the adversary $\mathcal{A}$ controls in $\text{QIND}_{\text{IKPIR}}^A$ (Figure 3). It picks the initial store, drives the state through any sequence of mutation batches with $\mathcal{O}_{\text{upd}}$, and reads the mutated database, the transcript, and the refreshed hint after each one. It may call $\mathcal{O}_{\text{chal}}$ any number of times, all under the single secret bit b , and it may interleave the two oracles as it likes. The state (D, hint) therefore keeps moving, and the argument below never freezes it: the reduction forwards every mutation batch to its own update oracle, so its simulation follows whatever interleaving $\mathcal{A}$ chooses. Write Q for the number of challenges it issues, and let the ℓ -th name the keys $k_0^{(\ell)}, k_1^{(\ell)}$ with candidate positions $(r_j^{(\ell)})_{j \in [d]}$ and $(r'_j{}^{(\ell)})_{j \in [d]}$.

What a challenge query carries. $\text{Query}(k)$ computes the candidate positions $\text{Candidates}(k)$ and runs one UIPIR.Query per segment on them. The candidate map is a deterministic function of the key and the public parameters pp , both of which the adversary holds, so it hides nothing by itself: an adversary that holds k_0 and k_1 computes both candidate tuples and has only to tell which of the two a query encodes. All the secrecy of a challenge query therefore sits in the d index PIR queries, and the keyword layer adds none of its own. This is why the argument moves segment by segment.

A hybrid over the segments. For $0 \leq i \leq d$ let H_i be the game that answers every one of the Q challenges by encoding the candidate of $k_1^{(\ell)}$ in the segments $j < i$ and the candidate of $k_0^{(\ell)}$ in the segments $j \geq i$. Then H_0 answers every challenge exactly as $\text{Query}(k_0^{(\ell)})$ and H_d exactly as $\text{Query}(k_1^{(\ell)})$, the two worlds of $\text{QIND}_{\text{IKPIR}}$ at $b = 0$ and $b = 1$. Two consecutive hybrids H_{i-1} and H_i agree on every segment but one: segment $i - 1$ encodes $r_{i-1}^{(\ell)}$ in the first and $r'_{i-1}{}^{(\ell)}$ in the second, and it does so in all Q challenges at once. Switching a whole segment across the game, rather than one challenge at a time, is what keeps the chain at d steps instead of $d \cdot Q$.

One index PIR reduction per step. Fix $i \in \{1, \dots, d\}$ and let $\mathcal{B}_i$ be the UIPIR adversary that plants segment $i - 1$ as its own challenge instance and simulates the other $d - 1$ segments itself. It can simulate them because it knows everything they encode: $\mathcal{A}$ names both challenge keys and the candidate map is public, so $\mathcal{B}_i$ computes every $r_j^{(\ell)}$ and $r'_j^{(\ell)}$ on its own. Concretely, $\mathcal{B}_i$ answers $\mathcal{A}$ as follows.

- At the start it takes the store DB that $\mathcal{A}$ outputs and runs the filter part of **Setup** on it, building the KV-SCF with coins of its own. It submits segment F_{i-1} as the database of its own UIPIR game and receives $(D_{i-1}, \text{hint}_{i-1})$, runs **UIPIR.Setup** on the other $d - 1$ segments itself, and hands $\mathcal{A}$ the D and hint that Figure 5 assembles from the two.
- On a mutation batch, it applies the batch to the KV-SCF and splits the resulting change set by segment exactly as **DBMutation** does. It forwards the record overwrites of segment $i - 1$ to its own update oracle $\mathcal{O}_{\text{upd}}$ and applies the rest to the segments it holds. It then returns to $\mathcal{A}$ the mutated database, the transcript, and the refreshed hint, taking segment $i - 1$'s share of all three from what its oracle returned.
- On the ℓ -th challenge $(k_0^{(\ell)}, k_1^{(\ell)})$, it calls its own $\mathcal{O}_{\text{chal}}(r_{i-1}^{(\ell)}, r'_{i-1}^{(\ell)})$ once for segment $i - 1$, answers the segments $j < i - 1$ with **UIPIR.Query** $(r_j^{(\ell)})$ and the segments $j > i - 1$ with **UIPIR.Query** $(r_j^{(\ell)})$, and hands back the d queries as one keyword query.
- At the end it outputs whatever $\mathcal{A}$ outputs.

The simulation is exact in both worlds. Under the UIPIR bit 0 the challenge oracle encodes $r_{i-1}^{(\ell)}$ and $\mathcal{A}$ sees H_{i-1} , and under bit 1 it encodes $r'_{i-1}^{(\ell)}$ and $\mathcal{A}$ sees H_i . The UIPIR game grants $\mathcal{B}_i$ the same two oracles the keyword game grants $\mathcal{A}$, so every mutation batch and every one of the Q challenges passes through without the reduction having to guess anything.

Write p_i for the probability that $\mathcal{A}$ outputs 1 in H_i . Since $\mathcal{B}_i$ forwards that output as its own guess, its advantage in the UIPIR game is exactly $\frac{1}{2}|p_i - p_{i-1}|$. It runs inside the time budget of Definition 2, by the accounting below, so that definition bounds its advantage by ε_{idx} and

$$|p_i - p_{i-1}| \leq 2\varepsilon_{\text{idx}} \quad \text{for every } i \in \{1, \dots, d\}. \quad (6)$$

Summing the steps. The keyword advantage carries the same normalization as the index one, $\varepsilon = |\Pr[b' = b] - \frac{1}{2}| = \frac{1}{2}|p_d - p_0|$, so the two halves cancel along the telescoping sum and Eq. (6) gives

$$\varepsilon = \frac{1}{2}|p_d - p_0| \leq \frac{1}{2} \sum_{i=1}^d |p_i - p_{i-1}| \leq d \cdot \varepsilon_{\text{idx}}.$$

The bound charges one index PIR advantage per segment and nothing per challenge, because the Q challenges ride inside each segment's single UIPIR game through the multi-challenge oracle of Figure 3. A factor Q appears only if one further expands ε_{idx} into the single-challenge advantage $\varepsilon_{\text{idx}}^{(1)}$ through the generic

hybrid $\varepsilon_{\text{idx}} \leq Q \cdot \varepsilon_{\text{idx}}^{(1)}$, a statement about the index PIR alone and not a cost of our construction.

Running time. Each $\mathcal{B}_i$ runs $\mathcal{A}$ once and adds only public bookkeeping around it: evaluating the candidate map, driving $d - 1$ simulated segments, and splitting each batch by segment. That overhead is a fixed polynomial in the public parameters, so $T' \approx T$.

The reduction carries the update oracle through rather than fixing a state, so the bound holds under mutations, with the static keyword scheme of Figure 5 read without its mutation phase as the special case of the empty update history. Composing with Lemma 5 makes RisePIR-F and RisePIR-S secure under decisional LWE [33]. $\square$

A.2 Proof of the incrementality reduction

Lemma 4 (Incrementality reduction). *Built on an updatable UIPIR, the construction of Figure 5 is incremental.*

Proof (of Lemma 4). Definition 6 asks two things of the mutation phase on a batch of τ operations. Its communication and its computation must both be proportional to τ and asymptotically below the cost of a full rebuild through Setup. We establish them in turn, and both rest on one fact about the KV-SCF, that a single mutation disturbs a constant number of buckets.

The footprint of one mutation. Let $\text{op} = (\text{type}, k, v)$ be one operation of a well-formed batch and let W be the write log the KV-SCF returns for it (Figure 4). An **update** finds the slot of $\text{Candidates}(k)$ carrying $\text{fp}(k)$ and overwrites its value field, and a **delete** finds the same slot and clears it. Each disturbs one slot, hence one bucket, in one segment. A successful **insert** either writes a free slot of a candidate bucket, disturbing one bucket, or runs an eviction walk. Each step of that walk rewrites one slot and moves the displaced fingerprint into a bucket of its own candidate set. The walk lasts at most MaxKicks steps, so the insertion disturbs at most $\text{MaxKicks} + 1$ buckets, spread over the d segments. An **insert** that spends its budget without settling undoes every eviction and returns with $W = \emptyset$, so it adds nothing to the batch, and rebuilding the filter at a larger capacity is a fresh Setup outside the mutation phase. In every case

$$|W| \leq \text{MaxKicks} + 1,$$

and MaxKicks is a filter parameter fixed at **Initialize**, before the store is built and independent of the number of pairs m . The rest of the proof carries this constant through.

Communication. Step 2 of DBMutation turns each bucket that W touched into one record overwrite in the segment holding it, so a batch of τ operations yields at most $(\text{MaxKicks} + 1)\tau$ overwrites, distributed among the d per-segment batches with $\sum_{j \in [d]} |\text{ops}_j| \leq (\text{MaxKicks} + 1)\tau$. A mutation is thus a sparse, position-predictable batch of index PIR updates. The UIPIR returns the transcript S_j for ops_j , whose size is proportional to $|\text{ops}_j|$ and asymptotically below what rebuilding that segment would publish (Definition 3). Summing over

the segments, $\text{trans} = (S_j)_{j \in [d]}$ is proportional to τ with a constant fixed by MaxKicks and the record width, and free of m . A keyword rebuild reruns **Setup**, which rebuilds the filter and every segment’s hint and so publishes at least what the d segment rebuilds do, which carries the comparison from Definition 3 over to Definition 6. That is the communication clause. A data owner that ships the current hint whenever the hint is smaller than the transcript only sends less, so the bound stands for that variant too.

Computation. **DBMutation** first applies the batch to the KV-SCF. Each operation scans the $d \cdot b$ slots of its candidate buckets, and an **insert** repeats that scan through **AltBuckets** at every step of its eviction walk, so the filter work is $O(\text{MaxKicks} \cdot d \cdot b)$ per operation, a constant in m , and $O(\tau)$ over the batch. It then converts W into the per-segment batches ops_j , a linear pass over at most $(\text{MaxKicks} + 1)\tau$ records, and hands each batch to **UIPIR.DBMutation**. **HintUpdate** hands each transcript S_j to **UIPIR.HintUpdate** and does nothing else. Both index PIR algorithms run in time proportional to the batch they receive and asymptotically below a rebuild of their segment (Definition 3), and those batches sum to $O(\tau)$.

The mutation phase therefore costs $O(\tau)$ filter work plus whatever the index PIR charges for $O(\tau)$ record overwrites, both proportional to τ and asymptotically below re-running **Setup** on the whole store. That is Definition 6, so the construction is incremental. The keyword layer contributes only the constant $\text{MaxKicks} + 1$, and the index PIR sets everything else.

The bound at our instantiations. The generic statement leaves the cost of one record overwrite to the index PIR. Under the entry-level hint patch of Figure 7, which both **UIPIR** schemes of Section 6.1 use, **UIPIR.HintUpdate** costs n additions and n multiplications in $\mathbb{Z}_q$ per rewritten cell, hence $n\tau w$ of each over a batch of **updates** and **deletes** and at most $(\text{MaxKicks} + 1)n\tau w$ once the batch holds **inserts** (Section 6.2), with no dependence on the segment size s . A keyword mutation is then independent of m outright, not merely asymptotically below a rebuild, which is what Section 7.3 measures. $\square$

B Parameter selection in detail

Section 6.3 states the two targets and the parameters that meet them. This appendix shows the work behind both. Appendix B.1 reports what the lattice estimator returns for the two LWE instances and why those numbers place both at NIST Category 1. Appendix B.2 splits the correctness budget, explains what the fingerprint width $f = 64$ buys, and derives the plaintext modulus p from the correctness statements of **FrodoPIR** and **SimplePIR**.

B.1 The LWE instances and their estimated security

Security rests entirely on the LWE instance each index PIR runs over, so we pin each instance’s dimension n to our target and report what the estimate returns there. Both instances use the modulus $q = 2^{32}$, the native **u32** word, which leaves n and the secret and error distributions as the free parameters.

Table 5. The two LWE instances and what the lattice estimator returns for each. RisePIR-F runs at $n = 1566$ with a ternary secret and error, meaning uniform on $\{-1, 0, 1\}$, at $m \leq 2^{23}$ exposed samples. RisePIR-S runs at $n = 1275$ with a uniform secret over $\mathbb{Z}_q$ and a discrete-Gaussian error of width $\sigma = 6.4$ at $m = \infty$. Both fix $q = 2^{32}$. Each entry is $\log_2$ of the cheapest attack cost across the estimator’s whole portfolio under that cost model, with the winning attack and its block size β beside it. The thresholds are the NIST Category 1 gate bound, our own core-SVP target, and the quantum core-SVP level their specification reports for Kyber-512.

Cost model	RisePIR-F over FrodoPIR $n = 1566$, ternary			RisePIR-S over SimplePIR $n = 1275$, Gaussian $\sigma=6.4$		
	attack	β	$\log_2$ cost	attack	β	$\log_2$ cost
MATZOV gates ($\geq 2^{143}$)	dual-hybrid	433	154.5	BDD	431	153.0
core-SVP classical ($\geq 2^{128}$)	dual-hybrid	439	128.2	uSVP	439	128.2
core-SVP quantum ($\geq 2^{107}$)	dual-hybrid	440	116.8	uSVP	439	116.3

How the estimate is read. A concrete LWE estimate has two halves that are easy to confuse. An *attack* fixes a lattice block size β , and a *cost model* turns that β into a bit count. We run the lattice estimator of Albrecht, Player, and Scott [2] at a pinned commit, take the cheapest attack across its whole portfolio, and report that minimum under three cost models. The minimum over the whole portfolio, rather than a single named attack, is what makes the estimate cover both secret distributions, since a uniform secret and a ternary secret are bound by different attacks.

The three cost models answer three different questions.

- The *MATZOV gate count* [27] is the realistic classical accounting, and it is the model NIST Category 1 is stated in. Category 1 asks for at least 2^{143} classical gates, the cost of AES-128 key search, so this is the binding check.
- The *core-SVP* model of ADPS16 [4] charges one sieving call in block size β at $2^{0.292\beta}$ classical operations, ignores everything else, and so understates the true cost by design. It is a deliberate floor rather than an estimate of real work, so an instance that clears 128 bits under it clears at least as much under any sharper accounting. This is the methodology FrodoKEM and FrodoPIR adopt [30,14], and we adopt it as a self-imposed target stricter than Category 1 itself.
- The *quantum core-SVP* model charges the same sieving call at $2^{0.265\beta}$. It is a floor in the same sense and, importantly, it is not a gate count, so we use it only to compare against the standardized reference schemes and never against Category 1’s quantum bar, which is stated in gates.

The result. Table 5 gives the two instances. Both clear all three bars. The gate counts, $2^{154.5}$ and $2^{153.0}$, sit more than ten bits above the 2^{143} Category 1 bound, which is what places both schemes at Category 1. Both land at $2^{128.2}$ classical core-SVP, a fraction of a bit above our own 2^{128} target, so neither instance carries

slack against the stricter of the two bars. The quantum core-SVP figures, $2^{116.8}$ and $2^{116.3}$, sit above the 2^{107} their specifications report for Kyber-512 and below the 2^{130} for FrodoKEM-640 [6,30], so on the conservative floor both instances are more demanding than the Category 1 reference KEM.

The two schemes need different dimensions because their noise differs. SimplePIR draws a discrete-Gaussian error of width $\sigma = 6.4$, whose unbounded tail buys security per dimension, so $n = 1275$ suffices. FrodoPIR draws a narrow ternary secret and error, which carry less noise per sample and demand $n = 1566$, above the $n = 1228$ that FrodoPIR itself computes for plain 128-bit ternary LWE [14]. The same difference decides the binding attack, the primal uSVP and BDD attacks against SimplePIR’s uniform secret and the dual-hybrid attack against FrodoPIR’s ternary one.

The sample cap that RisePIR-F carries. One modeling choice deserves to be stated rather than buried, because it is a deployment constraint and not a formality. The estimate exposes m LWE samples, and m is the query length, one sample per database row a query ranges over. For every lattice attack more samples only help the attacker, so $m = \infty$ looks like the safe setting, and for SimplePIR it is both safe and faithful, since a Gaussian error defeats the algebraic Arora-Ge attack [5] at any sample count.

For FrodoPIR the same setting would be unfaithful. Its ternary error is bounded, and bounded-error LWE with unlimited samples falls to Arora-Ge: at $m = \infty$ the estimator drops that attack to 2^{60} , far below anything else in the portfolio. The scheme never exposes unlimited samples. It exposes exactly one per row of the segment a query ranges over, and Table 6 shows how the attack’s cost tracks that number.

Table 6. Cost of the Arora-Ge attack [5] against the FrodoPIR instance at $n = 1566$, as a function of the exposed sample count m . Every other attack in the portfolio is unchanged across this range, since the lattice attacks draw on about 1500 samples whatever m allows.

m	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{26}
$\log_2$ Arora-Ge cost	304	253	197	169	141	92
clears 2^{143} ?	yes	yes	yes	yes	no	no

We therefore cap the instance at $m \leq 2^{23}$, which is the ceiling this dimension supports: at $m = 2^{24}$ the attack costs 2^{141} and falls through the gate bound. The parameters of Table 5 hold for RisePIR-F at segment size and query length $R \leq 2^{23}$, and a larger segment needs either a larger n or a Gaussian error. The largest RisePIR-F segment we report is 2^{18} buckets (Section 7.3), five bits inside the cap, so the constraint binds nothing here and is recorded for anyone deploying at a larger scale. Nothing else in the estimate moves with the cap.

Why we do not report a quantum gate count. Category 1’s quantum bar is stated in gates, and the estimator has no quantum gate-count model for sieving, so no honest quantum gate number exists to report. We do not invent one. The argument that discharges the quantum bar is the one FrodoKEM and Kyber use: AGPS20 finds the quantum speedup for lattice sieving marginal [1], so the classical gate count governs the security level, and an instance that clears the classical bar clears the quantum bar for every depth budget Category 1 admits. The quantum core-SVP column of Table 5 is reported as a conservative floor and compared only against the reference schemes’ figures in the same model.

B.2 The correctness budget

Correctness is two-sided here, a wrong value for a present key bounded by δ and a false hit for an absent one bounded by ξ (Definition 4), and we target 2^{-40} on both. By Lemma 2 each bound is a sum of two independent terms, one charged to the filter and one to the index PIR,

$$\delta, \xi \leq \underbrace{d \cdot \delta_{\text{idx}}}_{\text{index term}} + \underbrace{\left(1 + \frac{\tau_{\text{ins}}}{m}\right) d \cdot b \cdot 2^{-f}}_{\text{filter term}},$$

writing the filter term in the generic form that covers both sides. We split the budget evenly and ask each term for 2^{-41} , which gives $\delta, \xi \leq 2^{-40}$. The two halves are then independent design problems: the filter term is settled by the fingerprint width f alone, and the index term by the plaintext modulus p alone.

The fingerprint width. The filter term clears 2^{-41} once

$$f \geq 41 + \log_2(d \cdot b) + \log_2\left(1 + \frac{\tau_{\text{ins}}}{m}\right).$$

At the widest shape Section 7.2 admits, $d \cdot b = 9$, and at the most insert-heavy batch we measure, $\tau_{\text{ins}}/m \approx 0.011$, this reads $f \geq 44.19$. So $f = 45$ would already suffice, and $f = 46$ would cover every shape up to $d \cdot b \leq 16$.

We take $f = 64$ instead, for two reasons that have nothing to do with the bound. A 64-bit fingerprint keeps the slot arithmetic byte-aligned, and it matches the fingerprint width $\text{KPIR}^{\text{index}}$ already uses, which keeps the comparison of Section 7.3 like for like. What the extra bits buy is worth naming: at $f = 64$ the filter term sits between $2^{-62.1}$ and $2^{-61.0}$ at every configuration we report, about twenty bits below its share of the budget. The filter term is then negligible against the index term, and the whole 2^{-40} budget is effectively handed to the index PIR. That is the useful consequence, because it makes the correctness analysis a statement about the index PIR alone. At $f = 32$ the same term sits near 2^{-29} and would dominate the budget outright.

From the budget to the plaintext modulus. One keyword query makes d index PIR reads, and one read recovers one bucket of one segment. The index PIR holds that bucket as one record, a row of

$$w(p) = b \cdot \left\lceil \frac{f + \ell}{\log_2 p} \right\rceil$$

cells over $\mathbb{Z}_p$ in the notation of Section 6.1, where ℓ counts the bits of a value. A read is correct only if every one of those cells decodes correctly, so a union bound over the cells and then over the d reads turns the index half of the budget into a per-cell requirement,

$$\delta_{\text{cell}} \leq \frac{2^{-41}}{d \cdot w(p)}. \quad (7)$$

Note that p appears on both sides, since a smaller modulus packs a bucket into more cells and so tightens the per-cell requirement. We therefore scan p downwards from 2^{14} and take the largest power of two that satisfies Eq. (7), the largest plaintext modulus that still decodes. Taking the largest is what keeps the answer small, since a bucket occupies $w(p)$ cells.

What remains is to bound δ_{cell} for each scheme. Both bounds come from the correctness statements of the schemes we build on, which we restate in our notation and retarget. In both, the noise a single cell accumulates is summed over the ρ rows of the processed database $\mathbf{D}$ of Section 6.1, which is the segment's bucket count for **FrodoPIR**, whose reshape is the identity, and the near-square row count for **SimplePIR**. The distinction matters: at our shapes **SimplePIR**'s reshape brings ρ down by a factor of 6–33, and reading the bound at the segment's bucket count instead would misstate the tail badly.

SimplePIR. Theorem C.1 of [22] gives the condition directly in terms of a chosen correctness error δ : at plaintext modulus p over a processed database of ρ rows with Gaussian error of width σ , decoding fails with probability at most δ provided

$$\Delta = \left\lfloor \frac{q}{p} \right\rfloor \geq \sqrt{2} \cdot \sigma \cdot p \cdot \sqrt{\rho} \cdot \sqrt{\ln(2/\delta)}.$$

Their parameter selection instantiates this at $\delta = 2^{-40}$. We keep the bound and move δ , instantiating it at the per-cell budget of Eq. (7) instead, and we double the leading constant because our cells hold values in $[0, p)$ where their bound assumes the centered range.

FrodoPIR. Theorem 4.1 of [14] states the condition as $q \geq 8p^2\sqrt{\rho}$ and calls the scheme correct with high probability. That form cannot be retargeted, because it names no failure probability to move. Its proof bounds $\|\mathbf{e} \cdot \mathbf{D}\|_\infty \leq 4p\sqrt{\rho}$, and evaluating that tail explicitly shows why the qualitative phrasing matters: at that threshold a Hoeffding bound gives about $6.7 \cdot 10^{-4}$ per cell, which is nowhere near the 2^{-51} – 2^{-54} that Eq. (7) asks for at our shapes.

Table 7. The operating point of every configuration of Tables 3 and 4, at $f = 64$ and $q = 2^{32}$. w is the cells of the bucket a read must decode, “target” is the per-cell budget of Eq. (7) and “achieved” what Eq. (8) or the retargeted SimplePIR bound delivers at the chosen p , both as $\log_2$. The last column is κ with $\delta, \xi \leq 2^{-\kappa}$, against the target of $\kappa = 40$.

(d, b)	ℓ	scheme	p	w	target	achieved	κ
(2, 4)	256 B	RisePIR-F	2^9	940	-51.88	-536.21	60.98
(2, 4)	256 B	RisePIR-S	2^9	940	-51.88	-107.20	60.98
(2, 4)	1024 B	RisePIR-F	2^9	3672	-53.84	-536.21	60.98
(2, 4)	1024 B	RisePIR-S	2^8	4128	-54.01	-864.59	60.98
(4, 1)	256 B	RisePIR-F	2^9	235	-50.88	-271.74	61.98
(4, 1)	256 B	RisePIR-S	2^9	235	-50.88	-147.77	61.98
(4, 1)	1024 B	RisePIR-F	2^9	918	-52.84	-271.74	61.98
(4, 1)	1024 B	RisePIR-S	2^9	918	-52.84	-75.64	61.62
(4, 2)	256 B	RisePIR-F	2^9	470	-51.88	-536.21	60.98
(4, 2)	256 B	RisePIR-S	2^9	470	-51.88	-152.27	60.98
(4, 2)	1024 B	RisePIR-F	2^9	1836	-53.84	-536.21	60.98
(4, 2)	1024 B	RisePIR-S	2^9	1836	-53.84	-71.13	58.08

We therefore bound the same quantity explicitly. Decoding one cell is correct exactly when $|\langle \mathbf{e}, \mathbf{D}[\cdot][c] \rangle| < \Delta/2$. The error $\mathbf{e} \in \{-1, 0, 1\}^p$ is uniform ternary and independent per coordinate with variance $2/3$, and a database cell has magnitude at most p . The terms $X_i = \mathbf{e}_i \cdot \mathbf{D}[i][c]$ are therefore independent, zero-mean, bounded by p , and carry total variance $V \leq \frac{2}{3}pp^2$. Bernstein’s inequality at $t = \Delta/2$ gives

$$\delta_{\text{cell}} \leq 2 \exp\left(-\frac{t^2}{2V + \frac{2}{3}pt}\right), \quad t = \frac{\Delta}{2} = 2^{31 - \log_2 p}, \quad (8)$$

against a worst-case database. Hoeffding’s inequality on the same quantity is looser and selects the same p at every configuration we report, so nothing here turns on the choice between them.

The resulting operating points. Table 7 runs the rule over every configuration we report. It returns $p = 2^9$ everywhere but one cell, RisePIR-S at (2, 4) and $\ell = 1024$ B, which drops to $p = 2^8$ because $p = 2^9$ misses the bound by under one percent. That single cell is why the selection has to be evaluated rather than estimated.

Two things are worth reading off the table. The achieved per-cell error clears its target by a wide margin everywhere, by hundreds of bits under FrodoPIR’s ternary noise and by anywhere from 17 to 810 bits under SimplePIR’s Gaussian, because p moves in powers of two and the last passing power is usually far inside the bound. The widest of those margins is the one cell that dropped to $p = 2^8$. And the resulting κ sits between 58 and 62 at every configuration, so every number we report holds at $\delta, \xi \leq 2^{-58}$, comfortably inside the 2^{-40} we claim.

The claim is stated at 2^{-40} because that is the target the rule is written against, not the margin it happens to achieve.

What the target costs. Since 2^{-40} is a convention rather than a derived requirement (Section 6.3), it is fair to ask what a stricter one would cost here, and re-running the selection rule answers that directly. At $\delta, \xi \leq 2^{-128}$, RisePIR-F keeps $p = 2^9$ at every configuration of Table 7, because its ternary noise already clears the budget by hundreds of bits and the 88 bits of extra demand do not consume that margin. RisePIR-S drops from $p = 2^9$ to $p = 2^8$ at three of the six configurations, leaving four of the six at $p = 2^8$, which costs one bit of plaintext modulus and about an eighth more cells per bucket. So the target RisePIR pays for is not one that cost forces, which is why we treat 2^{-40} as inherited rather than derived and say so in Section 6.3.

Reproducing the selection. Both halves of this appendix are mechanical and we ran them as such. The security estimates come from the lattice estimator⁴ under the three cost models of Table 5, taking the minimum over its whole attack portfolio. The operating points of Table 7 come from the scan described above, which evaluates Eq. (8) and the retargeted SimplePIR bound at each candidate p and checks that the passing set is downward closed, so that the chosen p is the maximum and not merely a value that works. That selector is part of the implementation, with unit tests pinning every row of Table 7.

⁴ <https://github.com/malb/lattice-estimator>, pinned at commit 3e48ef4.