Incremental Offline/Online PIR (extended version)*

Yiping Ma[§] Ke Zhong[§] Tal Rabin[§†] Sebastian Angel[§‡]

[§]*University of Pennsylvania* [†]*Algorand Foundation* [‡]*Microsoft Research*

Abstract. Recent private information retrieval (PIR) schemes preprocess the database with a query-independent offline phase in order to achieve sublinear computation during a query-specific online phase. These offline/online protocols expand the set of applications that can profitably use PIR, but they make a critical assumption: that the database is immutable. In the presence of changes such as additions, deletions, or updates, existing schemes must preprocess the database from scratch, wasting prior effort. To address this, we introduce *incremental preprocessing* for offline/online PIR schemes, allowing the original preprocessing to continue to be used after database changes, while incurring an update cost proportional to the number of changes rather than the size of the database. We adapt two offline/online PIR schemes to use incremental preprocessing and show how it significantly improves the throughput and reduces the latency of applications where the database changes over time.

1 Introduction

Private information retrieval (PIR) [22] is an incredibly useful cryptographic building block that allows a client to download an object from a database without revealing which object was fetched. PIR has many applications, including privacy-preserving video streaming [3, 35], password checking [4], blocklists [40], ad delivery [34], friend discovery [12], subscriptions [20], and anonymous messaging [6, 42, 48]. While powerful, PIR is expensive: PIR imposes a computational linear lower bound since the database must operate on all objects in order to answer a query. After all, if even a single object is omitted when answering a query this would leak that this object is of no interest to the client.

The existence of this computational lower bound is unfortunate, as it limits the scale of applications that can profitably use PIR. But there is still hope! There is an active line of work [5, 8, 15, 18, 23, 45, 51, 52] that pursues the idea of preprocessing the database to generate auxiliary information or *hints* (which could be stored at the server or clients depending on the proposal) during an *offline* phase, and then use the hints during an *online* phase to answer one or more queries with sublinear computation and communication. This is an exciting proposition, as it enables applications that require quick answers for (online) queries to large databases, but that can afford an expensive query-independent preprocessing.

A major implicit assumption in all of the above works is that the PIR database is *immutable*. That is, once it has been preprocessed, no new items will be added, deleted, or updated. Should the database change, under all existing schemes, it is necessary to redo the expensive offline phase—defeating in many cases the benefits of preprocessing. Meanwhile, many of the proposed applications that use PIR naturally experience at least some moderate content churn. For example, Popcorn [35], which implements a private video service where clients stream movies using PIR, must deal with movies being added, deleted, or modified (e.g., to change codec) occasionally. Similarly, anonymous messaging systems like Pung [6] and Talek [20] have databases where new elements (messages) are added every few minutes, while contact discovery services like DP5 [12] have users creating and deleting accounts.

This paper extends offline/online PIR schemes to support *mutable* databases where the content can change at any time by introducing the notion of *incremental preprocessing*. The result is an incremental offline/online PIR scheme where additions, deletions, and edits to the database do not require a complete preprocessing after every change. Instead, the hints are updated at a cost that is proportional to the number of changes. To demonstrate the feasibility and benefits of incremental preprocessing, we extend two recent two-server offline/online PIR protocols: (1) the Corrigan-Gibbs and Kogan scheme (CK) [23], and (2) the Shi, Aqeel, Chandrasekaran, and Maggs scheme (SACM) [52]. Our experimental evaluation confirms that the savings of our approach are considerable. In a database with 1 million items, the computational cost of updating the hints in our incremental CK scheme (iCK) for a batch of 10,000 updates (additions, deletions, edits) is $56\times$ cheaper than preprocessing from scratch; the savings are even more pronounced when there are fewer updates. Furthermore, an implementation of PIR-Tor [48] that uses our iCK construction improves the throughput achieved by Tor directory nodes by roughly $7\times$ over an implementation of PIR-Tor that uses a state-of-the-art 2-server PIR scheme [13].

Our extensions to make these schemes incremental, however, come at a modest cost. For iCK, the cost is a small increase in storage at the client to keep additional auxiliary material needed for the client to construct the right query during the online phase, and a larger (though still sublinear and concretely very efficient) queries during the online phase. For our incremental version of SACM, called iSACM, the cost is higher online communication (though still sublinear in the size of the updated database), since our construction is

*This is the full version of our paper [47]. It includes additional details, experiments, and another PIR protocol (Appendices B–E).

not compatible with a specific cryptographic primitive used in SACM (a private puncturable PRF [52]).

Limitations. Our incremental preprocessing schemes work best when the database changes slowly (e.g., a few percent of entries are added, deleted, or updated at a given time). When the database changes significantly (e.g., doubles or triples in size), maintaining multiple preprocessed databases and querying all of them might be a better choice [40]. We also note that there is an interesting complication that occurs when one wishes to use *PIR by keywords* [21] with offline/onlne PIR protocols that support mutations. In PIR by keywords, clients can fetch an object from the database by using a meaningful label or "keyword" rather than an index. Prior works [4, 6, 21] accomplish this by structuring the database in a search data structure (e.g., binary search tree) and performing the search obliviously over this data structure using a PIR scheme as a black box. However, when the database mutates, this triggers a series of changes to the underlying search data structure (e.g., BST tree rotations) that negate the validity and usefulness of the hints generated during the offline preprocessing phase of existing offline/online PIR schemes. Finding a satisfying solution remains an open question.

2 Background and related work

This section surveys existing offline/online PIR constructions, and the issues that arise when the database is updated due to objects being deleted, edited, or inserted.

Private Information Retrieval (PIR) Chor et al. [22] introduce private information retrieval to allow a client to retrieve an object from a database managed by an untrusted server (or set of servers) without the server(s) learning which object was retrieved by the client and with total communication costs that are *sublinear* in the size of the database. There are two general deployment models for PIR: *multi-server* and *single-server*. In multi-server PIR [7, 13, 22, 24, 25, 31, 49] the database is replicated across two or more non-colluding servers, and the client issues a query to each server and locally combines all responses. On the other hand, single-server PIR protocols [3–5, 16, 17, 19, 27, 29, 38, 41, 43, 44, 53] are significantly more (computationally) expensive and require making cryptographic hardness assumptions. In exchange, they avoid making non-collusion assumptions.

The allure of preprocessing. Beimel et al. [8, §4] prove that if the database has no redundancies (i.e., neither the servers nor the client store any redundant bits to serve as auxiliary values), PIR requires $\Omega(n)$ total server-side computation to answer a single client query, where n is the number of elements in the database. To circumvent this unfortunate lower bound, they introduce the notion of *preprocessing*: by pushing the inevitable linear computation to a query-independent *offline phase* and generating auxiliary data or *hints* along the way, the computational cost of answering a query during an *online phase* with the help of the hints can be sublinear in n.

Given the above, a natural question is *where* the hints should be stored? Several works [8, 15, 18] propose that they be stored at the servers. For example, Beimel et al.[8] give a two-server PIR protocol that uses $O(n^{1+\epsilon})$ extra bits at the servers in exchange for online queries that require $O(n/\epsilon^2 \log^2 n)$ server computation. On the other hand, recent proposals [23, 51, 52] push the hints to the clients. For instance, Corrigan-Gibbs and Kogan [23] give a two-server PIR construction that stores $O(\sqrt{n})$ bits at the client and obtain $O(\sqrt{n})$ server computation for online queries.

The challenge of mutability. Regardless of who stores the hints, all of the above works have preprocessing schemes where the hints depend on *all* items in the database. As a result, if a single items changes, is deleted, or a new item is added, the hints are no longer useful to generate online PIR queries; it becomes necessary to update the hints. To our knowledge, none of the existing works propose a way to update these hints in an efficient fashion that avoids redoing the preprocessing from scratch.

In independent work, Checklist [40] explores the idea of an offline/online PIR database that supports additions. Despite the similar objectives, our approaches are fundamentally different. Checklist leverages ideas from ORAM [33], dividing the database into "buckets" whose capacity grows exponentially, where the initial data is held on the last (largest) bucket, and new objects are added to the earliest (smallest) bucket—overflowing to larger buckets if necessary. To fetch an object, the client queries all of the roughly $\log n$ buckets (as otherwise the client would leak which bucket contains the desired item). The key idea behind this scheme is that while updates to a bucket require redoing the preprocessing for that bucket from scratch to obtain the new hints, most updates impact smaller buckets and larger buckets change less frequently—hence the savings. In contrast, our work aims to make the preprocessing itself incremental; our techniques could help Checklist ensure that larger buckets need not be preprocessed from scratch. Furthermore, our technique does not require the online server to maintain $\log n$ buckets and does not increase the number of objects fetched by the client per query—which could be significant when objects are large.

3 Overview

In this work we consider a setting in which a PIR server (or servers) holds a large database consisting of n fixed-sized objects. The server preprocesses the database during an offline phase to generate *hints* (auxiliary information that speeds up the online phase) and either stores these hints or gives them to clients (depending on the scheme). Clients can then query this preprocessed database during an online phase and retrieve an item privately with both computation and communication costs sublinear in n. Over time, the database mutates: new objects are added and existing objects are modified or deleted. We will assume that the number of mutations at a given time,

m, is much smaller than n (otherwise one might as well rerun the preprocessing on the modified database from scratch). Our goal is to construct preprocessing schemes that (1) enable efficient online queries; and (2) are incremental, in the sense that updating the hints comes at a cost to the servers and clients that is proportional to m rather than $n + m$.

In the rest of this section we give a formal definition of offline/online PIR, discuss the additional functionality that we seek to support mutable databases, and then give concrete incremental preprocessing schemes for two existing offline/online PIR constructions.

3.1 Offline/Online PIR (OO-PIR)

We build on the definitions of the offline/online PIR (OO-PIR) model given by Corrigan-Gibbs and Kogan [23]. In particular, we restrict our focus to two-server PIR (we discuss the single server case in Section 9). In this setting there are two servers: the *offline* and the *online* server. The servers are *semi-honest*: they do not collude but are interested in learning which objects the client is fetching from the database. Below we give the definition for the single-query case; we discuss multiple queries in Section 4.1.

Notation. We consider a database D, which is replicated across both the offline and online servers and consists of n items of size b bits. We view D as an array; $D[i]$ represents the i-th item in D. We use $[x, y]$ to denote the set consisting of consecutive integers $\{x, \ldots, y\}$ where $x \leq y$. For simplicity, when $x = 1$, we use $[y]$ to denote the set $\{1, \ldots, y\}$. For a set S and an integer c, we use $S+c$ to denote a set $\{x+c | x \in S\}$. In addition, we assume $+$ has the highest precedence among all set operations.

Definition 1 (Offline/Online PIR [23])**.** An OO-PIR protocol consists of four algorithms $(\mathsf{Prep}, \mathsf{Query}, \mathsf{Resp}, \mathsf{Recov})$ defined over a database D of n items as follows:

- $\mathsf{Prep}(D) \to h$, a randomized algorithm executed by the offline server for each client that takes a database D and outputs a hint h.
- $\mathsf{Query}(h, i) \to q_i$, a randomized algorithm executed by the client that takes in h and the desired index i, and outputs a query q_i.
- $\mathsf{Resp}(D, q_i) \to r_i$, a deterministic algorithm executed by the online server that takes in a query q_i from the client, and outputs a response r_i.
- $\mathsf{Recov}(h, r_i) \to d_i$, a deterministic algorithm executed by the client that takes in a hint h (previously received from the offline server) and a response r_i from the online server, and outputs the client's desired data object d_i.

An OO-PIR scheme should satisfy the following properties:

Correctness. For every $\lambda, n \in \mathbb{N}, i \in [n]$, we require that

$$\Pr\left[d_i = D[i]: \begin{array}{rcl} h & \leftarrow & \mathsf{Prep}(D) \\ q_i & \leftarrow & \mathsf{Query}(h, i) \\ r_i & \leftarrow & \mathsf{Resp}(D, q_i) \\ d_i & \leftarrow & \mathsf{Recov}(h, r_i) \end{array} \right] \geq 1 - \mathrm{negl}(\lambda)$$

where the probability is over the randomness of all algorithms.

Security. For $\lambda, n \in \mathbb{N}$, and $i \in [n]$, define the distribution

$$\mathcal{P}(i) := \left\{ q_i : \begin{array}{rcl} h & \leftarrow & \mathsf{Prep}(D) \\ q_i & \leftarrow & \mathsf{Query}(h, i) \end{array} \right\}$$

An OO-PIR scheme is secure if for all PPT adversaries $\mathcal{A}$,

$$\max_{i,j \in [n]} \{\Pr[\mathcal{A}(\mathcal{P}(i)) = 1] - \Pr[\mathcal{A}(\mathcal{P}(j)) = 1]\} \leq \mathrm{negl}(\lambda)$$

Non-triviality. The combined size of the hint h, online query q_i, and response r_i should be sublinear in the size of D.

Informally, correctness means that a client can get its desired item with high probability; security means that any two queries are computationally indistinguishable to the online server; and non-triviality means that the protocol is actually cheaper than downloading and storing the whole database.

3.2 Incremental offline/online PIR

In this work we propose OO-PIR schemes that are *incremental*, in the sense that if the database mutates (new items are added, or existing items are modified or deleted) it is not necessary for the client and the offline server to rerun Prep. To that end, the OO-PIR scheme supports four new algorithms that incrementally modify the existing hint h so that it is compatible with the new database. Crucially, the cost of these algorithms depends on the number of mutations and not on the size of the original database.

Definition 2 (Incremental offline/online PIR)**.** An incremental OO-PIR protocol consists of eight algorithms, four of which are inherited from an OO-PIR scheme (Definition 1) and four new algorithms defined as follows:

- $\mathsf{DBUpd}(D, op) \to (D', \delta)$, a deterministic algorithm executed by offline server that takes in the original database D, and a set of operations op (additions, deletions, and in-place edits at certain positions), outputs a new database D' with size n' and a summary of the changes, δ. We discuss δ in detail in Section 6, but it is small and does not include the objects themselves. Note that the online server also updates D with op, but does not produce δ or interact with clients during offline phase.
- $\mathsf{HintReq}(h, \delta) \to u_q$, a randomized algorithm executed by the client that takes as input the hint that the client had previously obtained from the offline server and the update summary δ, and outputs an *update query* u_q.
- $\mathsf{HintRes}(D', u_q) \to u_r$, a deterministic algorithm executed by the offline server that takes in the new database

D' and an update query u_q from the client, and outputs an *update response* u_r. The cost of this algorithm should be proportional to the number of changes between D and D'.

- $\mathsf{HintUpd}(h, u_r) \to h'$, a deterministic algorithm executed by the client that takes in a hint h and a update response u_r from the offline server, and outputs a new hint h' that is valid with respect to the new database D'.

An incremental OO-PIR scheme should satisfy the following correctness, security, and non-triviality properties.

Correctness. For every $\lambda \in \mathbb{N}, i \in [n']$, we require that

$$\Pr\left[\begin{array}{l} d_i = D'[i]: \end{array} \begin{array}{rl} h \leftarrow & \mathsf{Prep}(D) \\ (D', \delta) \leftarrow & \mathsf{DBUpd}(D, op) \\ u_q \leftarrow & \mathsf{HintReq}(h, \delta) \\ u_r \leftarrow & \mathsf{HintRes}(D', u_q) \\ h' \leftarrow & \mathsf{HintUpd}(h, u_r) \\ q_i \leftarrow & \mathsf{Query}(h', i) \\ r_i \leftarrow & \mathsf{Resp}(D', q_i) \\ d_i \leftarrow & \mathsf{Recov}(h', r_i) \end{array}\right] \geq 1 - \mathrm{negl}(\lambda)$$

where the probability is over the randomness of all algorithms. The gray boxes show the new incremental operations.

For simplicity, we will use the notation

$$h' \leftarrow \mathsf{IncPrep}(D, op, h)$$

for the four hint update algorithms when there is no ambiguity.

Security. For $\lambda \in \mathbb{N}$, and $i \in [n']$, define the distribution

$$\mathcal{P}'(i) := \left\{ q_i : \begin{array}{rl} h \leftarrow & \mathsf{Prep}(D) \\ h' \leftarrow & \mathsf{IncPrep}(D, op, h) \\ q_i \leftarrow & \mathsf{Query}(h', i) \end{array} \right\}$$

An incremental OO-PIR scheme is secure if for all PPT adversaries $\mathcal{A}$,

$$\max_{i,j \in [n']} \{\Pr[\mathcal{A}(\mathcal{P}'(i)) = 1] - \Pr[\mathcal{A}(\mathcal{P}'(j)) = 1]\} \leq \mathrm{negl}(\lambda)$$

Non-triviality. The size of the initial hint h should be sublinear in n (the size of D), and the size of the online query q_i, response r_i, update summary δ, update query u_q, and update response u_r should be sublinear in n' (the size of D'). Finally, for a list of updates op, the computational cost of $\mathsf{IncPrep}$ to the offline server should be in expectation $O(|op| \log n)$.

3.3 Types of updates considered

We consider three types of mutations: addition of new objects, deletion of existing objects, and in-place edits that change the database's content but does not alter its size.

Additions. We aim to support databases where new items are appended to the end: if the initial database is of size n, then after m additions the database has size $n + m$ and the last m items are new. We make this restriction because supporting insertions at arbitrary locations is difficult without preprocessing the database from scratch as a single insertion changes the indexing of all subsequent objects.

In-place edits. After m in-place edits, the updated database is still of size n, and up to m of the n items have changed.

Deletions. Unlike the prior two operations, defining what it means for a database object to be deleted in an OO-PIR scheme is actually more subtle. We start by stating a trivial fact: if a client fetches an object from the database before the object is deleted, it is impossible to prevent the client from accessing this object again since the client could have saved a copy. Given this framing, we ask whether OO-PIR schemes can guarantee two more nuanced definitions of deletion:

- *Strong deletion.* Assuming that a client has not explicitly fetched a particular object in the past, an OO-PIR scheme provides strong deletion if it can guarantee that once this object is deleted from the server(s), the client will be unable to retrieve it. This is a desirable property in practice. For example, if the database operator learns that some of its objects are illegal or contain classified information, the operator may wish to delete them and prevent all clients who have not yet accessed these objects from doing so.
- *Weak deletion.* We relax the above definition to require only that *new* clients do not learn any deleted items.

We show that in OO-PIR schemes where the hint is stored at the client, strong deletion is very difficult to achieve (§4.2.3). The intuition is that the hint itself implicitly encodes information about all objects. Even if the client has not queried the deleted object explicitly in the past, the client can still recover it by querying *other* objects and reconstructing it with the existing hint. On the other hand, weak deletion is possible; we discuss our approach in the following sections.

3.4 Are Simple Solutions Good Enough?

Before discussing our design, we first consider a few simple approaches and describe their shortcomings. Suppose the servers add a few items to the database. The simplest approach is to merely let the client download and store all of the added items. This actually works relatively well in some settings, but when items are large this is problematic and in some applications it could leak information. For example, the server could infer whether the client is interested in the new items or not by observing changes in the client's query frequency. Another approach is to set up a new PIR database for the added items, preprocess that database, and give the client the new hints. When a client issues a query, it sends queries to both the old and the new databases, separately. This is roughly the approach taken by Checklist [40], with the caveat that Checklist carefully selects the sizes of the multiple databases.

However, this multi-database approach increases the online communication, since the client must send a query and receive a response from each of the databases.

In contrast, our solution does not require the client to store full items locally or to download multiple objects. However, our approach is not black-box. Instead, it requires exploiting the structure of the underlying OO-PIR protocol.

4 Incremental PIR: background and intuition

Our first solution builds on the two-server offline/online PIR scheme of Corrigan-Gibbs and Kogan (CK) [23]. We start by describing the CK protocol and then we describe our approach to make its preprocessing incremental.

4.1 Background: the CK protocol

We discuss the "toy protocol" given in CK [23] that conveys the key ideas of the scheme, but has high costs and violates PIR's non-triviality requirement. We later discuss how CK addresses these issues. The CK protocol has two phases.

Offline phase. This phase is independent of the index that the client wishes to query and includes the Prep step (§3.1).

- $\mathsf{Prep}(D) \to h$. The offline server generates $\sqrt{n}$ random subsets of indices in $[n]$, $S_1, \ldots, S_{\sqrt{n}}$, each of size $\sqrt{n}$, with the constraint that $S_1 \cup \cdots \cup S_{\sqrt{n}} = [n]$. For each S_j, the offline server uses the entries in S_j as database indices to compute $\sqrt{n}$ parities, $p_1, \ldots, p_{\sqrt{n}}$, where $p_j = \bigoplus_{e \in S_j} D[e]$ for $j = 1, \ldots, \sqrt{n}$. Each parity is b bits, which is the size of a database item. In short, a parity p_j for set S_j is the XOR of all objects in D referenced by the indices in S_j. The server sends back the hint $h = \{(S_1, p_1), \ldots, (S_{\sqrt{n}}, p_{\sqrt{n}})\}$ to the client.

Online phase. This phase depends on the specific index of the item that the client wishes to fetch and is performed with the online server who has no information about the subsets chosen by the offline server.

- $\mathsf{Query}(h, i) \to q_i$: The client generates a query for the item at index i in D (i.e., $D[i]$) by finding a set S (included in h) such that $i \in S$ (we denote the corresponding parity as p_S). Then, the client probabilistically removes an element from S: with probability $1 - \frac{\sqrt{n}-1}{n}$, it removes i; else it removes an element in $S \backslash \{i\}$ chosen uniformly at random. The query q_i is the resulting set S^*, which has size $\sqrt{n} - 1$. This query is sent to the online server.
- $\mathsf{Resp}(D, q_i) \to r_i$: The online server computes the PIR response as $r_i = \bigoplus_{e \in S^*} D[e]$, and sends r_i to the client.
- $\mathsf{Recov}(h, r_i) \to d_i$: Using the parity p_S (from the Query step), the client recovers $D[i]$ by computing $d_i = p_S \oplus r_i$.

The correctness of this scheme is probabilistic and follows from the fact that $p_S = \bigoplus_{e \in S} D[e]$ and $r_i = \bigoplus_{e \in S^*} D[e]$. With probability $1 - O(\frac{1}{\sqrt{n}})$, the set S^* is identical to S except that it is missing index i. In such case, the scheme is correct and d_i (the XOR of p_S and r_i) is $D[i]$. This scheme is secure against the offline server since the offline phase is independent of i. It is secure against the online server because q_i is a uniformly random subset of $[n]$ of size $\sqrt{n} - 1$.

In summary, this offline/online model pushes the expensive linear computation to the offline phase, while the online server only needs to do $O(\sqrt{n})$ computation.

Supporting multiple queries. To guarantee security against the online server, a set cannot be used more than once—if the online server sees two queries containing sets $S \backslash \{i\}$ and $S \backslash \{j\}$, computing the set difference results in the queried indices. Therefore, the client and the server have to redo the offline phase after the client issues a query. CK avoids this expense with two ideas. First, the use of independent random sets. Instead of the client generating and storing a hint h that consists of $\sqrt{n}$ sets and parities, where the sets have $\sqrt{n}$ elements with the restriction that $S_1 \cup \cdots \cup S_n = [n]$, the client represents h as $\sqrt{n} \log n$ independent random sets and their parities, where each set contains $\sqrt{n}$ indices sampled uniformly at random. As a result, with high probability, each index appears in at least one of the $\sqrt{n} \log n$ sets. Second, CK introduces a *refresh* operation that is performed between the client and the offline server after the client issues a query to the online server. This refresh operation is cheap: the cost to the offline server is simply $O(\sqrt{n})$. It consists of the client generating a new set with $\sqrt{n} - 1$ random indices and getting the corresponding parity from the server[1]. The client can then add the index of the item it had previously queried (i) to this set and update the parity (XOR with the queried item).

With the above two modifications, the client and the offline server can perform a single expensive offline phase, followed by refresh operations after every query. In this way, the expensive offline phase is amortized across multiple queries.

Reducing costs. In the above protocol, the client has to store and download from the offline server $n \log^2 n + b\sqrt{n} \log n$ bits: $\sqrt{n} \log n$ sets, each containing $\sqrt{n}$ indices, where each index is $\log n$ bits, and the corresponding $\sqrt{n} \log n$ parities, each of which is b bits. To ensure the scheme is non-trivial (i.e., has sublinear communication in n), CK uses a *puncturable pseudorandom set* (PPRS) [23] and assumes a computationally bounded adversary. With a PPRS, the client can use a cryptographic key k to represent (and generate) a pseudorandom subset S of $[n]$. The client can then puncture k to obtain k^*, which is a key that generates all but one element of S and which the client can send to the online server to generate S^*. This means that in the offline phase, the client needs only store (and upload to the offline server) $\sqrt{n} \log n$ cryptographic keys (one for each set), in addition to downloading and storing the parities. Consequently, the total communication and storage costs are $O(b\sqrt{n} \log n)$.

[1] With small probability, the client sends a set that contains i, in which case the refresh operation fails. Checklist [40] proposes a way to avoid this.

4.2 Intuition: making CK's preprocessing incremental

Consider a client that has already performed the offline phase and holds $\sqrt{n}\log n$ parities and secret keys. Suppose that m items are then added to the database, so that it consists of $n+m$ objects with indices in $[n+m]$. To support the updated database the client must obtain new parities and sets.

Strawman solution (not secure). The client could keep the *original* parities and sets. Then for new items, sample a few new sets from $[n+m]$, and acquire parities for those via a process similar to refreshing—proceeding with the online phase as before. This appears reasonable at first glance, but it is not secure. Observe that all of the original sets of indices were sampled uniformly from $[n]$. However, in an incremental OO-PIR scheme, security requires that *every* set be sampled uniformly from all indices in the database (see Definition 2), which is the set $[n+m]$. Otherwise, the distribution of the client's queries will be biased towards sets from $[n]$, and the online server will be able to make inferences about the likelihood of certain queries being for new or old items. We give details in Appendix B.

4.2.1 Introducing insertions to CK

In the previous section, we discuss how creating new sets that include the added indices is problematic since the original sets will not be uniformly sampled from $[n+m]$, and hence leaks information about the client's queries. We therefore ask whether instead of creating new sets and acquiring parities for those new sets, the client can modify the existing sets and parities it already has. That is, can we ensure that after the update process is complete, *all* the sets that the client holds are uniformly random subsets of $[n+m]$.

Proposed protocol (simplified). Consider the simple version of CK where the client stores, as a result of the offline phase, the actual random sets of indices rather than their compressed representation (which relies on secret keys to generate each set) in addition to the hints. Suppose that at a later point in time, the database operator appends a single item to the existing n-item database D. The new item will have index $n+1$ (the database is indexed from 1). Observe that a randomly sampled subset of $[n+1]$ of size $\sqrt{n}$ should contain index $n+1$ with probability $\frac{\sqrt{n}}{n+1}$. A straightforward approach to accomplish this is for the client to add index $n+1$ to a given set S_j probabilistically: with probability $\frac{\sqrt{n}}{n+1}$, the client replaces a random index in S_j with $n+1$; with the remaining probability, the client does nothing with set S_j. This guarantees that S_j is a random subset of $[n+1]$. Note that $\frac{\sqrt{n}}{n+1}$ is small, so with high probability the index will not actually be added to S_j. However, recall that in CK the client has $\sqrt{n}\log n$ sets; the client performs the above probabilistic addition for each one of them. This ensures that with high probability the index $n+1$ is added to at least one of the sets.

The above achieves our desired incremental property: when a new object is appended, the client and the offline server do not need to preprocess the database from scratch. Instead, the client—for each set for which there is a replacement (e.g., $n+1$ replaces some existing index e in the set)—asks the offline server for the parity difference ($D[e] \oplus D[n+1]$). The client then XORs $D[e] \oplus D[n+1]$ with the original parity of the set, which cancels out the contribution of $D[e]$ and adds $D[n+1]$ to the set. Note that we keep the size of each set at $\sqrt{n}$ to preserve the structure and simplicity of the CK scheme and its security proof. In addition, we show that this has negligible effect on query correctness (§6). Nevertheless, one could allow sets to diverge and have different sizes, but the original preprocessing would need to randomize the sizes of each set to begin with. We show an example of this in Appendix E where we discuss the SACM OO-PIR scheme [52].

We can generalize this scheme to a batch of appends as follows. To add m items in a batch, the client samples a number w from the hypergeometric distribution[2] $\mathcal{HG}(n+m, m, \sqrt{n})$ for each set S_j, and replaces w random indices in S_j with w random indices in $\{n+1, \ldots, n+m\}$. The client then tells the offline server about the changes to any set for which $w > 0$, so that the offline server can compute the parity difference and allow the client to update its hints. In Appendix A.1 we prove that this procedure ensures that the resulting $\sqrt{n}\log n$ sets, each of size $\sqrt{n}$, are uniform random subsets of $[n+m]$ and the client's hints are valid.

The protocol described here is too costly and does not meet OO-PIR's non-triviality since the client must explicitly store the indices of each of the sets. We address this in Section 5.

4.2.2 Supporting in-place edits

In-place edits occur when the operator of the database changes part of an object or replaces the object entirely. Such edits are easier to handle than insertions because the size of the database does not change, and therefore the existing sets that a client has remain valid. The client only needs to update the parities of all of the sets that contain the index of the object that has changed. Specifically, suppose that the database operator edits $D[e]$ to $D'[e]$ (this change must happen in both the offline and online servers). Then it is enough for the offline server to send e and $\Delta = D[e] \oplus D'[e]$ to the client. Given this information, the client can determine all of the sets that contain e and update their corresponding parity as $p \leftarrow p \oplus \Delta$.

4.2.3 Supporting deletions

There are many situations where a database operator may wish to delete items from the database so that clients can no longer fetch them. For instance, the database may contain an illegal object (e.g., a confidential government document). In such cases, we ask to what extent we can modify CK to support deletion. We consider two cases: deletion against honest clients who follow the protocol, and deletion against

[2]We use $\mathcal{HG}(N, M, K)$ to denote the distribution where the numbers of total items, good items, and quantity drawn are N, M, K.

malicious clients who may deviate arbitrarily. In either case, we assume that the client has not previously fetched the object that the operator wishes to delete.

Deletion against honest clients. Suppose the operator wishes to delete the item at index e. To ensure correctness in later retrievals, the client needs to update the parities of the sets that contain this index e. A trivial way to achieve this is for the offline server to send the item to the client and let the client do an XOR to cancel out the parity of that item for all of the relevant sets. However, this defeats the purpose as the server is deleting an item that it does not want the client to query by sending the item to the client. Instead, we handle deletions by replacing the data with a uniform random object r and using the in-place edit protocol of Section 4.2.2: the server sends $D[e] \oplus r$ to the client, who uses this information to update its hints without obtaining $D[e]$.

Deletion against malicious clients. The above scheme does not work against a malicious client since the client could keep a copy of the original parity p for set S or keep a copy of the update $(r \oplus D[e])$ sent by the server during deletion. In either case, the client can query the online server for the item at index e and either use p to recover $D[e]$ directly, or use the new parity to recover r and then compute $D[e] = r \oplus (r \oplus D[e])$.

Indeed, secure deletion against malicious clients is impossible in CK: when the client obtains the parities for its sets it has in effect an encoding of *all* of the items in the database and there is no way for the servers to remove the contribution of the deleted item. To see why, consider the following attack. The client holds a set S that includes the index e and other indices. It uses the online server as an oracle to obtain each and every item at indices in S except for e. Then, using the original parity of set S it computes the element at index e. This attack requires fetching only $\sqrt{n} - 1$ elements.

Our approach for honest clients does work with malicious clients that join the system *after* the element is deleted since they will not have a copy of the parity for the deleted object.

Reusing deleted slots. Since the server replaces the deleted item with a random mask, it is possible to utilize this empty slot when a new item is added by performing an in-place edit of the random mask with the new element. In some applications the server may even wish to batch a few deletions and insertions and replace them with in-place edits.

5 Incremental pseudorandom sets

In the previous section we discussed a protocol that extends CK to support insertions, deletions, and edits. However, it requires the client to explicitly store the indices for each set, which defeats non-triviality (§3.1) since the client storage would be $O(n \log^2 n)$: $\sqrt{n} \log n$ sets, each consisting of $\sqrt{n}$ indices in $[n+m]$. We therefore ask whether one can support an updatable compressed representation of sets.

To answer this question, recall that CK gets storage and communication efficiency with a *pseudorandom set* (PRS), which we define next. We denote the set size as $s(n)$, which is a function of n, and write $s(n)$ as s for simplicity.

Definition 3 (Pseudorandom set (PRS) [23])**.** A pseudorandom set with size s consists of key space $\mathcal{K}$, and algorithms:

- $\mathsf{Gen}(1^\lambda, n) \to k$: a randomized algorithm that takes as inputs a security parameter λ and the size of the set's range n, and outputs a secret key $k \in \mathcal{K}$.
- $\mathsf{Eval}(k) \to S$, a deterministic algorithm that takes as input a set key $k \in \mathcal{K}$ and outputs a set S.

A formal correctness and security definition for PRS appears in CK [23], but briefly, correctness means: given k, $\mathsf{Eval}(k)$ will output a set $S \subseteq [n]$ with size s; and security means: a PPT adversary cannot distinguish between S and a random size-s subset of $[n]$.

CK gives a simple PRS construction from a *pseudorandom function* (PRF) $f : [n] \to [n]$. Gen generates a key k for the PRF, and Eval produces the elements in the set by the evaluating the PRF at s points, i.e., $S = \{f(k,1), \ldots, f(k,s)\}$.[3] CK shows that S satisfies correctness and security [23, Appendix B.1]. As a result, with a PRS the client stores $\sqrt{n} \log n$ keys, which is a major reduction.

CK additionally requires the PRS to be *puncturable*, which they obtain with a *puncturable pseudorandom function* [11, 14, 36, 39]. This helps CK achieve logarithmic communication during the online phase (§4.1).

For clarity, what follows uses the notation $Set^{[n]}(k, x, y)$ to mean the set that results from evaluating a PRF with range $[n]$ using key k at consecutive points $x, \ldots, y$ for some $x \leq y$. When $x = 1$, we use the shorter notation $Set^{[n]}(k, y)$.

5.1 Incremental PRS

The above approach produces indices on the original database range, but we need to produce indices on an extended range. We thus propose a definition for an incremental PRS. The main change is the inclusion of auxiliary information aux that changes how indices are derived from the set's secret key.

Definition 4. An incremental PRS with set size s consists of the following three algorithms:

- $\mathsf{Gen}(1^\lambda, n) \to (k, aux)$: a randomized algorithm that takes a security parameter λ and a range size n, and outputs a secret key $k \in \mathcal{K}$, and auxiliary information aux.
- $\mathsf{Add}(aux, m) \to aux'$, a randomized algorithm that takes in auxiliary information aux and the size of the added range, $m = o(n)$ (which is equal to the number of additions), and outputs new auxiliary information aux'.
- $\mathsf{Eval}(k, aux) \to S$: a deterministic algorithm that takes as input a set key $k \in \mathcal{K}$ and (potentially updated) auxiliary information aux and outputs a set S.

[3] Note that with small probability a client might generate a key k for which there are collisions within the evaluations (e.g., $f(k,1) = f(k,2)$); in such cases, the client samples a different key.

There are two notes. First, while the range of the set changes, the set size s is kept the same (§4.2.1). Second, the range size from which Eval chooses the elements is encoded in aux. The correctness, security, and non-triviality definitions are:

Correctness. $\mathsf{Eval}(k, aux)$ outputs a set $S \subseteq [n]$ of size s, for a uniformly sampled k, and Add modifies aux correctly: given $(k, aux) \leftarrow \mathsf{Gen}(1^\lambda, n)$ and $aux' \leftarrow \mathsf{Add}(aux, m)$, the output of $\mathsf{Eval}(k, aux')$ is a set $S' \subseteq [n+m]$ of size s.

Security. The set S should be a pseudorandom subset of $[n]$, and the set S' should be a pseudorandom subset of $[n+m]$.

Non-triviality. We denote the *symmetric difference* of two sets S_1 and S_2 as $S_1 \ominus S_2 = (S_1 \cup S_2) \setminus (S_1 \cap S_2)$. We define a function $g(n)$ to be the expected number of elements in $U \ominus U'$ where U is a randomly sampled size-s set from $[n]$ and U' is a randomly sampled size-s set from $[n+m]$, and $m = o(n)$. That is, $E[|U \ominus U'|] = g(n)$. Let S and S' be as defined above. We say that an incremental PRS scheme is *non-trivial* if $E[|S \ominus S'|] = o(g(n))$.

Note that the correctness and security property can be inductively defined for multiple additions. The non-triviality definition means that the intersection of the new set S' with the old set S should be in expectation large. Without this requirement, one could trivially satisfy correctness and security by sampling a new set $S' \subseteq [n+m]$ from scratch.

5.2 High-level idea of incremental PRS construction

Suppose for a moment that we do not care about compression (i.e., we do not require that the PRS be concisely captured by a secret key). Then, one could just represent the PRS explicitly as the subset $S \subseteq [n]$. Our goal is then to devise a procedure to extend the range of the PRS; in other words, to obtain $S' \subseteq [n+m]$ by modifying, in expectation, only a handful of elements in S. We can achieve this by recalling the probabilistic procedure from Section 4.2.1: (1) sample w from a hypergeometric distribution $\mathcal{HG}(n+m,\ m,\ s)$; and (2) remove w random elements from S and add w random elements in $[n+1, n+m]$ to S, thereby creating S'. This strawman constructions guarantees all PRS properties (Appendix A.1).

We now discuss how to achieve the compressed representation. Given a set key k for a (non-incremental) PRS with range $[n]$, one solution to modify the range of the PRS while preserving the usefulness of the key k is to perform the above probabilistic replacement procedure and explicitly store the added elements from $[n+1, n+m]$ and the removed elements from $[n]$ as aux. In this way, Eval could generate the initial set S using k, and then use the explicit elements from aux to turn S into S'. This may work fine when m is tiny (since each element can be represented using $\log(n+m)$ bits), but for larger m, the storage for the elements being added/removed could be much larger than a PRF key.

Our construction improves on the above by keeping side information that is smaller than storing elements in the clear. Observe that since the w elements in $[n+1, n+m]$ are randomly chosen, we can actually use a PRF to generate them—similarly to how CK uses a PRF to generate the original subset S. Our first attempt is to use the *same key* that CK uses to generate S with an *additional* PRF that has a different range, which will capture the added elements. That is, we build the updated set S as $S^1 \cup S^2$ using a PRF key k, where:

$$S^1 := Set^{[n]}(k, s-w),\ S^2 := Set^{[m]}(k, w) + n.$$

In this way, the $s-w$ elements in the original set have not changed because the key is the same, and using the same key to evaluate w points in the new range produces w pseudorandom elements to be added. In other words, the first set S^1 removes w random elements from the original set S, and the second set S^2 adds w random elements from $[n+1, n+m]$ to S—which is what we want. Intuitively, this construction satisfies PRS security: S is a uniformly sampled subset of $[n+m]$ (each elements in $[n+m]$ has probability $\frac{s}{n+m}$ of being included in S). Appendix A.3 gives the full analysis.

Complications. There are two complications with the above approach. The first complication is that using a PRF to instantiate the PRS can result in collisions. This means that there is no guarantee that exactly w elements in the original set are removed. For this reason, we use a PRP instead.

The second complication is that if we use the *same* key to pick two pseudorandom permutations (or functions) from two pseudorandom permutation (or function) families with different ranges, the two could be related.[4] This defeats the security property in Definition 4. To avoid this, we use a different key for each of the PRPs. Specifically, for each set S, the original set key k becomes a master key used to derive keys k_1 and k_2 by a key derivation function KDF. The set S is then $S = Set^{[n]}(k_1, s-w) \cup Set^{[m]}(k_2, w) + n$.

Extending a set multiple times. The above approach supports multiple range extensions. Suppose a client holds the following information after increasing the range from $[n]$ to $[n+m]$: a PRS key k, auxiliary information $t_1 = s-w$ and $t_2 = w$. Together, they represent $S = S^1 \cup S^2$, where $S^1 = Set^{[n]}(k_1, t_1)$ and $S^2 = Set^{[m]}(k_2, t_2) + n$, and k_1, k_2 are derived from k. Now, say we want to incrementally modify S to be a random subset of $[n+2m]$ (without loss of generality). As before, we sample t_3 from $\mathcal{HG}(n+2m, m, s)$ and succinctly represent the added elements from $[n+m+1, n+2m]$ as $Set^{[m]}(k_3, t_3) + n + m$, where k_3 can be derived from k.

Dealing with the removed elements requires a little more work. We randomly select t_3 elements in $[s]$, and denote the number of elements in $[1, t_1]$ as $\bar{t}_1$, and that in $[t_1+1, s]$ as $\bar{t}_2$ (remember that $t_1 + t_2 = s$). The client updates the auxiliary information: $t_\ell \leftarrow t_\ell - \bar{t}_\ell$ for $\ell = 1, 2$. The resulting set $Set^{[n]}(k_1, t_1) \cup Set^{[m]}(k_2, t_2)$ is then a set with t_3 random indices removed.

[4] Imagine if one PRF is AES and the other PRF (which needs a smaller range) simply uses half of the output bits of AES. While individually each PRF is secure, when used together with the same key the PRFs are related.

procedure $\mathsf{Gen}(1^\lambda, n) \to (k, aux)$
  Sample $k \leftarrow \mathcal{K}$, and set $aux \leftarrow [(n, s)]$.
  Output (k, aux).
procedure $\mathsf{Add}(aux, m) \to aux'$
  Parse aux as $[(r_\ell, t_\ell)]_{\ell \in [L]}$.
  Compute $r \leftarrow \sum_{\ell \in [L]} r_\ell$.
  Sample $w \leftarrow \mathcal{HG}(r + m, m, s)$.
  Sample w points uniformly in $[s]$.
  Set $t_0 \leftarrow 0$.
  for $\ell \in [L]$ **do**
    Set $\bar{t}_\ell \leftarrow$ number of points in $(\sum_{z=0}^{\ell-1} t_z, \sum_{z=0}^{\ell} t_z]$.
    Update $t_\ell \leftarrow t_\ell - \bar{t}_\ell$
  Set $r_{L+1} = m, t_{L+1} = w$.
  Output $aux' \leftarrow [(r_\ell, t_\ell)]_{\ell \in [L+1]}$.
procedure $\mathsf{Eval}(k, aux) \to S$
  Parse aux as $[(r_\ell, t_\ell)]_{\ell \in [L]}$.
  for $\ell \in [L]$ **do**
    Derive keys $k_\ell \leftarrow \mathsf{KDF}(k, \ell)$
    Compute set $S^\ell \leftarrow Set^{[r_\ell]}(k_\ell, t_\ell) + \sum_{z<\ell} r_z$
  Output $S \leftarrow \bigcup_{\ell \in [L]} S^\ell$

FIGURE 1—Construction of incremental PRS Ψ.

Therefore, after extending the range twice (each time by m), S can be represented with k, t_1, t_2, t_3, and written as:

$$Set^{[n]}(k_1, t_1) \cup Set^{[m]}(k_2, t_2)+n \cup Set^{[m]}(k_3, t_3)+m+n,$$

where k_1, k_2, k_3 are derived from k, and $t_1 + t_2 + t_3 = s$.

5.3 Incremental PRS construction

The previous section gives the definition of incremental PRS that we seek, and the high level overview of our construction. We formalize that procedure in Figure 1.

A set is represented using a set key k and some short auxiliary information aux. The latter is a list of tuples $[(r_\ell, t_\ell)]_{\ell \in [L]}$, where L is the number of subranges of a set (the number of times the PRS's range has been extended from the initial range is hence $L - 1$). Each tuple (r_ℓ, t_ℓ) denotes the number of elements, t_ℓ, chosen from the subrange r_ℓ. For instance, the aux value of the example in Section 5.2 (which extends the range twice) is $[(n, t_1), (m, t_2), (m, t_3)]$, and the full range is $[n + 2m]$.

Theorem 1. Assuming a set of secure PRPs, PRS construction Ψ satisfies PRS correctness, security, and non-triviality.

Appendix A.3 gives a proof of this theorem.

Can we puncture our incremental PRS? Unfortunately our construction does not preserve the puncturable property since we use a PRP instead of a PRF to instantiate our incremental PRS and there does not exist puncturable PRP constructions [10]. For our incremental CK, this results in less succinct online communication ($O(\sqrt{n})$), but in practice it works well (§8). Designing a puncturable incremental PRS is an interesting open question.

procedure $\mathsf{Prep}(D) \to h$
  for $j \in [J]$ **do**
    Client samples $(k_j, aux_j) \leftarrow \Psi.\mathsf{Gen}(1^\lambda, n)$
  for $j \in [J]$ **do**
    Server computes $S_j \leftarrow \Psi.\mathsf{Eval}(k_j, aux_j)$
    and $p_j \leftarrow \bigoplus_{e \in S_j} D[e]$
  Output $h \leftarrow \{(k_j, aux_j, p_j)\}_{j \in [J]}$

FIGURE 2—Offline preprocessing between offline server and client. The client stores h and uses it during online queries.

6 Adding incremental PRS into OO-PIR

In order to incorporate our PRS into CK, we adapt two helper functions from CK [23].

Membership testing. Let $\mathsf{Member}(i, (k, aux))$ be an algorithm that uses a single PRP call to determine whether index i is in the set specified by (k, aux). Recall that aux is a list $[(r_\ell, t_\ell)]_{\ell \in [L]}$. We first identify which subrange i is in, say r_ℓ. Then we check if the corresponding t_ℓ is 0. If so, then the set does not contain any elements in subrange r_ℓ, and Member outputs 0. Otherwise, we compute the inverse permutation keyed by k_ℓ (derived from k) on i. If the result is in $[t_\ell]$, then Member outputs 1, otherwise it outputs 0.

Shifting PRS. During a set refresh (§4.1), the client needs to generate a set under the restriction that a certain index i is in the set. If the client simply generates a random PRP key k, which defines a set $S = Set^{[n']}(k, s)$ where n' is the updated database size, the probability that $i \in S$ is very low. We instead randomly choose an element $x \in S$, define a shift $sh = x - i \mod n'$, and let $S' = S + sh \mod n'$. In this way, k together with a shift sh, succinctly represents a set S' that contains i with all other $s - 1$ indices being random. For simplicity, in the next section we omit sh and just use $(k, aux) \leftarrow \mathsf{GenWith}(i, n')$ to denote the above procedure; we assume Eval takes the shift into account when computing the set. We give the details in Appendix A.4.

6.1 Updating hints with incremental PRS

We now discuss how to make the CK protocol incremental with the use of incremental PRS Ψ. We assume the client holds J sets; to ensure OO-PIR's non-triviality, J must be sublinear in n so we let $J = (n/s) \log n$ as in Section 4.1.

The offline preprocessing (Figure 2) is the same as in Section 4.1, except that the hints now include auxiliary information for each set. After preprocessing, if the database D changes due to a batch of operations op, the client and the offline server run $\mathsf{IncPrep}(D, op, h)$ to get a new hint h'. We describe each step in IncPrep below.

First, the offline server runs DBUpd to group the list of changes specified by op (e.g., add index 101, edit index 2, del index 7) and produces a short summary δ, which we define in Figure 3. The server sends δ to the client.

Upon receiving δ, the client runs HintReq to generate u_q (Figure 4). u_q consists of succinct representations for all the

procedure DBUpd$(D, op) \to (D', \delta)$
  Use op to operate on D, produce D'.
  Group operations in op by additions, edits, and deletions:
  if op contains m_{add} additions **then**
    Set $\delta_{\text{add}} \leftarrow$ ("add", m_{add})
  Set D^* be the database after the m_{add} additions.
  if op contains edits at $x_1, \ldots, x_{m_{\text{edit}}}$ **then**
    Set $\delta_{\text{edit}} \leftarrow$ ("edit", $[x_1, \ldots, x_{m_{\text{edit}}}]$)
    Set $(X_1, \ldots, X_{m_{\text{edit}}}) \leftarrow (D^*[x_1], \ldots, D^*[x_{m_{\text{edit}}}])$
    Set $(X'_1, \ldots, X'_{m_{\text{edit}}}) \leftarrow (D'[x_1], \ldots, D'[x_{m_{\text{edit}}}])$
  if op contains deletions at $y_1, \ldots, y_{m_{\text{del}}}$ **then**
    Set $\delta_{\text{del}} \leftarrow$ ("del", $[y_1, \ldots, y_{m_{\text{del}}}]$)
    Set $(Y_1, \ldots, Y_{m_{\text{del}}}) \leftarrow (D^*[y_1], \ldots, D^*[y_{m_{\text{del}}}])$
    Set $(Y'_1, \ldots, Y'_{m_{\text{del}}}) \leftarrow (r_1, \ldots, r_{m_{\text{del}}})$,
      where $r_z \xleftarrow{R} \{0,1\}^b$ for all $z \in [m_{\text{del}}]$
  Set $\delta \leftarrow (\delta_{\text{add}}, \delta_{\text{edit}}, \delta_{\text{del}})$, output (D', δ)

FIGURE 3—Algorithm for producing summary δ. The offline server keeps the parity difference for edits (the X-array) and for deletions (the Y-array) generated in DBUpd, which are used for HintRes.

procedure HintReq$(h, \delta) \to u_q$
  Parse δ as $(\delta_{\text{add}}, \delta_{\text{edit}}, \delta_{\text{del}})$
  Initialize an empty set Q
  Initialize each entry in $(aux'_1, \ldots, aux'_J)$ as $\perp$
  Initialize each entry in $(aux^*_1, \ldots, aux^*_J)$ as $\perp$
  for $j \in [J]$ **do**
    if $\delta_{\text{add}} =$ ("add", m_{add}) **then**
      Set $aux'_j \leftarrow \Psi$.Add(aux_j, m_{add}) // see Fig. 1, line 4
      Parse aux'_j as $[(r_\ell, t_\ell)]_{\ell \in [L]}$
      if $t_L \neq 0$ **then** add j into set Q, set $aux^*_j \leftarrow aux'_j$
    if $\delta_{\text{edit}} =$ ("edit", $[x_1, \ldots, x_{m_{\text{edit}}}]$) **then**
      if $\exists z \in [m_{\text{edit}}]$ s.t. Member$(x_z, (k_j, aux'_j)) = 1$ **then**
        Add j into Q
    if $\delta_{\text{del}} =$ ("del", $[y_1, \ldots, y_{m_{\text{del}}}]$) **then**
      if $\exists z \in [m_{\text{del}}]$ s.t. Member$(y_z, (k_j, aux'_j)) = 1$ **then**
        Add j into Q
  Output $u_q \leftarrow \{(k_j, aux_j, aux^*_j)\}_{j \in Q}$

FIGURE 4—Client algorithm for generating update request. It keeps a set Q for all changed sets. aux' indicates the updated auxiliary information, and aux^* encodes the information needed for the server to do incremental updates ($\perp$ or aux').

sets whose elements changed. In some cases, the number of changes will be large enough that with high probability all of the client's sets will be impacted. In our tech report [46, Appendix C], we describe optimized algorithms for this case.

The offline server, upon receiving u_q, runs HintRes (Figure 5) to generate the parity of data blocks that the client needs to update its hints. For the three types of changes, the server's computation is different. For additions, the server uses the helper function EvalDiff (Figure 5) to generate the indices that are added into or removed from a set. This function takes in a set key k, the original aux, and the updated aux' for one batch of additions (aux and aux' should have the same subranges). This helper function efficiently computes $S \ominus S'$, where $S \leftarrow \text{Eval}(k, aux)$ and $S' \leftarrow \text{Eval}(k, aux')$, without materializing either set. The server then computes the parity of data blocks indexed by elements in $S \ominus S'$.

procedure HintRes$(D', u_q) \to u_r$
  Parse u_q as $\{(k_j, aux_j, aux^*_j)\}_{j \in Q}$.
  Initialize a tuple $(p'_1, \ldots, p'_{|Q|})$ and set them all to 0^b.
  Initialize each entry in $(aux'_1, \ldots, aux'_{|Q|})$ as $\perp$.
  for $j \in Q$ **do** // process additions first
    if $aux^*_j \neq \perp$ **then**
      Set $aux'_j \leftarrow aux^*_j$
      Compute $S \leftarrow$ EvalDiff(k_j, aux_j, aux'_j)
      Compute $p'_j \leftarrow \bigoplus_{e \in S} D'[e]$
    else Set $aux'_j \leftarrow aux_j$
  for $j \in Q$ **do** // process edits and deletions
    Update $p'_j \leftarrow p'_j \bigoplus (X_z \oplus X'_z), \forall z \in [m_{\text{edit}}]$
      such that Member$(x_z, (k_j, aux'_j)) = 1$
    Update $p'_j \leftarrow p'_j \bigoplus (Y_z \oplus Y'_z), \forall z \in [m_{\text{del}}]$
      such that Member$(y_z, (k_j, aux'_j)) = 1$
  Output $u_r \leftarrow (p'_1, \ldots, p'_{|Q|})$
// compute symmetric difference between two sets
procedure EvalDiff$(k, aux, aux') \to S$
  Parse aux as $[(r_\ell, t_\ell)]_{\ell \in [L]}$
  Parse aux' as $[(r_\ell, t'_\ell)]_{\ell \in [L+1]}$
  for $\ell \in [L]$ **do**
    Derive keys $k_\ell \leftarrow \text{KDF}(k, j)$
    Compute $S^\ell \leftarrow Set^{[r_\ell]}(k_\ell, t'_\ell + 1, t_\ell), \forall t'_\ell < t_\ell$
  Compute $S^* \leftarrow \cup_{\ell \in [L]} S^\ell$
  Output $S \leftarrow S^* \cup Set^{[r_{L+1}]}(k_{L+1}, t'_{L+1})$

FIGURE 5—Offline server responds to an update request by computing the parity difference for each set that has changed.

procedure HintUpd$(h, u_r) \to h'$
  Parse h as $\{(k_j, aux_j, p_j)\}_{j \in [J]}$
  for $j \in [J]$ **do** Update $aux_j \leftarrow aux'_j$ // see Fig. 4, Line 8
  Parse u_r as $(p'_1, \ldots, p'_{|Q|})$
  for $j \in Q$ **do**
    Update $p_j \leftarrow p_j \oplus p'_j$.
  Output $h' \leftarrow \{(k_j, aux_j, p_j)\}_{j \in [J]}$

FIGURE 6—Client algorithm for updating local hints.

For deletions and edits, the offline server uses the client's set keys to find in which of the client's sets the changed indices fall, and then computes the parity difference. Finally, the server sends the hint response to the client, who then uses it in HintUpd (Figure 6) to update its hints.

6.2 Online query and refresh

The client's online query for some index i works as follows. First, the client identifies which set contains i using Ψ's Member function (§6). Then, the client uses Eval to derive the set's indices, and then removes i from the set with probability $(s-1)/n'$ (or another index with the remaining probability), where s is the set size and n' is the newest database size. This is in effect a trivial way of doing puncturing. Finally, the client sends the remaining indices in the clear to the online server as the query. We give the details in Figures 7 and 8.

Although the above approach leads to online communication of $O(\sqrt{n})$ bits (we have $n \leq n'$), which is asymptotically worse than the polylog communication in CK, we show in our

Client keeps $\mathsf{state} = (n', j^*)$, where n' is the current database size, and j^* indicates which set is currently used for query.
procedure $\mathsf{Query}(h, i) \to q_{\text{online}}$
  Parse $\mathsf{state} = (n', j^*)$
  for $j \in [J]$ **do** // find a set containing i
    Compute $\alpha \leftarrow \Psi.\mathsf{Member}(i, (k_j, aux_j))$
    if $\alpha = 1$ **then** set $j^* \leftarrow j$ and **break**
  if for all $j \in [J]$, $\alpha = 0$ **then** output fail
  // generate query
  Compute $S_{\text{online}} \leftarrow \Psi.\mathsf{Eval}(k_{j^*}, aux_{j^*})$
  Sample $\beta_{\text{online}} \leftarrow \mathsf{Bernoulli}(\frac{s-1}{n'})$
  if $\beta_{\text{online}} = 0$ **then** $S'_{\text{online}} \leftarrow S_{\text{online}} \backslash \{i\}$
  if $\beta_{\text{online}} = 1$ **then**
    $y_{\text{online}} \xleftarrow{R} S_{\text{online}} \backslash \{i\}$, $S'_{\text{online}} \leftarrow S_{\text{online}} \backslash \{y_{\text{online}}\}$
  Output $q_{\text{online}} \leftarrow S'_{\text{online}}$
procedure $\mathsf{Refresh}(h, i) \to q_{\text{offline}}$
  Parse $\mathsf{state} = (n', j^*)$
  Set $(k_{\text{new}}, aux_{\text{new}}) \leftarrow \Psi.\mathsf{GenWith}(i, n')$
  Compute $S_{\text{offline}} \leftarrow \Psi.\mathsf{Eval}(k_{\text{new}}, aux_{\text{new}})$
  Compute $\beta_{\text{offline}} \leftarrow \mathsf{Bernoulli}(\frac{s-1}{n'})$
  if $\beta_{\text{offline}} = 0$ **then** $S'_{\text{offline}} \leftarrow S_{\text{offline}} \backslash \{i\}$
  if $\beta_{\text{offline}} = 1$ **then**
    $y_{\text{offline}} \xleftarrow{R} S_{\text{offline}} \backslash \{i\}$, $S'_{\text{offline}} \leftarrow S_{\text{offline}} \backslash \{y_{\text{offline}}\}$
  Update $k_{j^*} \leftarrow k_{\text{new}}$, $aux_{j^*} \leftarrow aux_{\text{new}}$.
  Output $q_{\text{offline}} \leftarrow S'_{\text{offline}}$
procedure $\mathsf{QueryRecov}(r_{\text{online}}, h) \to d_i$
  Parse $\mathsf{state} = (n', j^*)$
  Parse h as $\{(k_j, aux_j, p_j)\}_{j \in [J]}$
  Output $d_i \leftarrow p_{j^*} \oplus r_{\text{online}}$
procedure $\mathsf{RefreshRecov}(r_{\text{offline}}, h)$
  Parse $\mathsf{state} = (n', j^*)$
  Parse h as $\{(k_j, aux_j, p_j)\}_{j \in [J]}$
  Update $p_{j^*} \leftarrow r_{\text{offline}} \oplus d_i$

FIGURE 7—Client algorithms for online query. Client fetches an item using Query and QueryRecov. To refresh a used set, the client samples a new set (Refresh) and gets its parity (RefreshRecov).

evaluation that the communication cost is reasonable given that element sizes in real databases are larger than one bit (§8).

To refresh a set, the client generates a new key, and updates the hint. After the refresh is complete, the refreshed set's auxiliary information will simply consist of one tuple, (n', s). Notice that refreshing a set has the nice side effect of reducing the size of the auxiliary information since we no longer need to maintain all the subranges of the incremental PRS (since a refreshed set is basically a set generated from scratch). As a result, when the number of queries following a database update is such that the client has used every set at least once, the client's auxiliary information will be in a state that is comparable to preprocessing the new database from scratch.

Failure probability. The proposed online phase does not meet our correctness definition (§3.2) because the client fails to puncture the set at index i with probability $O(1/\sqrt{n})$ where n is the original database size, rather than $\mathrm{negl}(\lambda)$. However, we can employ the refinement to CK given in Checklist [40] to reduce this error to $\mathrm{negl}(\lambda)$. For simplicity, we give the former in Figure 7 and discuss the latter in Appendix A.4.2.

Besides the aforementioned puncturing failure, another

procedure $\mathsf{Resp}(q, D) \to r$
  Parse q as a set S
  Output $r \leftarrow \bigoplus_{e \in S} D[e]$

FIGURE 8—Server algorithm during the online phase.

source of failure is when an index is not in any of the sets for which the client has hints. As we mention in Section 4, we chose to allow the database to grow while keeping the size of the sets held by the client the same (Appendix E discusses an incremental OO-PIR protocol which supports sets of different sizes). As a result, the probability that an index is not in any of the sets will necessarily increase. We analyze this both in theory and concretely, and argue that this probability is quite small.

Suppose each set has size s, is a random subset of $[n]$, and the client holds n/s such sets. The probability of failing to find a desired index i in any of the sets is $(1 - s/n)^{n/s}$, which is close to $1/e$ when $s = n^{1/2}$ and n is large. Considering that we (and CK) use a factor of $\log n$ more sets, the probability of failing to find i in any of the $\sqrt{n} \log n$ sets is roughly $1/n$.

When the size of the database grows to $n + m$ where $m = o(n)$, the probability of failing to find i in any of n/s sets is $(1 - s/(n + m))^{n/s}$, which is asymptotically $(1/e)^{1-o(1)}$. With a factor of $\log n$ more sets, this probability is $O(1/n^{1-o(1)})$. Concretely, suppose the database is of size 2^{20}, and the client holds 2^{14} sets, where each set has size 2^{10}. The failure probability in this case is 10^{-7}. If the server adds 2^{10} more items, the failure probability is still in the magnitude of 10^{-7}. When the number of additions reaches 2^{18}, the failure probability increases to the magnitude of 10^{-6}.

Theorem 2. Assuming an incremental PRS Ψ that satisfies correctness, security, and non-triviality (Definition 4), then the incremental OO-PIR defined in Figures 3–8 (with the online query improvement in Appendix A.4) satisfies correctness, security, and non-triviality (Definition 2).

We give the proof in Appendix A. In particular for non-triviality, we show in Appendix A.4.2 that the computational cost to the offline server for $\mathsf{IncPrep}$ with m operations (specified by op) on a size-n database is in expectation $O(bm \log n)$, with data item size b, set size s, and $J = (n/s) \log n$. In contrast, preprocessing from scratch for the entire updated database requires $O(b(m + n) \log n)$ server computation.

Nevertheless, communication costs for hint updates are similar to preprocessing from scratch since that operation is already sublinear in the database size in order to achieve non-triviality (Definition 1).

6.3 Other PIR Schemes

Due to space limitations, we only discuss our incremental CK protocol in the main paper. However, Appendix E discusses how to make the SACM OO-PIR scheme [52] incremental with similar high-level ideas as those presented here, but with vastly different concrete mechanisms.

7 PIR-Tor with incremental OO-PIR

While there are many applications that could benefit from our incremental OO-PIR constructions, we pick PIR-Tor [48] as a representative example. PIR-Tor is a good fit since the underlying database mutates slightly over time, and users query this database periodically. We review PIR-Tor below.

Tor [26] provides a simplified view of the network by maintaining *directory servers* that track the Tor relays that are currently available. To build an onion routing circuit, users download a description of the entire network from the directory servers every 10–15 minutes. PIR-Tor [48] proposes the use of PIR in the context of these directory servers so that clients can retrieve the description of the desired relays without revealing to the directory servers which relays were being requested. This approach has the benefit of lowering communication cost and improving scalability since communication costs were often the bottleneck for the directory servers. An obvious drawback is that the server's computation becomes *linear* in the number of Tor relays currently available, which ultimately places a hard limit on the number of concurrent queries that a directory server can process. Hence, using a PIR scheme that achieves sublinear computation, such as our incremental CK construction (§4), can improve scalability and gracefully handle updates to the directory server database.

Assigning roles to directory servers. Since Tor has multiple directory servers and a large number of clients, a natural question is how to assign the roles of offline and online servers to directory servers and how the clients should choose which servers to contact in a way that load balances the work (since offline servers perform the expensive preprocessing step). We make a simple observation: a server can act as an offline server for one client and an online server for another. Based on this, the Tor trusted authority propagates up-to-date relay descriptions to all the directory servers. When a client registers, the client can then decide on two random directory servers to use as an offline and online server, respectively.

Moreover, each online server has no idea of which offline server the client chooses. In order to figure out the exact index that the client queries, the online server needs to collude with the exact directory servers that is communicating with the client. Suppose there are p directory servers and an adversary controls q of them. Then, for each client, the security is compromised with probability roughly $(\frac{q}{p})^2$.

8 Evaluation

Our evaluation aims to answer the following questions:

- What are the concrete computational and communication costs of our incremental OO-PIR constructions?
- What is the throughput and latency of incremental preprocessing compared to preprocessing from scratch?
- What is the cost to the client to maintain the hints?
- What are the benefits of incremental OO-PIR in PIR-Tor?

To answer these questions, we implement and evaluate our incremental CK. We also have a construction for incremental SACM (Appendix E) but find that both the original [52] and our incremental version are not yet useful in practice (for a database with 2^{20} items, the size of the hints in both schemes is as large as the database).

Baselines. We use two state-of-the-art baselines: (1) CK OO-PIR [23], which we implement; and (2) the two-server DPF-based PIR scheme of Gilboa and Ishai [30], implemented by Kales et al. [1, 37]. The latter has no offline phase.

Implementation. Our incremental CK implementation is ~2,000 lines of C++. Implementing our PRP requires care since the range (and domain) can be very small because the database operator could add just a handful of elements; typical PRP constructions are not secure when the range is small. We use AES to implement a PRF for small range, and then apply Patarin's proposal [50] to the PRF to build a secure PRP that has a small power-of-two range. Finally, we use the technique by Black and Rogaway [9] for turning a PRP with a power-of-two range into a PRP with an arbitrary range.

We generate the hypergeometric sampling for hint updates using multiple Bernoulli samplings. For the CK baseline, we use the GGM construction [32] to instantiate the puncturable PRF out of a PRG. When evaluating the puncturable PRF at continuous points $1, 2, \ldots, s$, we apply breadth-first expansion in the PRG tree to reduce costs. In our experiments, we choose the number of sets J such that the failure probability of finding an index in one of the sets is around 10^{-6}.

Evaluation testbed. We run all of our experiments on CloudLab [28] m510 machines (8-core 2 GHz Intel Xeon D-1548 processor and 64 GB RAM) running Ubuntu 20.04. The network latency between machines is 20 ms and the throughput is around 1.1 Gbps. For results, we give the average over ten trials; we find that in all cases standard deviations are less than 10% of the mean.

8.1 Microbenchmarks for incremental PIR

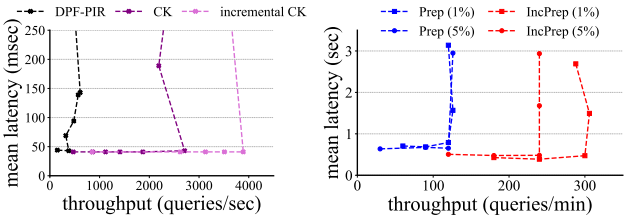
We run a series of microbenchmarks to measure the time required for the initial preprocessing, online query processing, and updating hints. Figure 9 shows the results under three different database sizes, where each data object is 32-bytes.

Online costs. A client's *online query* consists of two parts: a query for an item to the online server and, in parallel, a refresh of the used set to the offline server. Both servers' logic is the same: they perform a number of XORs that is sublinear in the database size (in our case roughly square-root). As a result, the costs are low; for a database with a million records, each server spends less than 0.1 ms. In terms of communication, even though clients send indices in the clear, it is still practical: for one million elements, the combined cost to query an item and refresh the used set is under 16.4 KB.

Database size	2^{16}	2^{18}	2^{20}
Client CPU costs			
Prep (ms)	0.36	0.74	1.28
Query (ms)	3.93	6.48	7.87
Refresh (ms)	1.35	2.07	4.90
IncPrep (sec)	0.06	0.50	3.96
Server CPU costs			
Prep (sec)	3.64	14.52	58.67
Resp (ms)	0.02	0.04	0.06
IncPrep (sec)	0.07	0.25	1.03
Communication costs			
Prep (MB)	0.18	0.37	0.74
Query (KB)	2.04	4.09	8.18
Refresh (KB)	2.04	4.09	8.18
IncPrep (MB)	0.19	0.38	0.76

FIGURE 9—Microbenchmarks for the operations in incremental CK when adding a batch of new elements (1% of the original database).

Update costs. For hint updates, we measure the cost of a batch of operations that increases the database size by 1%. Figure 9 shows that the computational cost of incremental preprocessing (IncPrep) is about two orders of magnitude lower than preprocessing from scratch (Prep). Note that the incremental preprocessing computation is not strictly linear in the number of additions since evaluating a PRP for non-power-of-two ranges is more expensive due to cycle walking [9].



(a) Online phase. (b) Preprocessing and hint updates.

FIGURE 10—Mean latency and server throughput as a function of increasing load (independent variable not depicted) during online phase and incremental preprocessing phase. Each data point represents the latency and the throughput achieved at a given load (low and to the right is better). A vertical spike indicates that the system has reached saturation: throughput is consistent but mean latency goes up since requests must spend time in queues waiting for service.

8.2 Online performance and costs

The previous microbenchmark shows the savings that OO-PIR servers can expect when there are additions to the database (edits and deletes are cheaper). In real applications, we also consider end-to-end metrics: (1) the latency of issuing online queries (fetching the desired item and refreshing the used set) and updating hints; (2) the cost to the servers to update the database; and (3) the client's storage cost over time.

We measure these metrics with parameters inspired by Tor: database items are 2KB (size of a Tor relay descriptor), and the initial database consists of 7,000 items (number of Tor relays [2]). For database updates, we compile and use historical traces of Tor relay updates from February to May 2021, and use 3 days worth of changes for each batch. We also give experiments for a larger deployment of a hypothetical Tor network with 70K relays in Appendix D.

Performance. We measure the latency experienced by a client and throughput achieved by the servers when issuing online queries as we increase the load on the system under our incremental CK (§4) and the baselines. Figure 10a depicts our results. Compared to DPF-PIR, incremental CK improves the throughput achieved by roughly 6×. Since incremental CK performs no cryptography (the indices are sent in the clear), it achieves a higher throughput and lower latency than both baselines which derive the indices from a key.

Online communication. Figure 11a shows the online communication for two baselines (DPF and CK) and incremental CK. We show the combined size of the queries sent to the two servers and the corresponding response. Queries in the DPF baseline are smaller than incremental CK since they send cryptographic keys rather than indices in the clear. [5] The size of each server's reply is optimal for all three schemes and consists of the size of a data element (2KB). This optimal response size is a benefit of our approach over Checklist [40], which supports additions by storing them in different buckets but requires fetching an element from every bucket.

8.3 Offline performance and costs

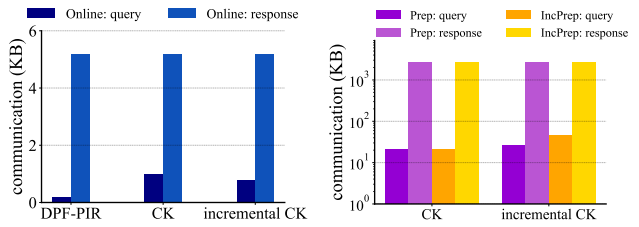
We then measure the latency experienced by the client and the throughput achieved by the server when updating hints as a result of a database change. The baseline lets the client and the offline server re-do the preprocessing from scratch.

Figure 10b depicts the results for preprocessing and hint updates for incremental CK. We find that it achieves 2–4× speedup (depending on batch size) compared to preprocessing from scratch. This results in both lower latency experienced by the client and higher throughput for the offline server. The benefit of our incremental preprocessing is more prominent when there are only a small number of data objects added, or if edits and deletions are performed.

Communication costs. Incremental CK incurs higher offline communication (initial preprocessing and hint updates) than CK because the client needs to also send auxiliary information to the server. This is depicted in Figure 11b.

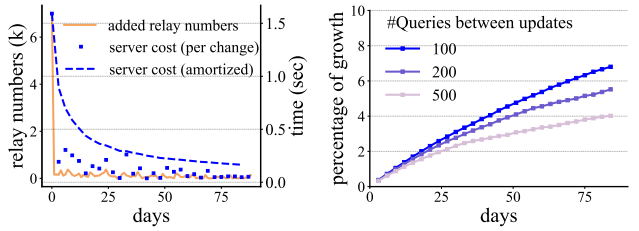
Computation and storage costs. We measure the costs incurred by the offline server and the client during a database update by relying on the traces we collected for Tor relays. Specifically, the offline server sets up a database with 7K relays, and the client and the server preprocess it. Then, the server adds relays over time with both client and the offline server engaging exclusively in incremental preprocessing.

[5] For small databases (e.g., 7K elements) the indices are smaller than the punctured key, but this is not the case for larger databases (Appendix D).



(a) Online phase. (b) Preprocessing and hint updates.

FIGURE 11—Online communication costs for DPF-PIR, CK, and incremental CK, and offline communication costs for CK and incremental CK for our experiment with 7K Tor relays.



(a) Server computation costs. (b) Client local storage changes.

FIGURE 12—Offline server computation costs and client storage over time based on a 3-month Tor relay update trace.

Figure 12a shows the results. The computation for the server is proportional to the number of changes, and is significantly lower than preprocessing the updated database from scratch. The downside for the client is that it must store more data over time. In particular, as shown in Figure 12b, the client storage over a period of 90 days grows based on the client's query and server's update frequency. The reason that the client's query frequency matters is that every time the client uses a set it refreshes the set, thereby eliminating the need to keep auxiliary information for that set (§6.2). As a result, the more queries that a client issues between database updates, the lower its storage overhead. Among all frequencies shown, the percentage of client storage growth is less than 8%. Furthermore, the client can preprocess the database from scratch when the local storage becomes too high.

9 Discussion

This work introduces the idea of incremental preprocessing for offline/online PIR with the hope of supporting real-world applications which often have mutable databases. Our evaluation shows that our changes to the CK OO-PIR scheme are affordable in terms of communication and storage costs and effective at reducing the impact of updates. However, our approach is not a panacea and open questions remain.

If the database items are large, OO-PIR schemes where the hints are stored at the client (e.g., CK and SACM) are a poor fit. Since our incremental preprocessing is not black-box, it remains to be seen how to apply it to schemes where the hints are kept at the servers [8]. Furthermore, this work investigates only two-server PIR schemes; designing efficient preprocesing for single-server PIR remains an open question (existing schemes rely on obfuscation [15, 18, 23]), and ensuring that the preprocessing is incremental is an exciting direction. Finally, it is unclear how to support incremental preprocessing and PIR-by-keywords [21] given that mutations that changes the keywords of existing items or add new keywords would impact the underlying search data structure.

Nevertheless, we are excited by the prospect of continuing to extend the class of applications that can profitably use PIR to build services that safeguard users' privacy.

Acknowledgments

We thank the anonymous reviewers, and our shepherd Cas Cremers, for their feedback, which improved the content and presentation of this paper. This work was funded in part by NSF grant CNS-2045861, DARPA contract HR0011-17-C0047, and a JP Morgan Chase & Co Faculty Award. Any views expressed herein are solely those of the authors listed.

References

[1] C++ dpf-pir library. https://github.com/dkales/dpf-cpp.

[2] Tor metrics. https://metrics.torproject.org/networksize.html, 2021.

[3] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIR: Private information retrieval for everyone. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[4] A. Ali, T. Lepoint, S. Patel, M. Raykova, P. Schoppmann, K. Seth, and K. Yeo. Communication-computation trade-offs in pir. Cryptology ePrint Archive, Report 2019/1483, 2019.

[5] S. Angel, H. Chen, K. Laine, and S. Setty. PIR with compressed queries and amortized query processing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2018.

[6] S. Angel and S. Setty. Unobservable communication over fully untrusted infrastructure. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2016.

[7] A. Beimel, Y. Ishai, E. Kushilevitz, and J.-F. Raymond. Breaking the $O(n^{1/(2k-1)})$ barrier for information-theoretic private information retrieval. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Nov. 2002.

[8] A. Beimel, Y. Ishai, and T. Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2000.

[9] J. Black and P. Rogaway. Ciphers with arbitrary finite domains. In *Cryptographers' track at the RSA conference*, 2002.

[10] D. Boneh, S. Kim, and D. J. Wu. Constrained keys for invertible pseudorandom functions. In *Proceedings of the Theory of Cryptography Conference (TCC)*, 2017.

[11] D. Boneh, B. Waters, K. Sako, and P. Sarkar. Constrained pseudorandom functions and their applications. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2013.

[12] N. Borisov, G. Danezis, and I. Goldberg. DP5: A private presence service. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, June 2015.

[13] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2016.

[14] E. Boyle, S. Goldwasser, and I. Ivan. Functional signatures and pseudorandom functions. In *Proceedings of the International Conference on Practice and Theory in Public Key Cryptography (PKC)*, 2014.

[15] E. Boyle, Y. Ishai, R. Pass, and M. Wootters. Can we access a database both locally and privately? In *Proceedings of the Theory of Cryptography Conference (TCC)*, Nov. 2017.
[16] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 2011.
[17] C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 1999.
[18] R. Canetti, J. Holmgren, and S. Richelson. Towards doubly efficient private information retrieval. In *Proceedings of the Theory of Cryptography Conference (TCC)*, Nov. 2017.
[19] Y.-C. Chang. Single database private information retrieval with logarithmic communication. In *Proceedings of the Australasian Conference on Information Security and Privacy*, July 2004.
[20] R. Cheng, W. Scott, E. Masserova, I. Zhang, V. Goyal, T. Anderson, A. Krishnamurthy, and B. Parno. Talek: Private group messaging with hidden access patterns. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Dec. 2020.
[21] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. Cryptology ePrint Archive, Report 1998/003, Feb. 1998.
[22] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1995.
[23] H. Corrigan-Gibbs and D. Kogan. Private information retrieval with sublinear online time. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 2020.
[24] D. Demmler, A. Herzberg, and T. Schneider. RAID-PIR: Practical multi-server PIR. In *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, Nov. 2014.
[25] C. Devet, I. Goldberg, and N. Heninger. Optimally robust private information retrieval. In *Proceedings of the USENIX Security Symposium*, Aug. 2012.
[26] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the USENIX Security Symposium*, Aug. 2004.
[27] C. Dong and L. Chen. A fast single server private information retrieval protocol with low communication cost. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Sept. 2014.
[28] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of cloudlab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.
[29] C. Gentry and Z. Ramzan. Single-database private information retrieval with constant communication rate. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, July 2005.
[30] N. Gilboa and Y. Ishai. Distributed point functions and their applications. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2014.
[31] I. Goldberg. Improving the robustness of private information retrieval. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2007.
[32] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4), Oct. 1986.
[33] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3), May 1996.
[34] M. Green, W. Ladd, and I. Miers. A protocol for privately reporting ad impressions at scale. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2016.
[35] T. Gupta, N. Crooks, W. Mulhern, S. Setty, L. Alvisi, and M. Walfish. Scalable and private media consumption with Popcorn. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2016.
[36] S. Hohenberger, V. Koppula, and B. Waters. Adaptively secure puncturable pseudorandom functions in the standard model. Cryptology ePrint Archive, Report 2014/521, 2014.
[37] D. Kales, O. Omolola, and S. Ramacher. Revisiting user privacy for certificate transparency. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.
[38] A. Kiayias, N. Leonardos, H. Lipmaa, K. Pavlyk, and Q. Tang. Optimal rate private information retrieval from homomorphic encryption. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2015.
[39] A. Kiayias, S. Papadopoulos, N. Triandopoulos, and T. Zacharias. Delegatable pseudorandom functions and applications. Cryptology ePrint Archive, Report 2013/379, 2013.
[40] D. Kogan and H. Corrigan-Gibbs. Private blocklist lookups with Checklist. In *Proceedings of the USENIX Security Symposium*, Aug. 2021.
[41] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1997.
[42] A. Kwon, D. Lazar, S. Devadas, and B. Ford. Riffle: An efficient communication system with strong anonymity. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2016.
[43] H. Lipmaa. First CPIR protocol with data-dependent computation. In *Proceedings of the International Conference on Information, Security and Cryptology (ICISC)*, Dec. 2009.
[44] H. Lipmaa and K. Pavlyk. A simpler rate-optimal CPIR protocol. In *Proceedings of the International Financial Cryptography Conference*, Apr. 2017.
[45] W. Lueks and I. Goldberg. Sublinear scaling for multi-client private information retrieval. In *Proceedings of the International Financial Cryptography and Data Security Conference*, Jan. 2015.
[46] Y. Ma, K. Zhong, T. Rabin, and S. Angel. Incremental offline/online PIR (extended version). Cryptology ePrint Archive, Oct. 2021. http://eprint.iacr.org/2021/.
[47] Y. Ma, K. Zhong, T. Rabin, and S. Angel. Incremental offline/online PIR. In *Proceedings of the USENIX Security Symposium*, Aug. 2022.
[48] P. Mittal, F. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg. PIR-Tor: Scalable anonymous communication using private information retrieval. In *Proceedings of the USENIX Security Symposium*, Aug. 2011.
[49] H. Mozaffari and A. Houmansadr. Heterogeneous private information retrieval. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2020.
[50] J. Patarin. Security of random feistel schemes with 5 or more rounds. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2004.
[51] S. Patel, G. Persiano, and K. Yeo. Private stateful information retrieval. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2018.
[52] E. Shi, W. Aqeel, B. Chandrasekaran, and B. Maggs. Puncturable pseudorandom sets and private information retrieval with near-optimal online bandwidth and time. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2021.
[53] X. Yi, M. G. Kaosar, R. Paulet, and E. Bertino. Single-database private information retrieval from fully homomorphic encryption. *IEEE Transactions on Knowledge and Data Engineering*, 25(5), May 2013.

A Proofs and analysis for incremental CK

A.1 Proofs for strawman proposal

Theorem 3 (Batched addition)**.** For a random subset $S \subseteq [n]$ of size s, construct S' as follows:

- sample a number w from $\mathcal{HG}(n+m, m, s)$,
- randomly choose w elements from $[n+1, n+m]$,

- replace w random elements in S with the above w elements.

Then S' is a random subset of size s of $[n+m]$.

Proof. We show that any index $i \in [n+m]$ is contained in the resulting set S' with the same probability $s/(n+m)$.

We examine the following two cases.

Case 1. For a fixed $i \in [n+1, n+m]$, S' contains i if and only if i is in the sampled w elements. That is,

$$\Pr[i \in S'] = \sum_{z=1}^{m} \Pr[w = z] \cdot \frac{z}{m} \quad \text{if } s \geq m, \text{ and}$$

$$\Pr[i \in S'] = \sum_{z=1}^{s} \Pr[z = 1] \cdot \frac{z}{m} \quad \text{if } s < m.$$

By the hypergeometric distribution formula, if $s \geq m$ the RHS:

$$\frac{1}{C_{n+m}^{s}} \cdot (C_{m-1}^{0} C_{n}^{s-1} + C_{m-1}^{1} C_{n}^{s-2} + ... + C_{m-1}^{m-1} C_{n}^{s-m})$$

and if $s < m$ the RHS is:

$$\frac{1}{C_{n+m}^{s}} \cdot (C_{m-1}^{0} C_{n}^{s-1} + C_{m-1}^{1} C_{n}^{s-2} + ... + C_{m-1}^{s-1} C_{n}^{0})$$

Both terms equal $C_{n+m-1}^{s-1}/C_{n+m}^{s} = s/(n+m)$.

Case 2. For a fixed $i \in [n]$, S' contains i if and only if i is sampled into S and i has not been kicked out by the replacement with the w sampled elements. Therefore we have

$$\Pr[i \in S'] = \frac{s}{n} \cdot (\sum_{z=0}^{m} \Pr[w = z] \cdot (1 - \frac{z}{s})) \quad \text{if } s \geq m, \text{ and}$$

$$\Pr[i \in S'] = \frac{s}{n} \cdot (\sum_{z=0}^{s} \Pr[w = z] \cdot (1 - \frac{z}{s})) \quad \text{if } s < m,$$

where the leading factor s/n captures i being selected into the set initially, and the rest stands for the probability that i remains in the set. Using the result from the previous case, we have

$$\begin{aligned}
\Pr[i \in S'] &= \frac{s}{n}(1 - \sum_{z=1}^{m} \Pr[w = z] \cdot \frac{z}{s}) \\
&= \frac{s}{n}(1 - \frac{m}{s}(\sum_{z=1}^{m} \Pr[w = z] \cdot \frac{z}{m})) \\
&= \frac{s}{n} \cdot (1 - \frac{m}{s} \cdot \frac{s}{m+n}) = \frac{s}{n+m} \quad \text{for } s \geq m, \\
\Pr[i \in S'] &= \frac{s}{n}(1 - \sum_{z=1}^{s} \Pr[w = z] \cdot \frac{z}{s}) \\
&= \frac{s}{n}(1 - \frac{m}{s}(\sum_{z=1}^{s} \Pr[w = z] \cdot \frac{z}{m})) \\
&= \frac{s}{n} \cdot (1 - \frac{m}{s} \cdot \frac{s}{m+n}) = \frac{s}{n+m} \quad \text{for } s < m.
\end{aligned}$$

□

A.2 Secure puncturable PRS from partially secure PRP

In this section, we show that in order to ensure security for puncturable PRS, having a *partially* secure PRP suffices. In order to use prior proofs from CK [23], we view our construction of incremental PRS as a puncturable PRS with trivial puncturing.

Definition 5 (Puncturable PRS security [23])**.** For $\lambda, n \in \mathbb{N}$, and a puncturable PRS $\Phi = (\mathsf{Gen}, \mathsf{Punc}, \mathsf{Eval})$ with size s, define the following game, played between a challenger and an adversary.

The challenger executes the following steps:

- $k \leftarrow \Phi.\mathsf{Gen}(1^\lambda, n)$
- $S \leftarrow \Phi.\mathsf{Eval}(k)$
- $x^* \xleftarrow{R} S$; $k_p \leftarrow \Phi.\mathsf{Punc}(k, x^*)$

and sends 1^λ and k_p to the adversary. The adversary outputs an integer $x' \in [n]$ and "wins" if $x^* = x'$. The advantage is defined as

$$\mathsf{AdvPPRS}(\lambda) = \Pr[x^* = x] - \frac{1}{n-s+1}.$$

We call a PPRS ϵ-secure if the above advantage is less than ϵ, where ϵ is a negligible function on the security parameter λ.

Lemma 1. Given a $\mathsf{PRP} = (\mathsf{Gen}, \mathsf{Eval}) : [n] \to [n]$ that is ϵ-secure (where ϵ is a negligible function on λ) over $s(n)$ queries, the following puncturable PRS Φ with linear-sized puncturable keys is ϵ-secure.

Proof. We modify Construction 4 in CK [23] as follows:

- $\Phi.\mathsf{Gen}(1^\lambda, n) \to k$:
  Sample $k \leftarrow \mathsf{PRP.Gen}(1^\lambda, n)$.
- $\Phi.\mathsf{Eval}(k) \to S$:
  Compute $S \leftarrow \{\mathsf{PRP.Eval}(k, 1), \ldots, \mathsf{PRP.Eval}(k, s(n))\}$.
- $\Phi.\mathsf{Punc}(k, i) \to k_p$:
  Define k_p as the elements in $\Phi.\mathsf{Eval}(k) \backslash \{i\}$.

Suppose we have an adversary $\mathcal{A}$ that breaks puncturable PRS security of Φ, then we can construct an adversary $\mathcal{B}$ that breaks the security of PRP. We denote the challenger in the PRP game as $\mathcal{C}$, where it responds to queries either using random permutation F or a pseudorandom permutation $\mathsf{PRP.Eval}(k, \cdot)$. The construction of $\mathcal{B}$ is as follows.

- $\mathcal{B}$ sends to $\mathcal{C}$ s inputs $x_1, \ldots, x_s$, and gets s distinct response values. Denote the set formed by these values as S.
- $\mathcal{B}$ removes a random element y in S and sends the resulting $s-1$ elements to $\mathcal{A}$.
- $\mathcal{A}$ outputs y'.
- $\mathcal{B}$ outputs 1 if $y' = y$, and 0 otherwise.

The advantage of $\mathcal{B}$ in PRP game is

$$\mathsf{AdvPRP}(\lambda) = |\Pr[\mathcal{B}(\mathsf{PRP.Eval}(k, \cdot)) = 1] - \Pr[\mathcal{B}(F(\cdot)) = 1]|.$$

Since F is a truly random permutation, $\Pr[\mathcal{B}(F(\cdot)) = 1] = 1/(n-s+1)$, i.e., $\mathcal{A}$ randomly guesses an element that is not in the $s-1$ received elements. And due to the construction of $\mathcal{B}$, we have that $\Pr[\mathcal{B}(\mathsf{PRP.Eval}(k, \cdot)) = 1] = \Pr[y' = y]$. In other words,

$$\mathsf{AdvPRP}(\lambda) = \Pr[x^* = x] - \frac{1}{n-s+1} = \mathsf{AdvPPRS}(\lambda),$$

i.e., $\mathcal{B}$ breaks the security of PRP. Contradiction. □

The above theorem implies that we only need a PRP (with range $[n]$) that is secure against $s(n)$ queries. This is crucial for efficiency, since for small range size, fully secure PRPs are more inefficient than partially secure PRPs.

A.3 Properties of incremental PRS

A.3.1 Non-triviality definition

Recall from Section 5 the definition of $g(n)$: the expected size of $U \ominus U'$, where U, U' are sets (both of size $s(n)$) uniformly sampled from $[n]$ and $[n + o(n)]$ respectively.

Theorem 4. For uniformly sampled size-s set U from $[n]$ and set U' from $[n + o(n)]$, $g(n) = E[|U \ominus U'|] = s^2/(n^2 + o(n^2))$.

Proof. We call the sampling of U as trial 1 and the sampling of U' as trial 2. We define a random variable X_i to indicate whether i is sampled in both trials. That is, $X_i = 1$ if and only if i is sampled into both U and U'. Note that $|U \cap U'| = \sum_{i\in[n+o(n)]} X_i$.

Now we examine two cases for the random variable X_i:

For $1 \le i \le n$, a uniform size-s subset of $[n]$ contains i with probability $C_{n-1}^{s-1}/C_n^s = s/n$. Similarly, a uniform size-s subset of $[n+o(n)]$ contains i with probability $s/(n+o(n))$. Hence $E[X_i] = \Pr[X_i = 1] = (s/n) \cdot (s/(n + o(n))) = s^2/(n^2 + o(n^2))$.

For $n + 1 \le i \le n + o(n)$, $E[X_i] = \Pr[X_i = 1] = 0$ because it is impossible for i to appear in U.

Therefore we have $E|U \cap U'| = E[\sum_{i\in[n+o(n)]} X_i] = \sum_{i\in[n+o(n)]} E[X_i] = \sum_{i\in[n]} E[X_i] = s^2/(n + o(n))$. □

For the proofs in the following sections, we will use the above theorem for $s(n) = \sqrt{n}$. Under this case, $E[|U \cap U'|] \le 1$, which means $E|U \ominus U'| \ge 2(\sqrt{n} - 1)$. On the other hand, $E[|U \ominus U'|] \le 2\sqrt{n}$, then we have $E|U \ominus U'| = \Theta(\sqrt{n})$.

A.3.2 Proof of Theorem 1

We use $\binom{[n]}{s}$ to denote the set of all subsets of $[n]$ with size s. We use $\{dist\}$ to denote a distribution ensemble, $\approx_c$ for *computationally indistinguishability* between two distribution ensembles.

Lemma 2 (Puncturable security implies pseudorandomness of a set [23])**.** Let $\Phi = (\mathsf{Gen}, \mathsf{Punc}, \mathsf{Eval})$ be an ϵ-secure PPRS with set size $s(n)$. For $\lambda, n \in \mathbb{N}$, define the two distributions

$$\mathcal{P}_{\mathsf{pseudo}} := \{\Phi.\mathsf{Eval}(\Phi.\mathsf{Gen}(1^\lambda, n))\}; \mathcal{P}_{\mathsf{true}} := \left\{ S \xleftarrow{R} \binom{[n]}{s} \right\}.$$

Then for every PPT adversary $\mathcal{A}$, the two distributions are computationally indistinguishable.

The proof is in Appendix B.1 of CK [23].

We will now introduce a series of helper lemmas before proving properties for incremental PRS. For notation simplicity, we use a superscript to denote the number n in $\mathcal{P}_{\mathsf{pseudo}}$ or $\mathcal{P}_{\mathsf{true}}$.

Lemma 3 (Proposed protocol in Section 4.2.1)**.** Define

$$\mathcal{P}_{\mathsf{single}} = \left\{ \begin{array}{lll} & S \xleftarrow{R} \binom{[n]}{s} & \\ & b \leftarrow & \mathsf{Bernoulli}(s/(n+1)) \\ S' : & \text{if } b = 0, & \text{do nothing} \\ & \text{if } b = 1, & i \xleftarrow{R} S, \\ & & S' \leftarrow (S\backslash\{i\}) \cup \{n+1\}, \end{array} \right\}$$

Then $\mathcal{P}_{\mathsf{single}} = \mathcal{P}_{\mathsf{true}}^{[n+1]}$.

Proof. Rewrite $\mathcal{P}_{\mathsf{true}}^{[n+1]}$ as

$$\left\{ \begin{array}{lll} & S \xleftarrow{R} \binom{[n+1]}{s} & \\ S' : & \text{if } n+1 \notin S, & S' \leftarrow S \\ & \text{if } n+1 \in S, & S' \leftarrow S\backslash\{n+1\}, \\ & & S' \leftarrow S' \cup \{n+1\} \end{array} \right\}$$

Define $\mathcal{P}_{\mathsf{true}}^*$ as

$$\left\{ \begin{array}{lll} & S \xleftarrow{R} \binom{[n+1]}{s} & \\ S' : & \text{if } n+1 \notin S, & S' \xleftarrow{R} \binom{[n]}{s} \\ & \text{if } n+1 \in S, & S' \xleftarrow{R} \binom{[n]}{s-1}, \\ & & S' \leftarrow S' \cup \{n+1\} \end{array} \right\}$$

and $\mathcal{P}'_{\mathsf{true}}$ as

$$\left\{ \begin{array}{lll} & b \leftarrow & \mathsf{Bernoulli}(s/(n+1)) \\ S : & \text{if } b = 0, & S \xleftarrow{R} \binom{[n]}{s} \\ & \text{if } b = 1, & S \xleftarrow{R} \binom{[n]}{s-1}, S \leftarrow S \cup \{n+1\} \end{array} \right\}$$

On one hand, $\mathcal{P}_{\mathsf{true}}^{[n+1]} = \mathcal{P}_{\mathsf{true}}^*$ because when $n + 1 \notin S$, the s elements of S are uniformly at random distributed over $[n]$ (this is by definition of randomly sampling s elements). Similarly, the same argument applies when $n + 1 \in S$.

On the other hand, $\mathcal{P}_{\mathsf{true}}^* = \mathcal{P}'_{\mathsf{true}}$, because in the former, the the only role of S is to decide which case out of the two cases (since S' is freshly generated anyway), so it can be captured by a Bernoulli distribution. Hence $\mathcal{P}_{\mathsf{true}}^{[n+1]} = \mathcal{P}'_{\mathsf{true}}$.

Now we look at $\mathcal{P}_{\mathsf{single}}$. Since Bernoulli sampling is independent of sampling S, we can swap the first two steps in $\mathcal{P}_{\mathsf{single}}$ and get

$$\mathcal{P}'_{\mathsf{single}} = \left\{ \begin{array}{lll} & b \leftarrow & \mathsf{Bernoulli}(s/(n+1)) \\ S' : & \text{if } b = 0, & S \xleftarrow{R} \binom{[n]}{s} \\ & \text{if } b = 1, & S \xleftarrow{R} \binom{[n]}{s}, i \xleftarrow{R} S, \\ & & S' \leftarrow (S\backslash\{i\}) \cup \{n+1\} \end{array} \right\}$$

Simplifying the $b = 1$ case in $\mathcal{P}'_{\mathsf{single}}$, we have

$$\left\{ \begin{array}{lll} & b \leftarrow & \mathsf{Bernoulli}(s/(n+1)) \\ S : & \text{if } b = 0, & S \xleftarrow{R} \binom{[n]}{s} \\ & \text{if } b = 1, & S \xleftarrow{R} \binom{[n]}{s-1}, S \leftarrow S \cup \{n+1\} \end{array} \right\},$$

which is exactly $\mathcal{P}'_{\mathsf{true}}$. □

Lemma 4 (Batched version in Section 4.2.1)**.** Define

$$\mathcal{P}_{\mathsf{multi}} := \left\{ \begin{array}{lll} & w \leftarrow & \mathcal{HG}(n+m, m, s) \\ S^1 \cup S^2 : & S^1 \xleftarrow{R} & \binom{[n]}{s-w} \\ & S^2 \xleftarrow{R} & \binom{[n+1,n+m]}{w} \end{array} \right\}$$

Then $\mathcal{P}_{\mathsf{true}}^{[n+m]} = \mathcal{P}_{\mathsf{multi}}$.

Proof. Define $\mathcal{P}'_{\mathsf{true}}$ as

$$\left\{ \begin{array}{ll} & S \xleftarrow{R} \binom{[n+m]}{s} \\ & \text{Let } w \text{ be the number of elements} \\ S^1 \cup S^2 : & \text{of } S \text{ in } [n+1, n+m] \\ & S^1 \xleftarrow{R} \binom{[n]}{s-w} \\ & S^2 \xleftarrow{R} \binom{[n+1,n+m]}{w} \end{array} \right\}$$

On the one hand, we have $\mathcal{P}'_{\mathsf{true}} = \mathcal{P}^{[n+m]}_{\mathsf{true}}$ by applying a similar approach to Lemma 3. On the other hand, by the definition of the hypergeometric distribution, $\mathcal{P}'_{\mathsf{true}} = \mathcal{P}_{\mathsf{multi}}$. □

Theorem 5 (Correctness)**.** Let $\Psi = (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Add})$ be the incremental PRS in Section 5.3. Then Ψ satisfies correctness.

Proof. First, given $aux = [(n, s)]$, $\Psi.\mathsf{Eval}$ will output a set in $\binom{[n]}{s}$ by construction. Second, after m indices are added, the new set will be $Set^{[n]}(sk, t_1) \cup Set^{[m]}(k_2, t_2)$ for some t_1, t_2 and $t_1 + t_2 = s$. Therefore the output set is a subset of $[n + m]$ of size s. □

Theorem 6 (Security)**.** Let $\Psi = (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Add})$ be the incremental PRS in Section 5.3. Then Ψ satisfies security.

Proof. The distribution generated by Ψ can be written as

$$\mathcal{P}_{\mathsf{prs}} := \left\{ S : \begin{array}{r} (k, aux) \leftarrow \Psi.\mathsf{Gen}(1^\lambda, n) \\ aux' \leftarrow \Psi.\mathsf{Add}(k, m) \\ S \leftarrow \Psi.\mathsf{Eval}(k, aux') \end{array} \right\}$$

By construction of Ψ, we rewrite $\mathcal{P}_{\mathsf{prs}}$ as

$$\left\{ S^1 \cup S^2 : \begin{array}{rl} w \leftarrow & \mathcal{HG}(n+m, m, s) \\ S^1 \xleftarrow{R} & Set^{[n]}(k_1, s - w) \\ S^2 \xleftarrow{R} & Set^{[m]}(k_2, w) + n \end{array} \right\}$$

Suppose we randomly sample k_1, k_2 from a PRP key space $\mathcal{K}$. By Lemma 2 and 4, we have $\{\mathcal{P}_{\mathsf{prs}}\}_\lambda \approx_c \{\mathcal{P}^{[n+m]}_{\mathsf{true}}\}_\lambda$. □

Theorem 7 (Non-triviality)**.** Let $\Psi = (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Add})$ be the incremental PRS in Section 5.3. Then Ψ satisfies non-triviality.

Proof. Let S and S' be the sets defined in Section 5.3. We know that $|S \ominus S'| = 2w$, where $w \leftarrow \mathcal{HG}(n + m, m, s)$, and $E[w] = \frac{m}{m+n} \cdot s$. Since $s = \sqrt{n}$, and $m = o(n)$, $E[|S \ominus S'|] = 2E[w] = 2 \cdot \frac{o(1) \cdot \sqrt{n}}{1+o(1)} = o(\sqrt{n})$. Therefore, $E[w] = o(g(n))$.

A.4 Properties of incremental CK construction

A.4.1 Eval with shifts

We modified Eval to take shifts into account. $\mathsf{Eval}(k, aux, sh)$ takes in a secret key k, aux and a shift sh, parses aux as $[(r_\ell, t_\ell)]_{\ell \in [L]}$, derives keys from k as $k_1, \ldots, k_L$, and computes the first subset $S_1 = \{\mathsf{PRP}^{[r_1]}(k_1, 1), \ldots, \mathsf{PRP}^{[r_1]}(k_1, t_1)\} + sh \mod r_1$ and the subsequent subsets as $S_2, \ldots, S_L$ as in Section 5.3. Then it outputs $\cup_{\ell \in [L]} S_\ell$.

A.4.2 Proof of Theorem 2

We first summarize a modification to CK in Checklist [40, §4.2] that doubles the size of the online query and refresh, but that reduces the correctness error of the online phase from $(s-1)/n$ to $\mathrm{negl}(\lambda)$, where s is the set size and n is the size of the database. Then we prove that our construction with this improvement incorporated satisfies correctness in Definition 2.

Checklist's proposal ensures that the client can always get the desired item at index i regardless of the result of the Bernoulli sampling (Figure 7, line 10 and 19). It works as follows.

The client samples $\beta \leftarrow \mathsf{Bernoulli}(2(s-1)/n)$. If $\beta = 0$, the client generates S_{offline} and finds S_{online} (both containing i) as in Figure 7. Additionally, it picks two random indices, γ_{offline} from $S_{\text{offline}} \setminus \{i\}$ and γ_{online} from $S_{\text{online}} \setminus \{i\}$. It sends to each server a set, along with an index: to the offline server it sends $(S_{\text{offline}} \setminus \{i\}, \gamma_{\text{offline}})$; to the online server it sends $(S_{\text{online}} \setminus \{i\}, \gamma_{\text{online}})$. Each server sends back the parity for the set (b bits) and the data item for the index (b bits). The client will only use the parity for $S_{\text{online}} \setminus \{i\}$ to recover the desired item at index i (Query), and use the parity for $S_{\text{offline}} \setminus \{i\}$ and block i to replace the used set ($\mathsf{Refresh}$). The two random indices γ_{offline} and γ_{online} are not used for query nor refresh; they are used for security reasons (see the $\beta = 1$ case below).

If $\beta = 1$, the client does *not* use any of its locally stored $\sqrt{n} \log n$ sets (hints). Instead, it generates a new set S_{new} that contains i. It randomly samples γ from $S_{\text{new}} \setminus \{i\}$ and γ' from $S_{\text{new}} \setminus \{\gamma\}$. Unlike in the original protocol where only the online server processed queries during the online phase, in this refined version the offline server is also asked to help with online queries. Now the client picks one of the two servers at random (it does so to hide which server helps to compute the parity for $S_{\text{new}} \setminus \{i\}$), and sends the chosen server $(S_{\text{new}} \setminus \{i\}, \gamma)$, and sends $(S_{\text{new}} \setminus \{\gamma\}, \gamma')$ to the other server. Each server sends back the parity for the provided set and the data item for the provided index (γ or γ') to the client. The client then uses three things: the parity for $S_{\text{new}} \setminus \{i\}$, the parity for $S_{\text{new}} \setminus \{\gamma\}$, and the data item at index γ, to recover the item at index i. The random index γ' is not used for recovering the data item; it is used to hide which server plays the role of computing $S_{\text{new}} \setminus \{i\}$. After the client recovers data block i, it discards S_{new} and never uses it again for future queries. In this case, the $\sqrt{n} \log n$ sets and their parities remain unchanged so there is no need to refresh any sets.

Theorem 8 (Correctness)**.** Let $\Pi = (\mathsf{Prep}, \mathsf{IncPrep}, \mathsf{Query}, \mathsf{Refresh}, \mathsf{Resp}, \mathsf{QueryRecov}, \mathsf{RefreshRecov})$ be the scheme in Section 6, and suppose the underlying construction of incremental PRS Ψ satisfies correctness. Then Π with the above improvement satisfies correctness in Definition 2.

Proof. Correctness contains two parts: the hint should be updated correctly, and the client should fetch its desired item during the online phase. The former follows directly from Algorithms 4 and 5. For the latter, since Algorithm 7 is compatible with Checklist's improvement (proved correct in Checklists's Lemma B.1 [40]), we can reduce the correctness error to $\mathrm{negl}(\lambda)$. □

Theorem 9 (Security)**.** Let $\Pi = (\mathsf{Prep}, \mathsf{IncPrep}, \mathsf{Query}, \mathsf{Refresh}, \mathsf{Resp}, \mathsf{QueryRecov}, \mathsf{RefreshRecov})$ be the scheme in Section 6, and suppose the incremental PRS Ψ satisfies correctness and security. Then Π satisfies security in Definition 2.

Proof. Security contains two parts: the hint update should not reveal any information of queried index and the queries during online phase should look indistinguishable (since $\mathsf{Refresh}$ is symmetric with Query, hence we only consider the latter here). The former is independent of the query index, hence security holds. For the latter, we show that, for databases D and D' where $|D| = n$ and $|D'| = n + o(n)$, the following two distributions,

$$\mathcal{P}_{\text{online}} = \left\{ q_{\text{online}} : \begin{array}{r} h \leftarrow \mathsf{Prep}(D) \\ h' \leftarrow \mathsf{IncPrep}(D, \delta, h) \\ q_{\text{online}} \leftarrow \mathsf{Query}(h', i) \end{array} \right\}$$

and

$$\mathcal{P}'_{\text{online}} = \left\{ q_{\text{online}} : \begin{array}{r} h' \leftarrow \mathsf{Prep}(D') \\ q_{\text{online}} \leftarrow \mathsf{Query}(h', i) \end{array} \right\}$$

are computationally indistinguishable ($\{\mathcal{P}_{\text{online}}\}_\lambda \approx_c \{\mathcal{P}'_{\text{online}}\}_\lambda$).

We define $\widetilde{\mathcal{P}}_{\text{online}}$ to be the distribution of outputting the set before removing an element in Query. From Lemma 6, we know $\{\widetilde{\mathcal{P}}_{\text{online}}\}_\lambda \approx_c \{\widetilde{\mathcal{P}}'_{\text{online}}\}_\lambda$. And by Lemma 45 in CK [23], the conclusion holds. Furthermore, Lemma B.2 in Checklist [40] shows that the aforementioned improvement maintains security. □

Theorem 10 (Non-triviality)**.** Let $\Pi = (\mathsf{Prep}, \mathsf{IncPrep}, \mathsf{Query}, \mathsf{Refresh}, \mathsf{Resp}, \mathsf{QueryRecov}, \mathsf{RefreshRecov})$ be the scheme in Section 6, and suppose the underlying incremental PRS Ψ satisfies non-triviality. Then Π satisfies non-triviality in Definition 2.

Proof. For additions, the non-triviality for incremental CK directly follows from PRS non-triviality (Appendix A.3). We give concrete formula for IncPrep's cost here.

Suppose the database is of size n, and there is $m = o(n)$ number of elements added. For each set S_j where $j \in [J]$, the client samples w_j from the distribution $\mathcal{HG}(m, m+n, s)$. The cost of evaluating the PRPs and computing parities for data objects is $O(b\sum_{j\in[J]} w_j)$. Since $J = (n/s)\log n$, we have $\sum_{j=1}^{(n/s)\log n} E[w_j] = \frac{m\cdot s}{n+m} \cdot (n/s)\log n \le m \log n$. Therefore, the total computation to the offline server is in expectation $O(bm \log n)$.

For edits and deletions, non-triviality holds when $m|Q|$ (Section 6.1) is sublinear in n. We additionally discuss an optimization for incremental updates when $m|Q|$ is linear or even super linear in n in Appendix C. □

B Insecure strawman

Recall the strawman from Section 4.2: the client keeps the *original* $\sqrt{n}\log n$ parities and sets, then for m new items, samples a few new sets from $[n+m]$. For instance, $m = 2\sqrt{n}$ and the client samples c new sets from $[n+m]$. We require c to be meaningfully smaller than $\sqrt{n}\log n$ (say $c = o(\sqrt{n})$), otherwise, sampling new sets is as expensive as preprocessing from scratch. We informally show that the online server can distinguish between the two cases: the client repeatedly queries for indices in $[n]$ and repeatedly queries for indices in $[n+1, n+m]$. In either case, we assume the client issues $\sqrt{n}$ queries.

In the former case, most of the sets the client uses will be the old sets (even considering that the client does refresh and the fraction of random sets from $[n+m]$ gradually increases). The upper bound of the fraction of the old sets, $(\sqrt{n}\log n - \sqrt{n})/(\sqrt{n}\log n + c)$, is noticeably larger than $2/3$, given n is large and c is small. This implies that with probability larger than $2/3$, the server will not see any of the new indices in all these $\sqrt{n}$ online queries.

In the latter case, the client will always use the new sets for query. Note that for a sufficient number of size-$\sqrt{n}$ sets from $[n + 2\sqrt{n}]$, at least one of them contains two new indices with noticeable probability. Concretely, when the number of sets equals $\sqrt{n}$, the above probability is larger than $2/3$. Therefore, at least $2/3$ chance, the server will see a new index (or indices) in these $\sqrt{n}$ online queries.

C Optimization and corner cases

In Section 6.1 we mention that the database updates could trigger elements changed in every set of the client (this happens when m_{add} is relatively large). In such cases, we let the client send the key and aux for each set to the server (Figure 14). The server computes parity difference for all the sets (Figure 13) and the client uses the responses to update the hints for every set.

1: **procedure** $\mathsf{HintRes}(D', u_q) \to u_r$
2: Initialize $(p'_1, \ldots, p'_J)$ all to 0^b
3: Parse u_q as $\{(k_j, aux_j, aux'_j)\}_{j\in[J]}$
4: **for** $j \in [J]$ **do**
5: // process additions
6: Compute $S \leftarrow \Psi.\mathsf{EvalDiff}(k_j, aux_j, aux'_j)$
7: Update $p'_j \leftarrow \bigoplus_{e\in S} D[e]$
8: // process edits
9: Update $p'_j \leftarrow p'_j \bigoplus (X_z \oplus X'_z), \forall z \in [m_{\text{edit}}]$
10: such that $\mathsf{Member}(x_z, (k_j, aux'_j)) = 1$
11: // process deletions
12: Update $p'_j \leftarrow p'_j \bigoplus (Y_z \oplus Y'_z), \forall z \in [m_{\text{del}}]$
13: such that $\mathsf{Member}(y_z, (k_j, aux'_j)) = 1$
14: Output $u_r \leftarrow (p'_1, \ldots, p'_J)$

FIGURE 13—Offline server responds to an update request when each set has changed. The X, Y arrays and $m_{\text{edit}}, m_{\text{del}}$ are defined as in Figure 3.

1: **procedure** $\mathsf{HintReq}(h, \delta)$
2: Parse h as $\{(k_j, aux_j, p_j)\}_{j\in[J]}$
3: Initialize each entry in $(aux'_1, \ldots, aux'_J)$ as $\perp$
4: **for** $j \in [J]$ **do**
5: Compute $aux'_j \leftarrow \Psi.\mathsf{Add}(aux_j, m)$
6: Output $u_q \leftarrow \{(k_j, aux_j, aux'_j)\}_{j\in[J]}$

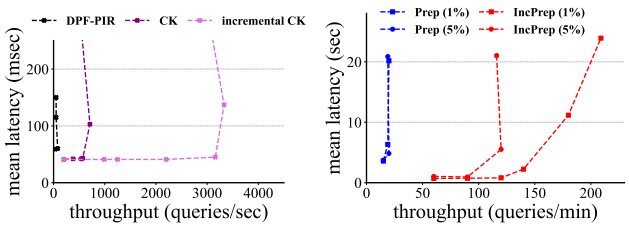
FIGURE 14—Client algorithms for hint request.

A caveat for edits (or deletions) is that, the server computation is $m_{\text{edit}}|Q|$ (Figure 4) or $m_{\text{edit}}|J|$ (Figure 13), and this could be linear (or even superlinear) in the database size when m_{edit} is large. In such a case, the offline server can directly send the parity bits for edits (or deletions) to the client, and the client updates the hints and discards the parity bits. This keeps both communication and server computation $O(m_{\text{edit}})$ (or $O(m_{\text{del}})$), while the client does more work for finding which sets should be updated (i.e., the client does line 9–14 in Figure 13 instead of the server).

D Additional evaluations

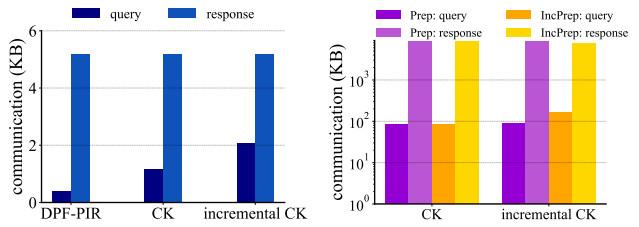
We additionally evaluate PIR-Tor for a hypothetical network with 70K relays.

Figure 15a shows the throughput results for online queries. Similar to the 7K setting, incremental CK (owing to its lack of cryptographic operations performed by the online server) achieves higher throughput. For online communication (Figure 16a), ours incurs the highest costs. This suggests that when databases are large, sending (punctured) keys as online queries is much more efficient than



(a) Online phase. (b) Preprocessnig and hint updates

FIGURE 15—Mean latency and server throughput under varying load during online phase for DPF, CK, and incremental CK.



(a) Online phase. (b) Preprocessing and hint updates.

FIGURE 16—Communication cost for DPF-PIR, CK, and incremental CK.

sending explicit indices.

For preprocessing and hint updates, the throughput (Figure 15b) and communication (Figure 16b) shows similar patterns as in 7K setting. This is because the benefit of incremental updates over preprocessing from scratch depends on the fraction of changes but not the absolute value of database size.

E Incremental OO-PIR from SACM

One of the most unusual features of our incremental CK scheme is that a client needs to kick out existing indices from their sets in order to add new indices. Part of the reason for this is that the initial sets in CK all have the same size, and we wish to maintain this invariant to ensure that the online server cannot make any inferences from set sizes. A natural question is then whether one can relax this restriction and build an incremental PRS that does not require replacement.

Indeed, having initial sets of the same size is not necessary: one can instead have an offline phase where Prep (§4.1) adds each element in $[n]$ to each of set with probability p. In expectation, each set will be of size np after the initial preprocessing is done.

This same observation is made by Shi et al. [52]. The key contribution in their work, which we call SACM, is a construction of a compressed representation of a PRS with the property that sets can be of different sizes. Given such a compressed representation (effectively a PPRS), one can use the same offline/online PIR framework as CK (§4.1). We therefore show how to adapt the PPRS in SACM to support our notion of incrementality, and obtain our second construction of an incremental offline/online PIR scheme.[6]

E.1 Overview of SACM

Pseudorandom set in SACM. In SACM, each index i is viewed as a binary string. We abuse the notation and use i to denote both the index value and its binary representation string. A set is still succinctly represented by a key as in CK, but the elements in the set are determined differently from CK. Specifically, given a PRF: $\{0,1\}^* \to \{0,1\}$, and a key k from the PRF key space, an element $i \in [n]$ is defined to be in the set if and only if evaluating the PRF using key k on $\lceil 1/2 \log n \rceil$ suffixes of i results in 1 for *all* evaluations (Figure 17).

Intuitively, we can view this process as tossing a random coin $\lceil 1/2 \log n \rceil$ times to determine if i is in the set. Therefore, the probability that i is sampled into the set is $p(n) = (1/2)^{\lceil 1/2 \log n \rceil}$, which is roughly $1/\sqrt{n}$, and since the initial database is of size n, each set has size in expectation of $\sqrt{n}$ elements.

[6] Even though SACM is concretely inefficient, we still provide the incremental construction based on it as an extension in theory.

1: Given PRF $f : \mathcal{K} \times \{0,1\}^* \to \{0,1\}$
2: Given $i \in [n], k \in \mathcal{K}$
3: **procedure** Member(i, k)
4: Set $\alpha = 1$
5: Set $Z = \lceil \log n \rceil - \lceil 1/2 \log n \rceil$
6: **for** $z \in [Z]$ **do**
7: **if** $f(k, \mathsf{suffix}(i, z)) = 0$ **then** set $\alpha = 0$ and **break**
8: Output α

FIGURE 17—Membership testing in SACM (strawman). The suffix of i starting from position z of i is denoted as $\mathsf{suffix}(x, z)$.

Given a set key k, to enumerate all the elements in the set specified by k, naively the client needs to call Member on every element in $[n]$. SACM [52] uses a dynamic programming approach, where enumerating all the elements in a single set specified by k costs $O(\sqrt{n}\,\mathrm{polylog} n)$ in expectation. Specifically, it starts from an empty set $I_{\lceil 1/2 \log n \rceil}$, and calls the PRF on every binary string with length $\lceil 1/2 \log n \rceil$. If the result is 1, it adds the string to $I_{\lceil 1/2 \log n \rceil}$. Then it initialize an empty set $I_{\lceil 1/2 \log n \rceil + 1}$. For each string $\omega \in I_{\lceil 1/2 \log n \rceil}$, call PRF on $1 \| \omega$ and $0 \| \omega$. If the result is 1, add the string to $I_{\lceil 1/2 \log n \rceil + 1}$. Proceed until it gets a set consisting of binary strings of length $\lceil \log n \rceil$.

To summarize, the above PRS gives two algorithms, Member and Eval. For the former, given an index i, and given a set key k, one can efficiently determine whether i is in the set specified by k. For the latter, given a set key k, one can efficiently compute all the elements in the set specified by k.

Strawman—problems and solutions. A strawman offline/online PIR construction that uses the above PRS is as follows. During the offline phase, the offline server gets $\sqrt{n}$ set keys from the client. It calls Eval on each set key, gets explicit indices for each set, and computes parity bits specified by these sets as in Section 4.1. The cost is $O(n\,\mathrm{polylog} n)$ in expectation—$\sqrt{n}$ set keys, calling Eval on each set key requires $O(\sqrt{n}\,\mathrm{polylog} n)$ computation. During the online phase, to find a desired index i, the client calls Member for each set key, and finds a set (specified by k) that contains i. Then, the client calls Eval on k and outputs all the indices in the clear. It removes i with probability $1 - p(n)$. (This is a simplified description of what SACM does.)

However, this strawman is not secure. The reason is that since the membership of an element in a set is determined by calling the PRF on the suffixes of the element, then some of the elements in a set could be *related*. For example, suppose that i_1 and i_2 differ only in the most significant bit. If the evaluation of the PRF on i_1 outputs 1 for all $\lceil 1/2 \log n \rceil$ PRF calls on suffixes of i_1, then given i_1 is in the set, the probability that i_2 is also in the set is $1/2$ instead of $1/\sqrt{n}$.

To ensure security, the set that the online server sees in the clear must contain no related elements. In other words, to remove a desired index i, the client must also remove all the indices in the set that relate to i (an index is related to i if and only if the longest common suffix no less than $\lceil 1/2 \log n \rceil$).

Although the above solves the security problem, this, in turn, brings a correctness challenge since the client might fail to recover the desired index if it has to remove other indices besides i. To address this problem, SACM prepends $H = 2\lceil \log \log n \rceil$ zeros to the binary string representation of every element $i \in [n]$, to bring down the probability that the number of related elements in a set is larger than one. Specifically, instead of calling the PRF for $\lceil 1/2 \log n \rceil$

times on suffixes of i, they call the PRF for $B + \lceil 1/2 \log n \rceil$ times, starting from $\mathsf{suffix}(0^B \| i, 1)$, until $\mathsf{suffix}(0^B \| i, B + \lceil 1/2 \log n \rceil)$. Under this tweak, for each set, an index $i \in [n]$ is sampled with probability $1/(\sqrt{n} \log^2 n)$. As a result, any two related strings are both sampled into a set with lower probability, and they proved that, under this tweak, the number of related elements in a set is in expectation $1/\log n$, and with probability $1 - o(1)$, a set contains no related elements. This means the online query fails with small probability $o(1)$, though not $\mathsf{negl}(\lambda)$. In SACM, this is called *occasional correctness* [52, §4.3].

A final complication is that, since each set has size in expectation $\sqrt{n}/\log^2 n$ due to the above tweak (recall that each element in $[n]$ is sampled with probability $1/\sqrt{n} \log^2 n$), to ensure the client can find a set that contains the desired index with high probability, they use $J = \sqrt{n} \log^3 n$ sets.

E.2 Offline Phase with Mutable Preprocessing

As in Section 4, we consider how to update a set (specified by key k) at the client when one element is appended to the database with size n. As before, we view the new index $n + 1$ as a binary string. If $n + 1$ can still be represented in $\lceil \log n \rceil$ bits, we prepend B zeros to it. If the binary representation of $n + 1$ needs at least $\lceil \log n \rceil + 1$ bits, we prepend $B - 1$ zeros to it. In either case, we denote the resulting string with length $B + \lceil \log n \rceil$ as ω.

For Member, we call the PRF using key k on suffixes of ω for $B + 1/2\lceil \log n \rceil$ times. Note that it is the same number of PRF calls as in Section E.1, therefore the probability that $n + 1$ is sampled into the set (specified by k) is still $p(n)$. That is, each index in the database is sampled to the set with probability $p(n)$, regardless of old or new indices.

For Eval, it works the same as before: it first enumerates all the strings with length $\lceil 1/2 \log n \rceil$ such that the results of calling PRF on these strings are all 1. Then it proceeds dynamic programming exactly as described in Appendix E.1. For simplicity, we denote the above procedure as $\mathsf{Eval}(k, n)$, which takes in a secret key k and outputs all the set elements in $[n]$.

A subtle complication is that the above approach will affect correctness error (recall that the correctness error comes from removing *all* the elements related to the query index i from a set). With our above tweak, the number of related elements to i is no longer $1/\log n$ in expectation. Fortunately, this is still in expectation bounded by $2/\log n$—see Lemma 5.

Hint updates. The DBUpd algorithm runs the same as in Figure 3, it sends the positions (indices) where the data items are changed. For notation clarity, the server also sends indices for additions to the client (see δ in line 3, Figure 18).

The client determines the changed elements for each set (Figure 18, line 4–10). Unlike incremental CK, no element is removed from the sets. For a number of m changes (additions, deletions, or in-place edits), the overall size of u_q is in expectation $m \cdot p(n) \cdot \sqrt{n} \log^3 n$ (each set has in expectation $m \cdot p(n)$ changes), which is $m \log n$. Then the client sends those indices in the clear to the offline server. The offline server computes parity for each set (Figure 19), and the overall cost is $O(m \log n)$. Finally, the client updates the local hints for each set.

Unfortunately, the client cost for generating the hint request u_q is sublinear in the database size only when $m = o(\sqrt{n})$. Otherwise, the client cost is linear (or superlinear when m is large), though the benefit is that the server will only do $O(m \log n)$ computation.

1: **procedure** $\mathsf{HintReq}(h, \delta) \to u_q$
2: Parse h as $\{(k_j, p_j)\}_{j \in [J]}$
3: Parse δ as $\delta_{\text{add}} = [a_1, \ldots, a_{m_{\text{add}}}]$, $\delta_{\text{edit}} = [x_1, \ldots, x_{m_{\text{edit}}}]$,
4: and $\delta_{\text{del}} = [y_1, \ldots, y_{m_{\text{del}}}]$
5: Set $\{S_j^{\text{add}}\}_{j \in [J]}, \{S_j^{\text{edit}}\}_{j \in [J]}, \{S_j^{\text{del}}\}_{j \in [J]}$ all as empty sets
6: **for** $j \in [J]$ **do**
7: **for** $e \in [a_1, \ldots, a_{m_{\text{add}}}]$ **do**
8: **if** $\mathsf{Member}(e, k_j)$ outputs 1 **then** add e to S_j^{add}
9: **for** $e \in [x_1, \ldots, x_{m_{\text{edit}}}]$ **do**
10: **if** $\mathsf{Member}(e, k_j)$ outputs 1 **then** add e to S_j^{del}
11: **for** $e \in [y_1, \ldots, y_{m_{\text{del}}}]$ **do**
12: **if** $\mathsf{Member}(e, k_j)$ outputs 1 **then** add e to S_j^{del}
13: Output $u_q \leftarrow (\{S_j^{\text{add}}\}_{j \in [J]}, \{S_j^{\text{edit}}\}_{j \in [J]}, \{S_j^{\text{del}}\}_{j \in [J]})$

FIGURE 18—Client algorithms for hint request. The total number of indices in the output sets is $O(m \log n)$, where m is the total number of additions, deletions and in-place edits.

1: **procedure** $\mathsf{HintRes}(D', u_q) \to u_r$
2: Parse u_q as $\{S_j^{\text{add}}\}_{j \in [J]}, \{S_j^{\text{edit}}\}_{j \in [J]}, \{S_j^{\text{del}}\}_{j \in [J]}$
3: Initialize each entry in $(p'_1, \ldots, p'_J)$ as 0^b
4: **for** $j \in [J]$ **do**
5: Set $p'_j \leftarrow \bigoplus_{e \in S_j^{\text{add}}} D'[e]$
6: Update $p'_j \leftarrow p'_j \bigoplus (X_z \oplus X'_z)$ for $z \in S_j^{\text{edit}}$
7: Update $p'_j \leftarrow p'_j \bigoplus (Y_z \oplus Y'_z)$ for $z \in S_j^{\text{del}}$
8: Output $u_r \leftarrow (p'_1, \ldots, p'_J)$

FIGURE 19—Offline server responds to an update request. The X-array and Y-array are the same as in Figure 3.

Considering that a server will serve many clients, it is reasonable to sacrifice client-side computation to reduce server-side computation.

E.3 Online Phase

To generate an online query with a desired index i, the client should first find a set containing i. It does so by calling Member on each set key until it finds such a set. Then the client calls Eval and outputs the indices in the clear (this is different from the SACM protocol since our tweak is not compatible with their private puncturable PRF). The client probabilistically removes i: with probability $1 - p(n)$, i will be removed; and with probability $p(n)$, i is kept in the set. The client then sends the resulting indices to the server and gets the corresponding parity.

Refresh is done similarly. A used set (key) should be discarded, and the client generates a new key. It sends the indices (with i probabilistically removed) to the offline server to fetch the parity for the new set.

E.4 Correctness and security analysis

Lemma 5 (Correctness error estimation)**.** Let Eval be the algorithm for incremental SACM construction described in Appendix E.2. For $S \leftarrow \mathsf{Eval}(k, n)$ (where k is a set key and n is the database size) and a fixed index $i \in S$, the number of elements in S that are related to i, which we denote as a random variable ζ, satisfies that $E[\zeta] \le 2/\log n$.

Proof. Consider the strings with longest suffixes of length z with the binary representation of i. Then each of such strings is in S with probability $\frac{1}{2^{\log n + B - z}}$.

Let T_z denote the set of strings that shares the longest common

suffix of length exactly z with i. We have $E[|T_z|] \leq \frac{1}{2^{\log n+B-z}} \cdot 2^{\log n-(z-1)} = \frac{1}{2^{B-1}}$. (This is because there are at most $2^{\log n+1-z}$ strings that could have the longest suffix of length z with x).

Aggregate on z, we have $E[\zeta] = \sum_{z=1/2\log n}^{1+\log n} E[T_z] \leq \sum_{z=1/2\log n}^{1+\log n} 1/2^{B-1} = (1+1/2\log n) \cdot \frac{2}{2^B} \leq \log n \cdot \frac{2}{\log^2 n} = 2/\log n$. □

To prove properties for incremental SACM, we re-define the correctness, security and non-triviality for incremental PRS as follows:

Correctness. The correctness definition is the same as in Section 5.3 except that we do not impose requirement on the set size.

Security. There are two parts. First, a PPT adversary cannot distinguish between S and U, where $S \leftarrow \mathsf{Eval}(k, n)$ for a randomly sampled k, and U is a set sampled from $[n]$ with each element sampled with probability $p(n)$. Second, a PPT adversary cannot distinguish between S' and U', where $S' \leftarrow \mathsf{Eval}(k, n + o(n))$ and U' is a set sampled from $[n + o(n)]$ with each element sampled with probability $p(n)$.

Non-triviality We define a function $g(n)$ to be $E[|U \ominus U'|]$. We say an incremental PRS satisfies non-triviality if $E[|S_1 \ominus S_2|] = o(g(n))$.

Theorem 11. Assuming the existence of a secure PRF $f : \mathcal{K} \times \{0,1\}^* \to \{0,1\}$. The incremental set construction with $\mathsf{Gen}, \mathsf{Eval}, \mathsf{Member}$ defined in Appendix E.2 satisfies correctness, security and non-triviality.

Proof. Correctness and security directly follows by construction and security follows from the security of PRF. Now we show non-triviality. Denote the set size as $s(n)$. Then by our construction, $E[|S_1 \ominus S_2|] = mp(n) = m/\sqrt{n}\log^2 n$. Note that $g(n) = O(s(n)) = \sqrt{n}/\log^2 n$. Therefore, when $m = o(n)$, $E[S_1 \ominus S_2] = o(g(n))$. □

Theorem 12. Incremental SACM satisfies *occasional correctness*, and incremental OO-PIR's security and non-triviality (Definition 2).

Proof. Correctness follows from Lemma 5 and Theorem 7.6 in SACM [52]. Security follows from incremental PRS security. Non-triviality follows from PRS non-triviality and the analysis in Section E.2. □