Simple and Practical Amortized Sublinear Private Information Retrieval using Dummy Subsets

Ling Ren
University of Illinois at Urbana-Champaign
Urbana, IL, USA
renling@illinois.edu

Muhammad Haris Mughees
University of Illinois at Urbana-Champaign
Urbana, IL, USA
mughees@illinois.edu

I Sun
University of Illinois at Urbana-Champaign
Urbana, IL, USA
is16@illinois.edu

Abstract

Recent works in amortized sublinear Private Information Retrieval (PIR) have demonstrated great potential. Despite the inspiring progress, existing schemes in this new paradigm are still faced with various challenges and bottlenecks, including large client storage, high communication, poor practical efficiency, need for non-colluding servers, or restricted client query sequences. We present simple and practical amortized sublinear stateful private information retrieval schemes without these drawbacks using new techniques in hint construction and usage. In particular, we introduce a dummy set to the client's request to eliminate any leakage or correctness failures. Our techniques can work with two non-colluding servers or a single server. The resulting PIR schemes achieve practical efficiency. The online response overhead is only twice that of simply fetching the desired entry without privacy. For a database with 2^{28} entries of 32-byte, each query of our two-server scheme consumes 34 KB of communication and 2.7 milliseconds of computation, and each query of our single-server scheme consumes amortized 47 KB of communication and 4.5 milliseconds of computation. These results are one or more orders of magnitude better than prior works.

CCS Concepts

• **Security and privacy** → **Cryptography**.

Keywords

Private information retrieval

ACM Reference Format:
Ling Ren, Muhammad Haris Mughees, and I Sun. 2024. Simple and Practical Amortized Sublinear Private Information Retrieval using Dummy Subsets. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3658644.3690266

1 Introduction

Private Information Retrieval (PIR) [8] allows a client to fetch an entry from a public database on a server without revealing which entry the client is interested in [8]. An efficient PIR scheme enables many privacy-preserving applications, such as password check [1], safe browsing [19], anonymous communication [2, 27], and private media streaming [16].

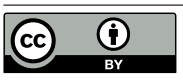
Despite decades of research [1, 2, 6, 8, 11, 13–15, 20, 25, 26, 28], PIR protocols are still quite expensive, especially in the single-server setting that does not assume the existence of non-colluding servers. This is due in large part to a well-known fundamental barrier that limits the practical efficiency of conventional PIR schemes: The amount of server computation will be linear in the size of the database. Intuitively, a PIR scheme must ask the server to touch every entry in the database; otherwise, the server learns that the untouched entries are not what the client is looking for.

Several directions have been explored to circumvent this fundamental barrier. A promising and fruitful recent attempt has been the paradigm of *stateful* PIR, first proposed by Patel, Persiano, and Yeo [31]. In this paradigm, the client stores hints (hence called stateful) and uses these hints to speed up queries. The hints, usually in the form of parities of subsets of database entries, need to be retrieved privately. This is done in an offline phase that can be fairly expensive or may even require downloading the entire database. After an expensive offline phase, the client can make many online queries cheaply before having to rerun the offline phase. This makes the stateful PIR scheme very efficient in an amortized sense after sufficiently many queries. Although the first stateful PIR scheme still incurred linear server computation per query, the paradigm proves promising.

Corrigan-Gibbs and Kogan [10] give the first stateful PIR scheme with amortized *sublinear* server computation. Follow-up works continue to make further improvements and unlock more potential of this paradigm [9, 19, 21, 22, 32–34]. Despite the inspiring progress, existing amortized sublinear stateful PIR schemes are still faced with various challenges, including large client storage, high communication, and subpar practical efficiency. Many schemes also have to resort to heavy-weight theoretical tools [21, 32, 33], parallel repetition [9, 21, 32, 33], or restricted client query sequences [34].

This paper presents simple and practical amortized sublinear stateful PIR schemes that avoid most of the aforementioned drawbacks. Our schemes achieve constant online response overhead, take milliseconds of computation, rely only on pseudorandom functions, need no repetition, and have no restrictions on client queries. We could not address all the bottlenecks, though. Our schemes still require relatively large client storage and moderate request size, and the single-server variant still requires streaming the entire database in the offline phase.

Overview of existing amortized sublinear stateful PIR. To better explain our techniques and contributions, let us briefly go over the blueprint of amortized sublinear stateful PIR by Corrigan-Gibbs



CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690266

Table 1: Comparison with recent practical amortized sublinear stateful PIR schemes. Request size and client computation are measured in words of size λ or $\log N$. Response size, client storage, and server computation are measured in database entry size (response is thus a blowup over the insecure baseline). Major performance bottlenecks are marked in red.

Scheme	No. of Servers	Amortized communication Request	Amortized communication Response	Storage Client	Amortized computation Client	Amortized computation Server
Corrigan-Gibbs-Kogan [10]	2	$O(\lambda\sqrt{N})$	$O(\lambda)$	$O(\lambda^2\sqrt{N})$	$O(\lambda\sqrt{N})$	$O(\lambda\sqrt{N})$
Checklist [19]	2	$O(\log N)$	$O(1)$	$O(N)$	$O(\sqrt{N})$	$O(\sqrt{N})$
TreePIR [22]	2	$O(\log N)$	$O(\sqrt{N})$ [1]	$O(\lambda\sqrt{N})$	$O(\sqrt{N}\log N)$	$O(\sqrt{N}\log N)$
This paper	2	$O(\sqrt{N})$	$O(1)$	$O(\lambda\sqrt{N})$	$O(\sqrt{N})$	$O(\sqrt{N})$
Corrigan et al. [9]	1	$O(\lambda\sqrt{N})$	$O(\lambda)$	$O(\lambda^2\sqrt{N})$	$O(\lambda\sqrt{N})$	$O(\lambda\sqrt{N})$
Piano PIR [2] [34]	1	$O(\sqrt{N})$	$O(\sqrt{N}/\lambda)$	$O(\lambda\sqrt{N})$	$O(\sqrt{N})$	$O(\sqrt{N})$
This paper	1	$O(\sqrt{N})$	$O(\sqrt{N}/\lambda)$	$O(\lambda\sqrt{N})$	$O(\sqrt{N})$	$O(\sqrt{N})$

[1] TreePIR [22] can invoke an extra single-server PIR to reduce the asymptotic response overhead, but a variant without this second PIR gives better practical efficiency and makes a fairer comparison.
[2] Piano requires a stronger assumption than standard PIR, i.e., client queries must have no adversarial influence.

and Kogan [10]. The client privately retrieves hints in an offline phase. Each hint involves a subset S of $\sqrt{N}$ random distinct indices within $[0, N-1]$ where N is the number of entries in the database. For each hint, the client stores the subset S and the corresponding parity $\bigoplus_{i\in S} \mathsf{DB}[i]$ where $\mathsf{DB}[i]$ is the i-th entry of the database, and $\bigoplus$ represents XOR. In the online phase, if the client wants to retrieve the i-th entry, it finds a subset S that contains i. Since the client stores the parity of entries in S, ideally, it just needs to ask the server for the parity of entries in $S \setminus \{i\}$, from which it can easily recover $\mathsf{DB}[i]$.

However, with the above high-level strategy, the client always sends the server a subset that does not contain the queried index i. This is insecure because the server learns that the queried entry is not one of those in $S \setminus \{i\}$. To fix this problem, Corrigan-Gibbs and Kogan suggest that the client occasionally removes an index other than i from S. However, when the client does so, the client loses the ability to retrieve the queried entry i. To compensate for this loss of correctness, λ instances of their protocol are executed in parallel to achieve an exponentially small (in λ) failure probability. This blows up all efficiency metrics (communication, computation, and client storage) by a factor of λ and renders the scheme impractical.

Two schemes have been proposed [19, 22] to avoid this λ factor blowup, but both come with notable drawbacks. First, both schemes require two non-colluding servers, and there is no clear way to extend them to single-server stateful PIR. Second, both schemes make sacrifices on efficiency. The Checklist scheme by Kogan and Corrigan-Gibbs [19] requires the client to pay either $\Omega(N)$ storage or $\Omega(N)$ computation *per query*, both of which are clearly undesirable. The TreePIR scheme by Lazzaretti and Papamathou [22] incurs a response overhead of $\Theta(\sqrt{N})$ on every query. This large response overhead would be prohibitive for databases with large entries. Though this problem could be mitigated in theory by invoking another regular (i.e., not stateful) single-server PIR scheme, that would not be efficient in practice. We will explain this in more detail in Section 5.

Another closely related recent work is Piano PIR by Zhou et al. [34]. Their paper adapts the TreePIR scheme to the single-server setting but has to make an extra assumption that the client queries have *no* adversarial influence. While this extra restriction may be justifiable in certain scenarios, it is not always valid and is a departure from the standard PIR model. Section 2 discusses this issue in more detail.

Our results. In this paper, we propose new techniques in hint construction and usage to obtain simple and practical amortized sublinear stateful PIR schemes. Our new hint system eliminates the aforementioned leakage associated with removing the queried index, thus obviating the need for parallel repetition. We achieve constant online response—just twice that of simply fetching the desired entry without privacy—as well as sublinear client storage and sublinear client computation. Our method works for both the two-server and the single-server settings. The two-server version further achieves constant amortized response overhead, while the single-server version has $O(\sqrt{N}/\lambda)$ amortized response overhead due to the need to stream the entire database in the offline phase.

Table 1 compares recent practical amortized sublinear PIR schemes in terms of asymptotic efficiency. We exclude schemes that rely on heavy theoretical tools, such as those based on oblivious locally decodable codes [5, 7, 23] or privately puncturable/programmable pseudorandom functions [21, 32, 33]. The three major performance bottlenecks in prior works are marked in red: λ factor repetition, $\Theta(\sqrt{N})$ response overhead, and linear client storage. Our schemes avoid all three bottlenecks. Compared with Piano PIR, our scheme has the same asymptotic efficiency but has better concrete efficiency and places no restriction on client queries.

Our scheme enjoys good concrete efficiency. Take for example an 8 GB database consisting of 2^{28} entries where each entry is 32 bytes. Our two-server scheme requires 60 MB of client storage, and consumes 34 KB of communication and 2.7 milliseconds of computation per query. In comparison, existing two-server schemes require either over 1 GB of client storage or over 1 MB of communication.

For the same database, our single-server scheme requires 100 MB of client storage, and consumes 47 KB of communication and 4.5 milliseconds of computation, *amortized* per query. In comparison, the best prior scheme achieving the standard PIR correctness would

be at least two orders of magnitude more expensive. Compared with Piano PIR [34], our protocol offers up to 43% reduction in client storage, up to 53% reduction in amortized communication, and up to 73% reduction in amortized computation, in addition to the stronger and standard PIR correctness.

2 Model and Preliminary

Private Information Retrieval (PIR). Given a database DB of N entries and a query index i, the client wants to privately retrieve the i-th entry in the database. A PIR protocol should satisfy the following two properties.

- **Correctness:** If the client and the server correctly execute the protocol, then the client retrieves the queried entry.
- **Privacy:** The server learns *nothing* about the client's query index.

The privacy requirement of PIR can be more rigorously captured by a game between the server, who is also the adversary, and the client. The game resembles the standard message indistinguishability game for encryption.

(1) The server picks two indices i and i', and send them to the client.
(2) The client flips a coin $b \leftarrow \{0, 1\}$. The client queries index i if $b = 0$ and queries index i' if $b = 1$.
(3) The server tries to guess b.

If the server can guess b correctly with $0.5 + \epsilon$ probability where ϵ is non-negligible, then the PIR protocol is insecure.

Stateful PIR. We now extend the above PIR definition with a single query to a stateful PIR that deals with a sequence of queries. Given a database DB of N entries and a sequence of query indices $\mathbf{I} = [i_1, i_2, i_3, \ldots]$, the client makes any (polynomial) number of queries one by one. A stateful PIR protocol should satisfy the following two properties.

- **Correctness:** If the client and the server correctly execute the protocol, then the client retrieves the i_j-th entry in the database at the end of the j-th query.
- **Privacy:** The server learns *nothing* about the client's sequence of query indices.

Similarly, the privacy requirement of stateful PIR can be more rigorously captured by a game between the client and the server.

(1) The server picks two sequences of query indices $\mathbf{I}$ and $\mathbf{I}'$ of equal length and send them to the client.
(2) The client flips a coin $b \leftarrow \{0, 1\}$. The client queries sequence $\mathbf{I}$ if $b = 0$ and queries index $\mathbf{I}'$ if $b = 1$.
(3) The server tries to guess b.

If the server can guess b correctly with $0.5 + \epsilon$ probability where ϵ is non-negligible, then the stateful PIR protocol is insecure.

Note that we let the server choose the two sequences of query indices it wants to distinguish, similar to the indistinguishability game for encryption. Likewise, correctness should also hold for any query sequence, including ones chosen by the adversary. We could make the server (adversary) even more powerful by letting it choose the query sequences *adaptively*, i.e., it can choose the next pair of query indices *after* interacting with the client for the previous query in the sequence. Likewise, correctness can also be stated for any adaptively constructed sequence. Most existing stateful PIR schemes, including ours, are correct and secure even for adaptively constructed sequences of queries.

Importance of supporting arbitrary query sequences. Piano PIR [34], however, does not satisfy the above definition (even for statically constructed queries) because it requires the client query sequence to have no adversarial influence. This assumption may be justifiable in some use cases but not always. It is not hard to conceive scenarios in which the client's query sequence is influenced by a malicious third party (who is different from the honest but curious server). As a concrete example, consider DNS lookup, which is a primary application that Piano PIR targets. The threat model of DNS typically assumes that the client may visit a malicious webpage that can trigger DNS queries of the adversary's choosing, e.g., as in the Kaminsky attack. This immediately breaks the assumption of no adversarial influence on the client queries. Such an adversary can easily make Piano PIR fail in correctness (see Section 5). If an upper-level application behaves differently when a PIR query succeeds vs. fails, a correctness failure can lead to a privacy violation.

Pseudorandom functions. We assume the server is computationally bounded. We will make use of pseudorandom functions (PRF). PRF is one of the most common cryptographic primitives and can be instantiated from any one-way function, including the standardized and widely used AES block cipher and SHA cryptographic hash functions. A PRF takes a secret key and an input. For convenience, we will omit writing the secret key since there should be no confusion in our schemes that the client holds the secret key (and shares the secret key with one of the servers in the two-server setting). The input to the PRF is often a concatenation of multiple values. For example, a PRF call in our algorithms will be written as $\mathsf{PRF}(x \parallel y \parallel z)$.

3 Algorithms

3.1 Overview

The key idea is a new type of hint that eliminates the information leakage due to the absence of the queried index. This immediately obviates the need for parallel repetition because there will be no (non-negligible) correctness failure. Our techniques can be applied to the original sublinear scheme of Corrigan-Gibbs and Kogan [10] or the partition-based hints of TreePIR [22]. We will describe our techniques on top of the partition-based hints because they offer advantages in compact hint storage and fast membership testing. In this context, our techniques help avoid the large responses and the need for non-colluding servers in TreePIR's partition paradigm.

Construction of our hints. A database of size N is divided into $\sqrt{N}$ partitions each of size $\sqrt{N}$. For convenience, we assume $\sqrt{N}$ is an even integer. The database can always be padded to the square of the next even integer with very small extra overhead.

Let $\mathcal{R}$ denote the following distribution: first, select $\sqrt{N}/2 + 1$ random distinct partitions (i.e., sample without replacement) out of the $\sqrt{N}$ total partitions; then pick one random index from each of these $\sqrt{N}/2 + 1$ partitions. Each hint in our algorithm consists of a sample from $\mathcal{R}$ and its corresponding parity. Hence, a hint contains $\sqrt{N}/2 + 1$ random indices from $\sqrt{N}/2 + 1$ random partitions, one

index per partition. Note that the number of partitions we select in a hint is exactly one more than half of the total number of partitions.

For correctness, the client needs to store M hints (M will be specified later). For each $j = 0, 1, 2, \ldots, M-1$, the client samples $S_j \leftarrow \mathcal{R}$ and stores S_j along with $\sum_{i \in S_j} \mathrm{DB}[i]$ as one hint.

Usage of the hint also resembles previous works in principle. When the client makes a query to the i-th entry of the database, the client looks for a hint whose subset S_j contains index i. The client sends $S_j \setminus \{i\}$ to the server. We will call it the query subset. The server returns the parity for $S_j \setminus \{i\}$. The client easily recovers $\mathrm{DB}[i]$ since it has been storing the parity for S_j. We need $M = \lambda\sqrt{N}$ where λ is a security parameter so that a subset containing the queried index can be found with all but exponentially small (in λ) probability.

Eliminating the leakage. We now tackle the main challenge mentioned in Section 1. With the approach described so far, the query subset sent by the client involves $\sqrt{N}/2$ random partitions and contains one random index from each of them. However, since the client always removes the queried index, the query subset will not contain any index from the partition to which the queried entry belongs. Thus, the server learns that the queried entry is definitely *not* in any of the $\sqrt{N}/2$ partitions in the query subset.

Our main idea to address this leakage is for the client to additionally send a dummy subset of indices. The dummy subset contains one random index from each of the $\sqrt{N}/2$ partitions that do not appear in the query subset. The client also randomly swaps the two subsets. I.e., with half probability, the client sends the query subset followed by the dummy subset, and with the other half probability, the dummy subset followed by the query subset. This way, the server cannot distinguish between the query subset and the dummy subset.

Note that this dummy subset is precisely what is needed to eliminate the aforementioned leakage: the two subsets together cover every partition! We next give a proof sketch. The full security proof is given in Section 3.5. The client sends two subsets that together cover all $\sqrt{N}$ partitions. A random index is picked from each partition, so we only need to show that the grouping of the partitions leaks no information. The grouping of the partitions is indistinguishable from a random arrangement. The dummy subset bundles the partition of interest with $\sqrt{N}/2-1$ other random partitions, and the query subset covers the remaining $\sqrt{N}/2$ partitions. A purely random arrangement would anyway group the partition of interest with $\sqrt{N}/2-1$ other random partitions and leave the rest as the other group.

The online phase. The online phase of our stateful PIR protocol follows naturally from the above hint system. Upon an input query index i, the client finds a hint whose subset contains the query index i. The client removes i from the subset (this is the real query subset). The client then constructs a dummy subset that consists of one random index from each partition not represented in the real query subset. The client now sends the two subsets, permuted, to the server. Figure 1 illustrates this process.

The server returns the two parities corresponding to the two subsets. The client discards the dummy parity and uses the parity of the real query subset to recover the desired entry. As a result, the response overhead of our scheme is close to optimal: only twice that of simply fetching the desired entry without privacy.

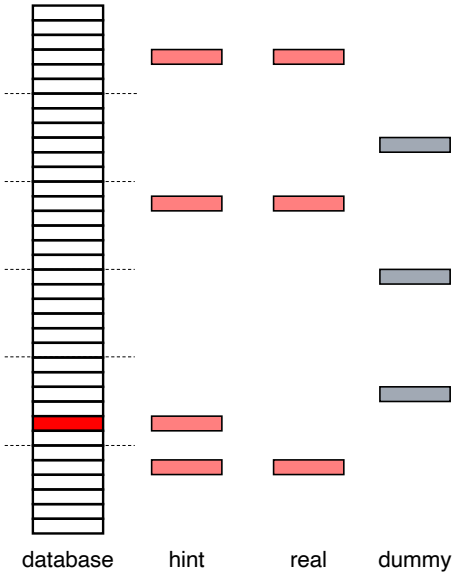


Figure 1: An illustration of the hint system and the client's request. The database has $N = 36$ entries and is divided into $\sqrt{N} = 6$ partitions. Each hint selects $\sqrt{N}/2+1 = 4$ random partitions and picks a random index from each. The queried index is removed to produce the real query subset. A dummy subset is constructed by picking one random index from each of the remaining three partitions.

Hint replenishment and the offline phase. After each query in the online phase, the client needs to replenish one hint since it has just consumed one. The replenished hint must follow the same distribution as the one just consumed, i.e., contains index i in the subset. How we carry out the hint replenishment and how we run the offline phase depend on whether we have a single server or two non-colluding servers. We will describe the two variants in detail later in the section.

3.2 Details of Hints and Online Phase

Sampling a subset of exact size. A step that warrants more clarification is how we sample a subset of size exactly $\sqrt{N}/2+1$ out of the $\sqrt{N}$ partitions. For reasons that will become clear later and involve hint replenishment, we will sample a subset of size exactly $\sqrt{N}/2$ and then supply one extra index.

To sample exactly half of the partitions, we compute a pseudorandom value using PRF for each hint-partition combination, i.e.,

$$v_{j,k} = \mathrm{PRF}(\text{“select”} \parallel j \parallel k)$$

for the k-th partition of a hint with ID j. The prefix "select" is added because we later need another pseudorandom value for each hint-partition combination.

For each hint j, compute $\mathbf{V}_j = [v_{j,0}, v_{j,1}, v_{j,2}, \ldots, v_{j,\sqrt{N}-1}]$. We then find a cutoff value $\hat{v}_j$ to divide $\mathbf{V}_j$ into two equal-sized halves, i.e., exactly $\sqrt{N}/2$ elements in $\mathbf{V}_j$ are smaller than $\hat{v}_j$ and exactly $\sqrt{N}/2$ elements in $\mathbf{V}_j$ are larger than $\hat{v}_j$. A natural choice of $\hat{v}_j$

is the median of $\mathbf{V}_j$. Since we assume $\sqrt{N}$ is an even integer, the median is the average of two elements in $\mathbf{V}_j$. We save this cutoff median value alongside the hint ID for each hint. This will give us an efficient method to check if a partition is selected by a hint, using $O(1)$ time and $O(1)$ client storage per hint.

Next, we need to pick one more index from a random partition among the $\sqrt{N}/2$ unselected partitions. An easy and effective way to do so is to keep picking random indices and checking if the corresponding partition has already been selected, until hitting an unselected partition.

Hint storage. Each hint is stored as a tuple $(j, \hat{v}_j, e_j, P_j)$ where j is a unique hint ID, $\hat{v}_j$ is the cutoff median value, e_j is the extra index, and P_j is the parity.

With the hint construction and storage details in place, we can now give more details of the algorithm for the online phase, shown in Algorithm 2.

Finding a suitable hint. Upon input query index i, the client computes $\ell = \lfloor i/\sqrt{N} \rfloor$, which is the partition that index i belongs to. The client then goes through the hints to find one whose subset contains i. There are two cases a hint's subset contains i. A straightforward case is that the extra index e_j equals i. The other case is when partition ℓ is selected and index i is picked from partition ℓ for that hint. For each hint j, the client computes $v_{j,\ell}$ and checks if $v_{j,\ell}$ is smaller than the median cutoff $\hat{v}_j$. If so, the client further computes a pseudorandom offset for the partition,

$$r_{j,\ell} = \mathsf{PRF}(\text{“offset”} \parallel j \parallel \ell),$$

and checks if $r_{j,\ell} = i \mod \sqrt{N}$. If both checks pass, or if the extra index $e_j = i$, then index i is included in the subset of hint j.

We remark that this step showcases the benefit of partition-based hints: the partitioning allows us to test in $O(1)$ time whether a hint's subset includes a particular index, as we only need to check the corresponding partition.

Constructing and encoding the two subsets. After finding a hint that contains the query index i, the client reconstructs the hint's subset and then removes the query index i to obtain the query subset. The client also constructs a dummy subset that contains one random index from each partition that is not in the query subset. Note that a random index (possibly i) will be drawn from the partition of interest. This index will certainly be part of the dummy subset, but the server cannot tell which subset is the dummy one once the client permutes the two subsets.

Although we write our pseudocode to send two subsets for simplicity, there is an equivalent and more compact way to encode the two subsets. We can use a bit vector $\mathbf{b} = [b_0, b_1, \ldots, b_{\sqrt{N}-1}]$ and an offset vector $\mathbf{r} = [r_0, r_1, \ldots, r_{\sqrt{N}-1}]$. The offset vector encodes which index is picked from each partition. Concretely, $s_k = r_k + k\sqrt{N}$ is the index picked from partition k. The bit vector encodes whether each partition is part of the first or the second subset. In other words, let S_0 and S_1 denote the two subsets of indices that the client would have sent in the pseudocode of Algorithm 2. Then, $s_k \in S_0$ if $b_k = 0$ and $s_k \in S_1$ if $b_k = 1$. We note again that the two subsets are permuted by the client, so the real subset may be either S_0 or S_1, with half-half probability.

It is not hard to see that this encoding is equivalent to sending S_0 and S_1 as done in the pseudocode of Algorithm 2, but is slightly more efficient. Sending S_0 and S_1 directly would cost $\sqrt{N} \log N$ bits. The encoding using $\mathbf{b}$ and $\mathbf{r}$ costs $\sqrt{N} + \sqrt{N} \log \sqrt{N} = (\sqrt{N}/2 + 1) \log N$ bits, roughly reducing the client's request size by half.

3.3 The Two-Server Scheme

When there are two non-colluding servers, we use one server for the offline phase and hint replenishment and the other for the online queries. For convenience, we call the two servers the *offline server* and the *online server*, even though the offline server also helps with the hint replenishment step during the online phase.

The offline phase only needs to run once at the beginning of the entire protocol to help the client start with sufficiently many hints. After that, the client invokes hint replenishment on the fly at the end of each online query. The pseudocode of the complete two-server stateful PIR protocol is given in Algorithms 1, 2, and 3.

The offline phase. The offline phase is shown in Algorithm 1 and is fairly straightforward. The client initiates the offline phase by sending its PRF evaluation key to the offline server. This allows the offline server to fully construct the hints. For each hint, exactly half of the partitions are selected using the cutoff median method described in Section 3.2, and a random index is picked from each selected partition. After that, an extra index is picked from a partition that has not been selected yet. The offline server can easily compute the parity of these entries. Lastly, the offline server sends the cutoff, the extra index, and the parity for each hint to the client. This completes the offline phase.

Hint replenishment. To replenish a hint after querying index i, the client asks the offline server to run Algorithm 3. Since the offline server has the PRF evaluation key, it can construct a new hint using the next available hint ID, similar to what it did in the offline phase. But there are two new catches. First, the offline server does not add the extra index because the client needs to add index i, the index that is just queried, so that the replenished hint follows the same distribution as the consumed one, i.e., has i in the subset. Second, we do not want the offline server to learn the new hint's subset, because that would reveal some information to the offline server about the query the client just made. Therefore, we let the offline server compute the parities of both halves and send both parities to the client along with the new hint ID and median cutoff.

Upon receiving the above from the offline server, the client chooses the half that does *not* select the partition of i as the real half. To do so, the client may have to redefine the operator $<$ for this hint. In other words, the client stores a bit that indicates whether this hint chooses all partitions whose pseudorandom values ($v_{J,k}$) are smaller or larger than the median cutoff. This essentially permutes the two halves and makes them indistinguishable to the offline server. The client then adds index i to the hint's subset as the extra index and adds $\mathsf{DB}[i]$ (which the client has just retrieved) to the parity. The new hint is now fully constructed and replaces the consumed hint.

Algorithm 1 The offline algorithm with two non-colluding servers, run by the offline server

for $j = 0, 1, 2, \ldots, M-1$ **do**
  Initializes parity $P_j = 0$
  Compute $\mathbf{V}_j = [v_{j,0}, v_{j,1}, v_{j,2}, \ldots, v_{j,\sqrt{N}}]$ where $v_{j,k} = \mathsf{PRF}(\text{“select”} \parallel j \parallel k)$
  Find the median $\hat{v}_j$ of $\mathbf{V}_j$ as the cutoff for selection
  $S = \{k \mid v_{j,k} < \hat{v}_j\}$ ▷ the set of partitions selected by this hint
  $P_j = \bigoplus_{k \in S} \mathsf{DB}[r_{j,k} + k\sqrt{N}]$ where $r_{j,k} = \mathsf{PRF}(\text{“offset”} \parallel j \parallel k)$ ▷ one random index per selected partition
  Set the extra index e_j to a random index from a random partition not in S
  $P_j = P_j \oplus \mathsf{DB}[e_j]$
  Send $(j, \hat{v}_j, e_j, P_j)$ to the client to be stored
end for
Set $J = M$, the next available hint ID ▷ J will be strictly increasing

Algorithm 2 The online algorithm, run by the client

Input: queried index i ▷ $v_{j,k}, r_{j,k}, h_j, \hat{v}_j, e_j, P_j$ as defined in Algorithm 1 or 4
$\ell = \lfloor i/\sqrt{N} \rfloor$ ▷ ℓ is the partition that i belongs to
Find main hint j such that $v_{j,\ell} < \hat{v}_j$ and $r_{j,\ell} == i \mod \sqrt{N}$ ▷ hint j contains i
Initialize $S = \emptyset$ and $S' = \emptyset$ ▷ S will be the real subset and S' will be the dummy subset
for $k = 0 : \sqrt{N} - 1$ **do**
  if $v_{j,k} < \hat{v}_j$ **then**
    $S = S \cup \{r_{j,k} + k\sqrt{N}\}$
  else if e_j belongs to partition k **then**
    $S = S \cup \{e_j\}$
  else
    $S' = S' \cup \{\mathsf{rand}() + k\sqrt{N}\}$ ▷ add a random index from partition k to the dummy subset
  end if
end for
$S = S \setminus \{i\}$ ▷ remove the queried index from the real subset
$S' = S' \cup \{\mathsf{rand}() + \ell\sqrt{N}\}$ ▷ add a random index from partition ℓ to the dummy subset
Send (S, S') or (S', S) to the server with half-half probability ▷ permute the real and dummy subsets
Receive the two subset parities P and P' from the server ▷ in the order S and S' are sent
Return $P \oplus P_j$ as $\mathsf{DB}[i]$
Replenish a hint that contains index i from partition ℓ using Algorithm 3 or 5

Algorithm 3 The hint replenish algorithm with two non-colluding servers, run by the offline server and the client

Use the next available hint ID J ▷ The client asks the offline server to start hint replenishment
Initializes parity $P_J = P'_J = 0$
Compute $\mathbf{V}_J = [v_{J,0}, v_{J,1}, v_{J,2}, \ldots, v_{J,\sqrt{N}-1}]$
Find the median $\hat{v}_J$ of $\mathbf{V}_J$
$S = \{k \mid v_{J,k} < \hat{v}_J\}$
$P_J = \bigoplus_{k \in S} \mathsf{DB}[r_{J,k} + k\sqrt{N}]$ ▷ recall $r_{j,k} = \mathsf{PRF}(\text{“offset”} \parallel j \parallel k)$
$P'_J = \bigoplus_{k \notin S} \mathsf{DB}[r_{J,k} + k\sqrt{N}]$
Send $J, \hat{v}_J, P_J, P'_J$ to the client ▷ The rest of the algorithm is run by the client
if $v_{J,\ell} < \hat{v}_J$ **then** ▷ pick the half that does not select partition ℓ
  $P_J = P'_J$
  Set a bit to redefine $<$ to be “greater than” for this hint ▷ Algorithm 2 should check this bit and interpret $<$ accordingly for each hint, but we omitted these details in Algorithm 2 for readability of the pseudocode
end if
Replace hint j with new hint $(J, \hat{v}_J, i, P_J \oplus \mathsf{DB}[i])$ ▷ add i as the extra index to the new hint J

Algorithm 4 The streaming offline algorithm with a single server, run by the client

for $j = 0, 1, 2, \ldots, 1.5M - 1$ **do** ▷ M main hints and $0.5M$ pairs of backup hints
  Initialize parity $P_j = 0$, and additionally initialize $P'_j = 0$ if $j \geq M$ ▷ backup hints come in pairs
  Compute $\mathbf{V}_j = [v_{j,0}, v_{j,1}, v_{j,2}, \ldots, v_{j,\sqrt{N}-1}]$ where $v_{j,k} = \mathsf{PRF}(\text{“select”} \parallel j \parallel k)$
  Find and store the median $\hat{v}_j$ of $\mathbf{V}_j$ as the cutoff for partition selection
  if $j < M$ **then** ▷ main hints
    Set the extra index e_j to a random index from a random partition not in $\{k \mid v_{j,k} < \hat{v}_j\}$
  end if
end for
for $k = 0 : \sqrt{N} - 1$ **do**
  Download $\mathsf{DB}[k\sqrt{N} : (k+1)\sqrt{N} - 1]$ from the server ▷ download partition k
  for $j = 0, 1, 2, \ldots, 1.5M - 1$ **do**
    $x = \mathsf{DB}[r_{j,k} + k\sqrt{N}]$ where $r_{j,k} = \mathsf{PRF}(\text{“offset”} \parallel j \parallel k)$ ▷ a pseudorandom entry is picked from partition k
    if $v_{j,k} < \hat{v}_j$ **then** ▷ partition k is selected by hint j
      $P_j = P_j \oplus x$
    else if $j \geq M$ **then**
      $P'_j = P'_j \oplus x$ ▷ also construct the backup hint in the pair
    end if
    if $\lfloor e_j / \sqrt{N} \rfloor == k$ **then** ▷ the extra index e_j is in partition k
      $P_j = P_j \oplus \mathsf{DB}[e_j]$
    end if
  end for
end for

Algorithm 5 The hint replenish algorithm with a single server, run by the client

Let J be the ID of the next unused pair of backup hints
if $v_{J,\ell} > \hat{v}_J$ **then** ▷ pick the half that does not select partition ℓ
  $P_J = P'_J$
  Set a bit to redefine $<$ to be “greater than” for this hint ▷ Algorithm 2 checks this bit to interpret $<$
end if
$h_j = J$
$e_j = i$
$P_j = P_J \oplus \mathsf{DB}[i]$
Replace hint j with backup hint $(J, \hat{v}_J, i, P_J \oplus \mathsf{DB}[i])$ ▷ add i as the extra index to the new main hint J

3.4 The Single-Server Scheme

Hint replenishment using backup hints. With a single server, we no longer have the luxury of replenishing a hint on the fly. Instead, we will use the idea of backup hints from [9]. The client retrieves additional backup hints in the offline phase so that the client can replenish a hint during the online phase without contacting the server. Since backup hints will eventually run out, the offline phase needs to be run periodically. The pseudocode of our complete single-server stateful PIR protocol is given in Algorithms 4, 2, and 5.

In the offline phase, the client retrieves not only the $\lambda\sqrt{N}$ primary hints but also $\lambda\sqrt{N}$ backup hints. A backup hint does not have the extra index and thus contains one fewer index in its subset than a main hint. After the client makes a PIR query for index i, it finds a backup hint that does not select i's partition. The client then adds index i to the subset as the extra index and adds $\mathsf{DB}[i]$ to the parity. The new subset and parity now form a regular main hint that follows the same distribution as the consumed one, i.e., has i in the subset.

A simple strategy is to have $\lambda\sqrt{N}$ independent backup hints. Then, there are in expectation $0.5\lambda\sqrt{N}$ backup hints that skip any given partition. So the client can make close to, but fewer than, $0.5\lambda\sqrt{N}$ (say $0.4\lambda\sqrt{N}$) online queries before having to rerun the offline phase. Even if the client keeps querying entries from the same partition, it will not run out of backup hints that skip that partition, except for exponentially small (in $\lambda\sqrt{N}$) probability.

A more clever strategy is to have backup hints in pairs, similar in spirit to the two-server hint replenishment algorithm. This is the strategy taken in the pseudocode of Algorithm 5. From a backup hint ID J, the client computes $\mathbf{V}_J$ as well as the cutoff $\hat{v}_J$. The cutoff $\hat{v}_J$ divides the partitions into two equal-sized halves. The client will store the parities of both halves. When it is time to replenish a hint that contains index i, the client picks the half that does *not* select the partition ℓ that index i belongs to, and then adds i as the extra index. Like the two-server scheme, the client needs to store a bit indicating whether $<$ is redefined to be “greater than” for this hint. This way, the client only needs one pair of backup hints per query, as one of the two halves will certainly meet the need. The client

can now store $\lambda\sqrt{N}/2$ pairs of backup hints and can make exactly $\lambda\sqrt{N}/2$ online queries before having to rerun the offline phase.

Offline phase. In the offline phase, the client needs to retrieve main hints and backup hints in a private manner. This can be done in a few ways. The simplest and most practical way is perhaps to stream the entire database, one partition at a time. The pseudocode of the streaming offline phase is given in Algorithms 4. The extra index of each main hint can be sampled in the same way described in Section 3.2: keep picking a random partition and checking if it is already selected. This is now done by the client prior to streaming the database. After downloading a partition, it is straightforward to use $v_{j,k}$ and $r_{j,k}$ to determine, for each main or backup hint j, which index, if any, should be drawn from the current partition k. For each main hint, the client also checks if its extra index is from the current partition. For each backup hint pair, the client updates the parity corresponding to the correct half based on whether $v_{j,k}$ is smaller or larger than the median cutoff.

3.5 Correctness and Privacy Analysis

We will first focus on the very first query after the offline phase and then extend the analysis to subsequent queries.

Correctness. For correctness, we need to prove that, upon an input query index i, the client will be able to find, with overwhelming probability, a hint whose subset includes i. To this end, we first observe the following simple fact.

Lemma 1. *Each hint in our construction has at least $\frac{1}{2\sqrt{N}}$ probability of containing a particular index.*

Proof. A hint contains a particular index i if the hint selects i's partition and picks i from that partition. The former happens with $(\sqrt{N}/2+1)/\sqrt{N} > 1/2$ probability (the plus one is due to the extra index), and the latter happens with $1/\sqrt{N}$ probability. □

For correctness to be violated, none of the $\lambda\sqrt{N}$ main hints contains the query index. This happens with less than $(1-\frac{1}{2\sqrt{N}})^{\lambda\sqrt{N}} < e^{-\lambda/2}$ probability. For a sufficiently large λ, this probability is astronomically small.

Privacy. We need to prove that the two subsets sent by the client reveal no information about the query index. We will carry out the proof as if the PRF is perfectly random. The privacy of our PIR protocol is then reduced to the pseudorandomness of the PRF.

It is more convenient to reason about privacy with the more compact encoding described in Section 3.2. Recall that the client sends a bit vector $\mathbf{b}$ grouping partitions into two subsets along with an offset vector $\mathbf{r}$ encoding the index picked from each partition. First, observe that the offset vector $\mathbf{r}$ consists of pseudorandom values that are independent of the query index.

- For partitions not selected by the hint, a fresh pseudorandom dummy offset is used (Line 11 of Algorithm 2).
- For the partition that contains the query index i, i is removed and is replaced with a fresh pseudorandom dummy offset (Line 15 of Algorithm 2).
- For the remaining partitions that are selected by the hint, the offsets are picked pseudorandomly during the offline phase, and this is the first (and only) time they are revealed to the (online) server.

Thus, from the (online) server's perspective, all $\sqrt{N}$ offsets are fresh, pseudorandom, and independent of the query index.

The crux of the proof is to show that the bit vector $\mathbf{b}$ reveals no information about the query index. Formally, we will prove that the distribution of $\mathbf{b}$ is not affected by, and hence reveals no information about, the query index.

Lemma 2. *For any two query indices i and i', $\Pr(\mathbf{b} \mid i) = \Pr(\mathbf{b} \mid i')$.*

Proof. Let ℓ denote the partition index i belongs to and ℓ' denote the partition index i' belongs to. When the query index is i, an index from partition ℓ is added to the dummy subset. For the client to send $\mathbf{b}$, two events must happen. First, the bit b_ℓ represents the dummy subset (as opposed to the opposite bit $1-b_\ell$). This happens with $1/2$ probability. Second, besides partition ℓ, the set of partitions selected by this hint are those marked by the opposite bit, i.e., $T = \{k \mid b_k \neq b_\ell\}$. Since each hint selects $\sqrt{N}/2+1$ partitions at random, the probability for the other $\sqrt{N}/2$ selected partitions to be those in T is $\tau = \binom{\sqrt{N}-1}{\sqrt{N}/2}^{-1}$. These two events are independent, so $\Pr(\mathbf{b} \mid i) = \tau/2$. By the exact same argument, we have

$$\Pr(\mathbf{b} \mid i') = \tau/2 = \Pr(\mathbf{b} \mid i). \qquad \square$$

Lemma 2 is sufficient to establish the privacy of our protocol. But to make things more explicit, we can derive the following simple facts from Lemma 2.

$$\Pr(\mathbf{b}) = \sum_i \Pr(\mathbf{b} \mid i) \cdot \Pr(i) = \tau/2 \cdot \sum_i \Pr(i) = \tau/2.$$

Thus, for all query index i,

$$\Pr(i \mid \mathbf{b}) = \frac{\Pr(i, \mathbf{b})}{\Pr(\mathbf{b})} = \frac{\Pr(\mathbf{b} \mid i) \cdot \Pr(i)}{\Pr(\mathbf{b})} = \Pr(i).$$

The fact that $\Pr(i \mid \mathbf{b}) = \Pr(i)$ for all i means that observing $\mathbf{b}$ does not change an observer's prior on the query index, which is to say $\mathbf{b}$ does not reveal any information about the query index. Therefore, the server will have no advantage in distinguishing the two queries in the privacy game.

Extension to subsequent queries. The above completes the correctness and privacy proofs for the first query after the offline phase. Next, we extend the proofs to subsequent queries. For this step, we need to show that after a query consumes and replenishes a hint, the distribution of the main hints remains the same. Then, our privacy proof above would apply directly to all subsequent queries, and the correctness failure probability over a sequence of queries can be upper bounded by a simple union bound.

Let $H_{j,k}$ be the random variable representing the index picked from partition k in hint j. If hint j does not select from partition k, $H_{j,k} = \perp$. Then, the following matrix of random variables fully describes the main hints.

$$\mathbf{H} = \begin{bmatrix} \overrightarrow{H_0} \\ \overrightarrow{H_1} \\ \vdots \\ \overrightarrow{H_{M-1}} \end{bmatrix} = \begin{bmatrix} H_{0,0} & H_{0,1} & \dots & H_{0,\sqrt{N}-1} \\ H_{1,0} & H_{1,1} & \dots & H_{1,\sqrt{N}-1} \\ \vdots & \vdots & \ddots & \vdots \\ H_{M-1,0} & H_{M-1,1} & \dots & H_{M-1,\sqrt{N}-1} \end{bmatrix}$$

Let $\mathbf{H}$ represent the main hints before the current query and $\mathbf{H}'$ represent the main hints after the current query. We want to show that $\mathbf{H}'$ and $\mathbf{H}$ are identically distributed.

Each hint (row vector) in $\mathbf{H}$ is drawn from the distribution $\mathcal{R}$ described in Section 3.1. Let $\mathcal{R}_i$ be the distribution of a hint *conditioned on* the event that it contains index i. Let $\mathcal{R}_{-i}$ be the distribution of a hint *conditioned on* the event that it *does not* contains index i.

Suppose we scan the main hints from 0 to $M-1$ to look for the query index i. Each hint independently has a probability $q = \frac{\sqrt{N}/2+1}{\sqrt{N}} \cdot \frac{1}{\sqrt{N}}$ to contain i: partition ℓ needs to be selected and i needs to be picked from partition ℓ. Let J be the hint consumed. J follows a geometric distribution with parameter q. (The event that no hint contains i is a negligible one, and for convenience, we can assume no hint is consumed or replenished in that case.) Thus,

$$\Pr(J > j) = (1-q)^{j+1},$$
$$\Pr(J = j) = (1-q)^j q,$$
$$\Pr(J < j) = \sum_{l=0}^{j-1} (1-q)^l q = 1 - (1-q)^j.$$

Both the consumed and the replenished hints follow distribution $\mathcal{R}_i$. All the other hints are unmodified. Moreover, all the hints prior to J follow distribution $\mathcal{R}_{-i}$, and all the hints after J follow distribution $\mathcal{R}$.

Let us now focus on any particular hint j in $\mathbf{H}'$. Given the distribution of J, we can think of hint j in $\mathbf{H}'$ to be sampled in the following manner: with $1-(1-q)^j$ probability, sample from $\mathcal{R}$; for the remaining $(1-q)^j$ probability, sample from $\mathcal{R}_i$ with probability q and sample from $\mathcal{R}_{-i}$ with probability $(1-q)^{j+1}$.

Observe that the q vs. $1-q$ ratio is exactly the likelihood that an original hint in $\mathbf{H}$ does vs. does not contain index i, or equivalently, follows $\mathcal{R}_i$ vs. $\mathcal{R}_{-i}$. Thus, every hint j in $\mathbf{H}'$ follows the same distribution as the hint j in $\mathbf{H}$. This shows that the main hints after a query are identically distributed as they were before the query. Then, by transitivity, the main hints at any point are identically distributed as their original states right after the offline phase. Therefore, our correctness and privacy proofs apply to all subsequent queries.

3.6 Efficiency Analysis

The two-server scheme. The offline phase costs $O(\lambda\sqrt{N})$ communication and $O(\lambda N)$ computation at the offline server. But because the offline phase runs only once, these costs do not factor into the amortized costs after sufficiently many queries. Hence, the amortized cost of our two-server scheme only depends on the online phase and the hint replenishment step. The online request size is $(\sqrt{N}/2+1)\log N$ bits using the compact encoding of the two subsets. The online response overhead is $O(1)$, or 4× to be precise, since the online server and the offline server both send back two parities.

The expected computation cost of the client is $O(\sqrt{N})$ due to searching for a hint and reconstructing the hint's subset. Because each hint has at least $\frac{1}{2\sqrt{N}}$ probability of containing a particular index by Lemma 1, the client will find a suitable hint after checking $2\sqrt{N}$ hints in expectation (and each check takes $O(1)$ time). The computation cost of the server is $O(\sqrt{N})$ due to computing the parities. These give the two-server results in Table 1.

The single-server scheme. The online phase is very similar to the two-server scheme: each online query costs $O(\sqrt{N})$ bits in request, $O(1)$ overhead in response, $O(\sqrt{N})$ client computation, and $O(\sqrt{N})$ server computation. The streaming offline phase costs N communication and $O(\lambda N)$ computation, and needs to be run every $0.5\lambda\sqrt{N}$ online queries. This leads to the single-server results in Table 1. The only difference from the two-server case is that the response overhead is $O(\sqrt{N}/\lambda)$ because the $O(N)$ offline communication is amortized over $0.5\lambda\sqrt{N}$ online queries.

4 Evaluation

4.1 Implementation Details

We implemented our scheme in C++. The implementation is available at https://github.com/renling/S3PIR/. Due to the simplicity of our schemes, the two-server version of our implementation comprises about 600 lines of code and the single-server version comprises about 500 lines of code. We set the parameter λ to 80. We use AES as the pseudorandom function. We use CryptoPP's implementation of AES, which leverages Intel's AES-NI instructions. We break up a single 128-bit AES output into four to eight pseudorandom numbers (i.e., $v_{j,k}$ and $r_{j,k}$ in the algorithms) across different hints or partitions to save computation.

We use 32-bit numbers for elements in $\mathbf{V}_j$ to save client storage and computation. It is worth noting that this gives rise to a corner case where two or more elements in $\mathbf{V}_j$ are equal to the median. When this happens, the median alone does not give a way to evenly divide $\mathbf{V}_j$ into two equal-sized halves. We could add additional metadata to handle this corner case, but because this corner case happens with a very small probability, we simply consider such a hint invalid and discard it. We have omitted the handling of this corner case from the pseudocode for readability.

The median finding procedure, if implemented naïvely, would be the bottleneck of the single-server scheme's offline phase. We could directly use introselect [30] or a similar linear time selection algorithm. But we can take advantage of the fact that elements of $\mathbf{V}_j$ are uniformly random. We can filter out elements that are too large or too small, i.e., outside two heuristic bounds, and run introselect on the reduced array. We keep count of the number of filtered elements. If we filter out X small elements, we use introselect to find the $(\sqrt{N}/2-X-1)$-th and $(\sqrt{N}/2-X)$-th smallest elements among the remaining elements. These will be the two middle elements that give the median of $\mathbf{V}_j$. With appropriately selected bounds, the probability of filtering out one of these two elements is very small. (And when that happens, we simply consider this hint invalid and discard it.) We think of the random values as 32-bit fixed-point numbers between 0 and 1, and choose the two filtering bounds as $\frac{1}{2} \pm \frac{1}{16}$. In expectation, this filters out 7/8 of the elements. The probability that one of the middle elements is filtered out is 6×10^{-5} for a database of size 2^{20}, and this probability keeps decreasing with the size of the database.

When $\log N$ does not exceed 32, we use 32-bit integers for the extra indices. The hint IDs in our single-server version can also use 32-bit numbers since they will reset periodically upon offline phases.

In the two-server version, however, hint IDs can grow unbounded, so we use 64-bit integers for them.

4.2 Experimental Setup

Baselines. We compare with several practical two-server and single-server schemes, which we briefly describe below.

Two-server baselines include:

- The protocol of Boyle, Gilboa, and Ishai [4] based on distributed point functions (DPF) is the state-of-the-art two-server PIR scheme that uses linear server computation. It has a logarithmic request size and a constant response overhead. We use their C++ implementation.[1]
- The protocol of Kogan and Corrigan-Gibbs for checklists [19] is the first two-server amortized sublinear PIR scheme that is implemented. Their scheme has a logarithmic request size and a constant response overhead but requires either linear client computation or linear client storage. Their implementation in Go[2] uses linear client storage.
- TreePIR by Lazzaretti and Papamathou [22] is the state-of-the-art two-server amortized sublinear PIR scheme. Their scheme uses sublinear client storage and client computation and has a logarithmic request size. The downside of their scheme is the $O(\sqrt{N})$ response overhead. We use their implementation in Go.[3]

Single-server baselines include:

- Spiral PIR by Menon and Wu [26] is the latest single-server single-query PIR. It is based on lattice-based leveled FHE and needs to perform a linear amount of homomorphic operations at the server. We use their C++ implementation.[4]
- SimplePIR by Henzinger et al. [17] is a single-server stateful PIR scheme that still uses linear arithmetic operations on the server. We use their implementation in Go.[5]
- Piano PIR by Zhou et al. [34] is the latest single-server stateful PIR. We discuss it in detail in Section 5. We use their implementation of the updated version in Go.[6]

Experimental setup. We run all experiments on an AWS m5.8xlarge instance equipped with a 3.1 GHz Intel Xeon processor and 128 GB RAM. Our instance runs Ubuntu 22.04, GCC 11.3, and Golang 1.18. We run our scheme and all baselines with a single thread. We test the performance of our scheme and the baseline schemes with databases of varying entry counts and entry sizes. We first test databases with 2^{20}, 2^{24}, and 2^{28} entries while fixing the entry size to 32 bytes. We then fix the database to 2^{28} entries and test 8-byte and 256-byte entry sizes.

4.3 Evaluation Results

Two-server schemes. Table 2 gives a performance comparison of two-server PIR and stateful PIR schemes. The offline phases of the three stateful PIR schemes are run only once, so their amortized per-query costs are simply the online costs after sufficiently many queries are made. The checklist implementation crashed in our last experiment, so its results for the 64 GB database are missing. The DPF implementation does not support 256-byte entries, so its computation result for the 64 GB database is estimated.

DPF-PIR requires no offline phase or client storage. It also has efficient communication ranging from 0.91 KB to 1.52 KB in our tests. Its computation is linear in database size and grows from 2.5 milliseconds on a 32 MB database to 5960 milliseconds on a 64 GB database. Overall, DPF-PIR is very efficient in all aspects for small databases but is costly in computation for large databases.

In comparison, the three stateful PIR schemes require offline phases and client storage and, in return, achieve orders of magnitude lower per-query computation.

The checklist scheme boasts the lowest communication cost among the schemes we test. It also has a low online computation cost that is comparable to our scheme. Its biggest downside is the linear client storage. This cost is manageable for small databases but becomes prohibitive for large databases. For example, on the 8 GB database, the checklist scheme's client storage is over 1 GB, about one-eighth of the entire database and $\geq 20\times$ of TreePIR and our scheme.

TreePIR requires the smallest client storage among the three but has a high per-query communication cost that is two orders of magnitude larger than our scheme. Its per-query computation is also around $3.8 - 12.8\times$ higher than our scheme. We also test TreePIR with an extra single-server PIR call (not shown in the table). Its communication would improve to around 30 KB (refer to the Spiral result in Table 3), but its computation would worsen significantly (refer to the discussion in Section 5).

Our scheme achieves a balance of low client storage, low communication, and low computation for all database parameters, by avoiding major bottlenecks in previous schemes such as linear client storage, linear server computation, or high communication.

Single-server schemes. Table 3 gives a performance comparison of single-server PIR and stateful PIR schemes. The amortized per-query cost of Piano and our scheme are calculated as the offline cost divided by the number of queries supported per offline, plus the online cost. Spiral has no offline phase, and SimplePIR has a one-time offline phase, so their amortized per-query costs are simply the online costs after sufficiently many queries are made. Spiral crashed in our last experiment, so its results for the 64 GB database are missing. The SimplePIR implementation's offline phase also crashed in our last experiment; luckily, their implementation provides a way to test online efficiency without running the offline phase (and naturally, without correctness).

Spiral's communication cost remains relatively stable at different database parameters. Its linear server computation, however, is expensive even for small databases and becomes prohibitive for large databases. Concretely, its per-query computation is over 3 seconds for a 512 MB database and over 30 seconds for an 8 GB database. In comparison, our scheme is thousands of times faster than Spiral in per-query computation: just 4.5 milliseconds on the same 8 GB database. In terms of per-query communication, our scheme is better than Spiral on small databases but becomes worse on large databases due to the $\Omega(\sqrt{N})$ request size.

SimplePIR's online server computation is a constant factor better than Spiral's, but it is still linear and still very expensive for

[1] https://github.com/dkales/dpf-cpp
[2] https://github.com/dimakogan/checklist
[3] https://github.com/alazzaretti/treePIR
[4] https://github.com/menonsamir/spiral
[5] https://github.com/ahenzinger/simplepir
[6] https://github.com/wuwuz/Piano-PIR-new

Table 2: Comparison of two-servers PIR schemes.

	Database Parameters	Client Storage (MB)	Offline		Online	
			Comm. (MB)	Compute (s)	Comm. (KB)	Compute (ms)
DPF-PIR		-	-	-	0.91	2.5
Checklist	2^{20} 32-byte entries	7.07	2.88	3.3	0.50	0.17
TreePIR	32 MB in total	2.88	2.88	1.0	65.9	0.45
This paper		3.76	3.76	2.3	2.26	0.12
DPF-PIR		-	-	-	1.1	47
Checklist	2^{24} 32-byte entries	78.60	11.53	73	0.56	0.72
TreePIR	512 MB in total	11.53	11.53	23	262.6	4.9
This paper		15.04	15.04	41	8.64	0.54
DPF-PIR		-	-	-	1.21	182.4
Checklist	2^{28} 8-byte entries	1085.27	11.53	1394	0.52	1.9
TreePIR	2 GB in total	11.53	11.53	398	262.6	20
This paper		30.16	30.16	636	34.0	2.19
DPF-PIR		-	-	-	1.31	745
Checklist	2^{28} 32-byte entries	1119.74	46.14	1141	0.64	1.8
TreePIR	8 GB in total	46.14	46.14	430	1049.6	14
This paper		60.16	60.16	842	34.1	2.7
DPF-PIR	2^{28} 256-byte entries	-	-	-	1.52	5960
TreePIR	64 GB in total	369.09	369.09	1843	8389.6	67
This paper		340.16	340.16	2242	35.0	5.23

Table 3: Comparison of single-server PIR schemes.

	Database Parameters	Client Storage (MB)	Offline		Online		Amortized per query	
			Comm. (MB)	Compute (s)	Comm. (KB)	Compute (ms)	Comm. (KB)	Compute (ms)
Spiral		-	-	-	28	767	28	767
SimplePIR	2^{20} 32-byte entries	20.9	20.9	4.8	40	14	40	14
Piano	32 MB in total	7.32	32	5.4	4.03	0.54	6.34	0.92
This paper		6.25	32	4	2.18	0.14	2.99	0.25
Spiral		-	-	-	34.0	3177	34.0	3177
SimplePIR	2^{24} 32-byte entries	86.8	86.6	154	168	103	168	103
Piano	512 MB in total	32.97	512	96	16.03	1.35	23.73	2.8
This paper		25	512	65	8.56	0.62	11.76	1.0
Spiral		-	-	-	34.5	8427	34.5	8427
SimplePIR	2^{28} 8-byte entries	173.4	173.4	623	338	319.1	338	319.1
Piano	2 GB in total	70.5	2048	1565	64	4.1	70.6	9.0
This paper		40	2048	989	34.02	2.4	37.22	3.9
Spiral		-	-	-	35.0	30273	35.0	30273
SimplePIR	2^{28} 32-byte entries	352.98	352.98	2788	688	1123	688	1123
Piano	8 GB in total	144.75	8192	1822	64.03	3.8	90.41	9.6
This paper		100	8192	1146	34.06	2.7	46.86	4.5
SimplePIR	2^{28} 256-byte entries	983.64	983.64	failed	1965	7935	1965	7935
Piano	64 GB in total	837.75	65536	2775	64.25	5.3	275.3	14.0
This paper		660	65536	2327	34.5	4.2	136.9	7.8

large databases. Piano, being a sublinear scheme, addresses the server computation bottleneck, but it has to weaken the correctness guarantee of PIR.

Our scheme achieves better communication and computation than these two stateful PIR schemes. Compared with SimplePIR, the state-of-the-art scheme that provides the same standard PIR correctness, our scheme is $9-14\times$ better in communication and hundreds of times faster in computation. Compared with the latest version of Piano PIR, which is concurrent with our work, our communication is about $2\times$ better, and our amortized computation is $1.7-3.7\times$ better. Moreover, we achieve these improvements while providing a stronger correctness guarantee.

Our single-server scheme does have a drawback (shared by Piano): offline communication is very high for large databases due to streaming the whole database. Even though this can be amortized over many online queries, it is still undesirable as it significantly delays the very first query.

5 Related Works

Private Information Retrieval (PIR) was first introduced by Chor et al. [8]. There has been an extensive list of works on both multi-server PIR and single-server PIR. Since this work focuses on the two-server and the single-server settings, we will focus on these two settings in this section and omit schemes that require three or more servers.

Single-query PIR with linear server computation. Research on PIR started with the simplest and most standard variant: a client has a *single* entry to fetch from the server. We call it single-query PIR. Chor et al. [8] gives the first single-query PIR scheme. Their scheme uses multiple non-colluding servers. With two servers, the communication cost of their scheme is $O(N^{1/3})$. The state-of-the-art two-server single-query scheme is based on *distributed point functions* [15], uses polylogarithmic communication, and is reasonably fast in computation.

Kushilevitz and Ostrovsky give the first single-server single-query PIR scheme [20] based on additive homomorphic encryption (AHE). Subsequent AHE-based schemes include [6, 11, 14]. Recent practical single-server single-query PIR schemes [1, 2, 25, 26, 28] have switched from AHE to lattice-based leveled Fully Homomorphic Encryption (FHE) to reduce server computation.

All the above schemes, multi-server and single-server ones alike, require linear server computation. As mentioned, this is unavoidable in the most standard single-query PIR model. This is formalized by Beimal, Ishai, and Malkin [3] as a lower bound that any PIR scheme where the server stores an unmodified N-entry database must incur $\Omega(N)$ computation at the server. Three avenues have been explored in an attempt to circumvent the linear server computation barrier: database preprocessing, batch PIR, and stateful PIR. We focus on stateful PIR in this section after briefly discussing the other two approaches below.

PIR with database preprocessing. In the same paper that established the $\Omega(N)$ server computation lower bound, Beimel, Ishai, and Malkin [3] also show that the lower bound can be circumvented by preprocessing and encoding the database offline. This approach is also taken by a line of works known as doubly efficient PIR [5, 7, 23]. These efforts have so far remained largely theoretical because they have to significantly blow up server storage (superlinearly or by the number of clients), require heavyweight theoretical tools (such as oblivious locally decodable codes or virtual black box obfuscation), or suffer from both drawbacks.

Batch PIR. Batch PIR [2, 18] also assumes the client has many entries to fetch from the server. The difference between batch PIR and stateful PIR is that batch PIR assumes the client has many queries to fetch in one go, while stateful PIR allows the client to generate queries sequentially (e.g., the client decides what the next query is after receiving the response for its previous query). This can be formally captured by the adaptive version of the stateful PIR in Section 2. Note that batch PIR is an easier problem than stateful PIR because the client can always send a batch of queries one by one, but it cannot batch chronologically sequential (and potentially causal) queries.

Ishai et al. [18] propose the first batch PIR scheme (and called it amortized PIR in their paper) using batch codes. Angel et al. [2] gives the first practical batch PIR scheme using cuckoo hashing. The Angel et al. scheme nicely amortizes the linear server computation cost: it costs $O(N)$ server computation to fulfill all the queries in the batch, no matter how large the batch is. But their scheme does not amortize the response overhead: $O(b)$ ciphertexts must be returned for a batch of b queries. Mughees and Ren [29] give a batch PIR scheme that amortizes the response overhead over the batch using vectorized FHE where a single ciphertext can hold as many queried entries as what can fit.

Stateful PIR. Patel, Persiano, and Yeo [31] propose the paradigm of stateful PIR in which the client retrieves hints privately in an offline phase and later uses these hints to speed up online queries. At some level, this offline phase can also be viewed as a preprocessing step, but it does not alter the server's database and hence requires no extra server storage. The goal of this first stateful PIR was less ambitious: it was to replace the linear homomorphic encryption operations with linear PRF evaluations, rather than circumventing the linear server computation bound. Recent works like SimplePIR and FrodoPIR [12, 17] further improved server computation to linear arithmetic operations.

Corrigan-Gibbs and Kogan [10] give the first amortized sublinear stateful PIR scheme. Their scheme initially works in the two-server setting and is later extended to the single-server setting using the idea of backup hints [9]. Several works resort to privately puncturable or programmable pseudorandom functions (PRF) to improve the request size from $\Omega(\lambda\sqrt{N})$ to polylogarithmic, first in the two-server setting [32] and later in the single-server setting [21, 33]. These works are mostly theoretical at the moment because privately puncturable/programmable PRFs are heavyweight theoretical tools and do not have practical instantiations.

It is worth noting that a more pressing performance bottleneck than the request size is the parallel repetitions. All of the above works [9, 10, 21, 32, 33] allow a small probability of correctness failure. Thus, their schemes must be repeated λ times in parallel to make the correctness failure probability negligible. This will blow up all efficiency metrics, including request size, response size, client storage, client computation, and server computation. To our knowledge, none of the above schemes has been implemented.

Two recent works give methods to eliminate this correctness failure and avoid the parallel repetitions [19, 22], leading to practical amortized sublinear stateful PIR schemes that have been implemented. As we have mentioned, both schemes only work for the two-server setting and have no clear paths to be extended to the single-server setting. Moreover, both schemes come with substantial efficiency losses. The Checklist scheme by Kogan and Corrigan-Gibbs [19] requires either $\Theta(N)$ client storage or $\Theta(N)$ client computation per query. Since the motivation of stateful PIR is to avoid linear server computation, it is hard to justify shifting a linear cost to the client, which is often more resource-constrained than

the server. The TreePIR scheme by Lazzaretti and Papamathou [22] is more relevant to our work, and we discuss it in more detail next.

TreePIR [22]. The TreePIR scheme adds a logarithmic factor to both client and server computation due to the use of the tree-based weakly private puncturable PRF. It also increases the response overhead to $\Theta(\sqrt{N})$. This large response overhead is usually more problematic than a large request size because requests are measured in $\log N$-sized words (usually less than 32 bits), while the responses are measured in the database entry size, which can be hundreds of bytes or more.

In theory, the large response overhead can be mitigated by invoking an extra single-server single-query PIR. However, this would not be efficient in practice. Most importantly, state-of-the-art FHE-based single-query PIR schemes perform a one-time preprocessing of the database using Number-Theory Transform (NTT) [1, 2, 26, 28]. Since the $\Theta(\sqrt{N})$-sized response is computed by the server based on the query, this NTT preprocessing step will have to be performed on the fly at the end of each query. This will make FHE-based PIR at least an order of magnitude slower. For example, for the $N = 2^{20}$ database with 32-byte entries, the NTT preprocessing phase of Spiral PIR on $\sqrt{N} = 2^{10}$ entries would cost 1100 ms, which is even slower than simply running Spiral PIR on the entire (preprocessed) $N = 2^{20}$ database. In addition, this approach will inherit other drawbacks of FHE-based PIR, such as large ciphertexts of lattice encryption, key-dependent security, and megabytes of server storage per client, the latter two of which result from the substitution keys for query (de-)compression.

Despite the shortcomings in the response overhead and the need for two servers, the TreePIR scheme introduces an elegant technique that is crucial for our work. Their scheme uses a more structured hint construction where the database is divided into equal-sized partitions, and each subset consists of one index per partition. These partition-based hints are more amenable to succinct pseudorandom representations and faster membership testing. Thus, they enable more space-efficient hint storage and offline processing. Our work adopts their partition-based hints.

Piano PIR [34]. The initial version of Piano PIR [34] adapts the TreePIR scheme to the single server setting by combining TreePIR's partition-based hints and Corrigan-Gibbs et al.'s backup hints [9]. Naturally, it inherits the $\Theta(\sqrt{N})$ online response overhead from TreePIR. Concurrent with our work, an updated version of Piano PIR achieves $O(1)$ online response by adding singleton entries as extra hints.

A major downside of (both versions of) Piano PIR is that they have to weaken the correctness guarantee of PIR and require that the query sequence is not influenced by the adversary. This is because their backup hints are "partition-specific". In other words, each backup hint is tied to a particular partition and can only replenish a consumed hint when the client queries a database entry belonging to that partition. As a result, they require the queries in the sequence to be balanced across all partitions. To do so, they let the server permute the entire database and publish the permutation key. They then require the client's query sequence to be independent of the server's permutation. Because the server's permutation is public, an adversary that can influence the client queries can easily force the client to query many entries in the same partition and make Piano PIR fail in correctness. Therefore, Piano PIR is weaker than a standard (stateful) PIR scheme defined in Section 2. We also discussed the importance of the standard PIR definition in Section 2.

In terms of techniques, Piano PIR and our work may appear similar at first glance because both works adopt established techniques in the literature: TreePIR's partition-based hints [22] and Corrigan-Gibbs et al.'s backup hints [9]. But beyond that, our work does not borrow any idea invented in Piano PIR, and vice versa. The main techniques of Piano PIR are partition-specific backup hints and singleton hints. We propose a new hint system that only involves a random half of the partitions and the dummy subset technique to make the new hint system secure.

6 Conclusion

We have presented simple and practical stateful PIR schemes with amortized sublinear communication and computation for both the two-server and single-server settings. Our schemes avoid the major performance bottlenecks in prior works: parallel repetition, linear client storage, and large response overhead.

Our schemes also have drawbacks that call for further studies. An obvious one is the $\Omega(\sqrt{N})$ request size. There exist techniques to reduce the request size, but the challenge is to do so without sacrificing other aspects of the algorithm. A limitation shared by all existing amortized sublinear schemes is that the $O(\lambda\sqrt{N})$ client storage, while sublinear, is still quite large in practice. An indirect consequence is that the single-server offline phase cannot do much better than streaming the whole database when the client needs so many hints. Other general challenges involving stateful PIR include how to handle updates to the database and how to support queries by keywords, and recent works have made some progress in these directions [19, 24].

Acknowledgement. This work is funded in part by the National Science Foundation award #2246386 and the Google Research Scholar Program.

References

[1] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. 2021. Communication–Computation Trade-offs in PIR. In *30th USENIX Security Symposium*. 1811–1828.

[2] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. 2018. PIR with compressed queries and amortized query processing. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 962–979.

[3] Amos Beimel, Yuval Ishai, and Tal Malkin. 2000. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *20th Annual International Cryptology Conference (CRYPTO)*. Springer, 55–73.

[4] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1292–1303.

[5] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. 2017. Can we access a database both locally and privately?. In *Theory of Cryptography: 15th International Conference*. Springer, 662–693.

[6] Christian Cachin, Silvio Micali, and Markus Stadler. 1999. Computationally private information retrieval with polylogarithmic communication. In *International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT)*. Springer, 402–414.

[7] Ran Canetti, Justin Holmgren, and Silas Richelson. 2017. Towards doubly efficient private information retrieval. In *Theory of Cryptography: 15th International Conference*. Springer, 694–726.

[8] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. 1998. Private information retrieval. *Journal of the ACM (JACM)* 45, 6 (1998), 965–981.

[9] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. 2022. Single-server private information retrieval with sublinear amortized time. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 3–33.

[10] Henry Corrigan-Gibbs and Dmitry Kogan. 2020. Private information retrieval with sublinear online time. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 44–75.

[11] Ivan Damgård and Mads Jurik. 2001. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In *4th International Workshop on Practice and Theory in Public Key Cryptosystems (PKC)*. Springer, 119–136.

[12] Alex Davidson, Gonçalo Pestana, and Sofía Celi. 2023. Frodopir: Simple, scalable, single-server private information retrieval. *Proceedings on Privacy Enhancing Technologies* (2023).

[13] Zeev Dvir and Sivakanth Gopi. 2016. 2-server PIR with subpolynomial communication. *Journal of the ACM (JACM)* 63, 4 (2016), 1–15.

[14] Craig Gentry and Zulfikar Ramzan. 2005. Single-database private information retrieval with constant communication rate. In *International Colloquium on Automata, Languages, and Programming*. Springer, 803–815.

[15] Niv Gilboa and Yuval Ishai. 2014. Distributed point functions and their applications. In *33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Springer, 640–658.

[16] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. 2016. Scalable and private media consumption with Popcorn. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 91–107.

[17] Alexandra Henzinger, Matthew M Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. 2023. One Server for the Price of Two: Simple and Fast Single-Server Private Information Retrieval. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3889–3905.

[18] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2004. Batch codes and their applications. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. 262–271.

[19] Dmitry Kogan and Henry Corrigan-Gibbs. 2021. Private blocklist lookups with checklist. In *30th USENIX security symposium (USENIX Security 21)*. 875–892.

[20] Eyal Kushilevitz and Rafail Ostrovsky. 1997. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings 38th annual symposium on foundations of computer science*. IEEE, 364–373.

[21] Arthur Lazzaretti and Charalampos Papamanthou. 2023. Near-optimal private information retrieval with preprocessing. In *Theory of Cryptography Conference*. Springer, 406–435.

[22] Arthur Lazzaretti and Charalampos Papamanthou. 2023. Treepir: Sublinear-time and polylog-bandwidth private information retrieval from ddh. In *Annual International Cryptology Conference*. Springer, 284–314.

[23] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. 2023. Doubly efficient private information retrieval and fully homomorphic RAM computation from ring LWE. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*. 595–608.

[24] Yiping Ma, Ke Zhong, Tal Rabin, and Sebastian Angel. 2022. Incremental Offline/Online PIR. In *31st USENIX Security Symposium*. 1741–1758.

[25] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. 2016. XPIR: Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies* (2016), 155–174.

[26] Samir Jordan Menon and David J Wu. 2022. Spiral: Fast, high-rate single-server PIR via FHE composition. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 930–947.

[27] Prateek Mittal, Femi Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. 2011. PIR-Tor: Scalable Anonymous Communication Using Private Information Retrieval. In *20th USENIX security symposium*.

[28] Muhammad Haris Mughees, Hao Chen, and Ling Ren. 2021. OnionPIR: Response efficient single-server PIR. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2292–2306.

[29] Muhammad Haris Mughees and Ling Ren. 2023. Vectorized batch private information retrieval. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 437–452.

[30] David R. Musser. 1997. Introspective Sorting and Selection Algorithms. , 983-993 pages.

[31] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. 2018. Private stateful information retrieval. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1002–1019.

[32] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce Maggs. 2021. Puncturable pseudorandom sets and private information retrieval with near-optimal online bandwidth and time. In *41st Annual International Cryptology Conference (CRYPTO)*. Springer, 641–669.

[33] Mingxun Zhou, Wei-Kai Lin, Yiannis Tselekounis, and Elaine Shi. 2023. Optimal single-server private information retrieval. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 395–425.

[34] Mingxun Zhou, Andrew Park, Wenting Zheng, and Elaine Shi. 2023. PIANO: Extremely Simple, Single-Server PIR with Sublinear Server Computation. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 55–55.