Single-server Stateful PIR with Verifiability and Balanced Efficiency

Pranav Shriram Arunachalaramanan
University of Illinois at Urbana-Champaign
psa3@illinois.edu

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

***Abstract*—Recent stateful private information retrieval (PIR) schemes have significantly improved amortized computation and amortized communication while aiming to keep client storage minimal. However, all the schemes in the literature still suffer from a poor tradeoff between client storage and computation. We present `BALANCED-PIR`, a stateful PIR scheme that effectively balances computation and client storage. For a database of a million entries, each of 8 bytes, our scheme requires 0.2 MB of client storage, 0.2 ms of amortized computation, and 11.14 KB of amortized communication. Compared with the state-of-the-art scheme using a similar storage setting, our scheme is almost 9x better in amortized computation and 40x better in offline computation.**

Verifiable private information retrieval has been gaining more attention recently. However, all existing schemes require linear amortized computation and huge client storage. We present Verifiable `BALANCED-PIR`, a verifiable stateful PIR scheme with sublinear amortized computation and small client storage. In fact, our Verifiable `BALANCED-PIR` adds modest computation, communication, and storage costs on top of `BALANCED-PIR`. Compared with the state-of-the-art verifiable scheme, the client storage of our scheme is 100x smaller, the amortized computation is 15x less, and the amortized communication is 2.5x better.

1. Introduction

Private information retrieval (PIR) [4] enables a client to retrieve entries from a database hosted by a server without the server knowing which entries are being retrieved. PIR has a wide range of applications on the internet including anonymous communication [5], [6], [7], e-commerce [8], [9], biometric security [10], [11], secure advertising [12], [13], [14], [15], privacy-preserving media streaming [16] among many others.

PIR schemes in the standard stateless setting are subject to a fundamental barrier that the server must operate on every database entry. Fundamentally, if the PIR scheme does not require the server to operate on any specific entry, then the server can conclude that the specific entry is not retrieved by the client, leaking information about the client's query. While recent schemes have made significant progress [17], [18], [19], [20], [21], [22], the computational cost remains high due to this barrier.

A more recent and encouraging attempt to overcome this fundamental computational barrier is the paradigm of *stateful* PIR schemes. At a high level, in a stateful PIR scheme, the client and/or the server perform a (usually costly) offline preprocessing phase, and the client stores the preprocessed information (called hints) in its local storage. The hints are then leveraged to speed up queries in the online phase. Therefore, a stateful PIR scheme has three key efficiency metrics, i) computation, ii) communication, and iii) client storage.

There have been significant recent efforts in stateful PIR schemes [23], [24], [25], [26], [27], [28], [1], [2], [29], [30] that aim to improve the computation and communication while keeping the client storage as small as possible. Despite this wide body of work, most single-server stateful PIR schemes suffer from inefficiencies in computation, communication, and/or client storage. The state-of-the-art practical schemes in this setting [1], [27] have efficient computation and communication but require a large amount of client storage. For instance, to use the PIR scheme from the work of Ren et al. [1] with a 32 MB database consisting of 2^{20} entries, each of 32 bytes, the client needs to store 6.25 MB of data, which is almost 20% the size of the database.

The recent scheme of Wang and Ren [2] achieves optimal client storage, but requires heavy client computation due to the use of an expensive cryptographic primitive known as the small-domain PRP [31]. Therefore, the first goal of this paper is to construct a single-server stateful PIR scheme that has a good balance between client storage and computation.

Another fundamental but understudied issue to the deployment of PIR schemes is the integrity of retrieved database entries. Most PIR schemes assume that the server is semi-honest and follows the protocol faithfully. If the server ends up being malicious, it can result in significant security risks. A recent line of work known as *Verifiable PIR* schemes [32], [3], [33] seeks to guarantee the integrity of retrieved entries. At a high level, in these schemes, the server commits to a database and publishes a database digest. Thereafter, the client can verify the integrity of the retrieved database entries against the committed database. When the client detects malicious behaviors by the server, it can abort without leaking any information about the query indices. However, the state-of-the-art practical verifiable PIR scheme [3] requires linear server computation and enormous client

TABLE 1: Approximate and relative client storage, amortized computation, and amortized communication of various single-server stateful PIR schemes for small entry sizes.

Scheme	Verifiable	Client storage	Amortized computation	Amortized communication
SimplePIR [1]	✗	$\sim 400\times$	$\sim 50\times$	$\sim 10\times$
Ren et al. [1]	✗	$\sim 100\times$	$\sim 1\times$	$1\times$
Wang and Ren [2]	✗	$1\times$	$\sim 400\times$	$\sim 15\times$
BALANCED-PIR	✗	$\sim 10\times$	$1\times$	$\sim 10\times$
VeriSimplePIR [3]	✓	$\sim 1600\times$	$\sim 100\times$	$\sim 20\times$
Verifiable BALANCED-PIR	✓	$\sim 15\times$	$\sim 10\times$	$\sim 10\times$

storage. Therefore, our second goal is to construct a single-server verifiable stateful PIR scheme that has sublinear server computation and small client storage. This was raised as an open problem in the work of de Castro and Lee [3].

Contribution 1: Practical single-server stateful PIR with small client storage. The first contribution of this paper is a new PIR scheme, called BALANCED-PIR, which achieves a good balance between client storage and computation. BALANCED-PIR uses an order of magnitude smaller storage than the state-of-the-art practical scheme of Ren et al. [1]. If we normalize the amount of client storage of the two schemes, then the amortized computation of our scheme is $7\times$ to $8.9\times$ better, and the amortized communication is comparable or better for small entry sizes up to 32 bytes. Additionally, the offline phase of our scheme is $36\times$ to $39\times$ faster, thereby significantly shortening the initial wait time before the client can make queries. In comparison with the scheme of Wang and Ren (WR) [2], BALANCED-PIR has $390\times$ better amortized computation, similar amortized communication, but $8.5\times$ to $10\times$ larger client storage.

Contribution 2: Practical verifiable stateful PIR. We present a verifiable version of BALANCED-PIR that guarantees integrity with small extra client storage and minimal additional server computation on top of BALANCED-PIR. This is the first Verifiable PIR scheme with sublinear amortized computation. In comparison with the state-of-the-art verifiable single-server PIR scheme of de Castro and Lee [3], our scheme has $117\times$ better client storage, $15\times$ to $55\times$ faster computation, and $2.9\times$ better communication for 8-byte entries. The paradigm of Verifiable stateful PIR can also be adapted to work with other stateful PIR schemes such as [1], [27].

2. Model and Preliminaries

2.1. Private Information Retrieval (PIR)

Private information retrieval (PIR) is a protocol between a client and a database wherein the client wants to retrieve specific data entries from the database. The database DB is an array consisting of N entries indexed from 1 to N. A PIR protocol must satisfy the conditions of correctness and privacy. At a high level, the conditions are defined as follows. i) Correctness - Given a sequence of query indices, the client must always be able to retrieve the database entries corresponding to the query indices, ii) Privacy - Given the client's interaction with the database server, the server learns nothing about the client's query indices.

2.2. Stateful PIR

Stateful PIR, at a high level, involves two phases called the offline phase and the online phase. During the offline phase, the client interacts with the server and samples information about the database entries called hints. The client stores the hints locally. During the online phase, the client executes a sequence of queries to retrieve database entries one after the other in the following manner. In each query, given a query index, the client uses a hint to construct a *request* and sends the *request* to the server. The server sends back a *response*. Using the server's *response*, the client then computes an *answer*. The correctness and privacy requirements of a stateful PIR scheme are captured using the following definitions:

- **Correctness**: Given any sequence of queries, the *answer* of each query matches the entry at the query index of the database, except with negligible probability.
- **Privacy**: Privacy can be defined using the following game between the client and the (semi-honest) server:
  - The server selects two sets of query indices $[i_{0,1}, i_{0,2}, ..., i_{0,k}]$ and $[i_{1,1}, i_{1,2}, ..., i_{1,k}]$, and sends them to the client.
  - The client samples a random bit $\mathrm{b} \xleftarrow{\$} \{0,1\}$, and interacts with the server to execute PIR queries for indices $[i_{b,1}, i_{b,2}, ..., i_{b,k}]$ according to the PIR scheme.
  - The server tries to determine b.

  A PIR scheme is secure if the server guesses b correctly with at most $\frac{1}{2} + \epsilon$ probability, where ϵ is negligible.
  Adaptive privacy. A PIR scheme achieves privacy against an adaptive adversary if, in the above security game, the adversary can choose every pair of query indices $i_{0,k}, i_{1,k}$ after the client and the adversary complete the previous $k-1$ queries.

2.3. Verifiable Stateful PIR

A verifiable stateful PIR scheme is a stateful PIR scheme with additional procedures to ensure the integrity of the retrieved database entries. Specifically, the server commits to a database and the retrieved database entries are verified against that commitment.

To elaborate, the server computes and publishes a *database digest*. In our scheme, the digest is simply a collision-resistant hash of the database. During the offline phase, the client verifies that the database is consistent with the published *database digest*. Moreover, the client may compute and store additional information about the database entries to be used for verification online. During the online phase, the client executes a sequence of queries, and for each query, verifies if the *answer* matches the queried entry of the committed database. In the event that the client detects malicious behavior by the server, the client *protests* against the server and can *abort* its execution. Note that the protest must be done in a secure manner that leaks no information about the query indices to the server.

A verifiable stateful private information retrieval scheme must satisfy the following properties except with negligible probability.

- **Completeness**: If the server is semi-honest, then the *answer* of each query matches the queried entry of the committed database, and the client verification passes.
- **Soundness**: If the *answer* of a query passes the verification, then it matches the queried entry in the committed database.
- **Privacy**: Identical to the privacy definition of (non-verifiable) stateful PIR, except that the adversary is now actively malicious and also sees the client's protest (if any) in addition to the requests.

2.4. Cryptographic primitives

A Pseudorandom function (PRF) is a function that maps inputs in the domain to seemingly random outputs in the range. In our schemes, we instantiate the PRF using AES.

A collision-resistant hash function maps a message of arbitrary length to an output of fixed length. Finding two messages that map to the same hash output is computationally infeasible. We use AES-hash [34] in our verifiable stateful PIR scheme due to its practical efficiency from the hardware support for AES in modern processors. Our implementations leverage the AES-NI instruction set in the Intel processors.

A Pseudorandom Permutation (PRP) is a function that maps inputs in the domain to seemingly random outputs in the range with a one-to-one mapping between inputs and outputs. A block cipher like AES is a PRP whose domain and range are both 128-bit strings. In contrast, a small-domain PRP has a "small" domain in the sense that a polynomial-time adversary that seeks to distinguish the PRP from a truly random permutation can examine all the input-output pairs. Small-domain PRP turns out to be challenging and expensive to construct. The state-of-the-art small-domain PRP construction by Morris and Rogaway [31] requires $O(\log N + \lambda)$ PRF calls in expectation per evaluation, where N is the domain size of the permutation and λ is a statistical security parameter. For instance, given $N = 2^{20}$ and $\lambda = 60$, each small-domain PRP evaluation requires around 600 AES calls in expectation.

3. BALANCED-PIR

3.1. High-level idea

To construct a single-server stateful PIR scheme, we begin by devising two single-server stateful PIR schemes PIR-PRIMARY and PIR-BACKUP with the following characteristics. PIR-PRIMARY has efficient computation, but the client has to store a large number of hints to achieve a small probability of correctness failure. On the other hand, PIR-BACKUP requires heavy computation but is guaranteed to be correct even with much smaller client storage. We then combine the two schemes to design what we call BALANCED-PIR, with moderate client storage and efficient computation.

The high-level idea behind the construction of BALANCED-PIR is as follows. During the offline phase, the client samples hints for both PIR-PRIMARY and PIR-BACKUP. The number of PIR-PRIMARY hints sampled by the client determines the likelihood that PIR-PRIMARY fails to fulfill a query. The client will provision fewer hints than what correctness requires. Then, to guarantee the correctness of BALANCED-PIR, the client uses PIR-BACKUP whenever PIR-PRIMARY fails. The number of hints stored for PIR-PRIMARY is tuned to ensure that the client uses PIR-PRIMARY in most cases and falls back to PIR-BACKUP on rare occasions. Consequently, the amortized computation of BALANCED-PIR is close to that of PIR-PRIMARY.

We emphasize that for this plan to work, it is critical that the server does not learn whether PIR-PRIMARY or PIR-BACKUP is used for any query. Otherwise, the server can gain insights on the hint distribution of PIR-PRIMARY, and eventually break privacy. Therefore, the *request* distributions of PIR-PRIMARY and PIR-BACKUP must be identical. Additionally, the server protocol of the two schemes must also be identical.

Concretely, PIR-BACKUP is instantiated using the recent scheme of Wang and Ren [2]. We will design a new PIR scheme to serve as our PIR-PRIMARY, and as required, it has the same request distribution and server protocol as PIR-BACKUP.

In both PIR-PRIMARY and PIR-BACKUP, the database is logically segmented into $\sqrt{N}$ partitions with each partition consisting of $\sqrt{N}$ entries. The i^{th} partition consists of database entries with indices in the range $[i\sqrt{N}, (i+1)\sqrt{N} - 1]$.

3.2. PIR-BACKUP construction

We use the PIR scheme from the work of Wang and Ren [2] (WR PIR) to instantiate PIR-BACKUP. We describe the construction of WR PIR at a very high level and then describe how this scheme is adapted for PIR-BACKUP.

Offline. During the offline phase, the client samples hints according to a hint table. Specifically, the hint table consists of $\sqrt{N}$ rows and $m = (\beta + 1)\sqrt{N}$ columns where β is a

configurable parameter. The i^{th} row in the table consists of the $\sqrt{N}$ indices from partition i and $\beta\sqrt{N}$ $\perp$ symbols, randomly permuted. A $\perp$ indicates that the position in the row is empty. To minimize client storage, instead of storing the hint table locally, the client stores the permutation corresponding to each row. Specifically, the scheme uses a computationally expensive cryptographic primitive called small-domain PRP (section 2.4) and the client stores the small-domain PRP key for each row.

The j^{th} hint is the parity of the entries corresponding to the indices in the j^{th} column of the hint table. The client uses the small-domain PRP keys to determine the indices in each column of the hint table.

Online. During the online phase, given a query index, the client first selects the column in the hint table that contains the query index. Using the selected column, the client creates a *request* consisting of all the indices in the column. However, the query index in the *request* is switched to a random index from the same partition. Then, the client sends the *request* to the server and receives as the *response* the database entries at the indices in the *request*. The client computes the queried database entry using the *response* and the hint (column parity). Lastly, each index in the column is relocated to a pseudorandom cell in the same row that contains $\perp$. Additionally, we note that the client requires $O(\sqrt{N})$ small-domain PRP executions in expectation, for each query.

Efficiency. We emphasize again that the computational cost of the scheme is high due to invocations of small-domain PRP. Specifically, the client makes $O(N)$ invocations of small-domain PRP in the offline phase and $O(\sqrt{N})$ invocations of small-domain PRP per query in the online phase, which significantly blow up the computational cost.

Adapting WR PIR for `PIR-BACKUP`. We make two minor changes to WR PIR. Note that the WR scheme may contain $\perp$ in the request, indicating that no index is selected from specific partitions. Looking ahead, this is a property that cannot be met by `PIR-PRIMARY`'s request. Specifically, a WR PIR *request* may not select any index from some partitions, whereas in `PIR-PRIMARY` a random index is selected from each partition to the *request*. Therefore, in `PIR-BACKUP`, we replace each $\perp$ with a random dummy index from the same partition. This change makes the *request* distribution of `PIR-BACKUP` identical to that of `PIR-PRIMARY`, with a small additional communication overhead.

Second, β is always set to 1 in [2], but we make it a configurable parameter. β determines the number of columns in the hint table. A larger β results in larger client storage but allows the client to make more queries per offline phase, thereby reducing the amortized cost of the offline phase. Looking ahead, selecting β judiciously plays a crucial role in ensuring efficient amortized computation of `BALANCED-PIR`.

Summary. For the purpose of understanding our construction of `BALANCED-PIR`, a reader does not have to know all the details of WR PIR. It is sufficient to know the following: i) each *request* consists of one random index from every partition, and ii) the server returns entries at *request* indices as the *response*.

Algorithm 1 The offline algorithm for `PIR-PRIMARY`

Input: α - Storage parameter
State: n - Next backup hint position, hints - Hints, bHints - Backup hints

$M \leftarrow \alpha\sqrt{N}$
sHints $\leftarrow$ []
for $k \in [0, 2M-1]$ **do**
  $\text{hint}_\text{k} \leftarrow$ { id: k, parity: 0, eP: $\perp$, ePI: $\perp$ }
  sHints.push(hint_k)
end for
for $j \in [0, \sqrt{N}-1]$ **do**
  Stream partition $\mathsf{DB}[j\sqrt{N} : (j+1)\sqrt{N}-1]$ from the server
  for $k \in [0, 2M-1]$ **do**
    $r_{j,k} \leftarrow \mathsf{PRF}(j \parallel k) \mod \sqrt{N}$
    $x \leftarrow \mathsf{DB}[r_{j,k} + j\sqrt{N}]$
    sHints[k].parity $\leftarrow$ sHints[k].parity $\oplus$ x
  end for
end for
hints $\leftarrow$ sHints$[0, M-1]$
bHints $\leftarrow$ sHints$[M, 2M-1]$
$n \leftarrow 0$

Algorithm 2 selectHint procedure

Input: q - Query index, hints - Hints

$s \leftarrow \lfloor \frac{q}{\sqrt{N}} \rfloor$
$t \leftarrow q \mod \sqrt{N}$
for $k \in [0, M-1]$ **do**
  if hints[k].eP = s and hints[k].ePI = t **then**
    return (hints[k], k)
  else if hints[k].eP $\neq s$ **then**
    $r_{s,\text{hints[k].id}} \leftarrow \mathsf{PRF}(s \parallel \text{hints[k].id}) \mod \sqrt{N}$
    if $r_{s,\text{hints[k].id}} = t$ **then**
      return (hints[k], k)
    end if
  end if
end for
return ($\perp$, $\perp$)

3.3. `PIR-PRIMARY` construction

We emphasize again that the request distribution and server protocol need to be identical in `PIR-PRIMARY` and `PIR-BACKUP`. Recall that in `PIR-BACKUP`, the *request* consists of $\sqrt{\text{N}}$ indices, one random index per partition, and the server protocol simply returns all the entries at the *request* indices.

We note that the state-of-art schemes of Zhou et al. [27] and Ren et al. [1] could be adapted to have the above request distribution and server protocol. However, most of

Algorithm 3 getRequestIndices procedure

Input: h - Selected hint, l - ID of next backup hint, s - Partition of the query index

request $\leftarrow$ []
for $j \in [0, \sqrt{N} - 1]$ **do**
  if $j = s$ **then**
    $r_{j,l} \leftarrow \mathsf{PRF}(j \parallel l) \mod \sqrt{N}$
    request.push($r_{j,l}$)
  else if $j \neq$ h.eP **then**
    $r_{j,\mathrm{h.id}} \leftarrow \mathsf{PRF}(j \parallel \mathrm{h.id}) \mod \sqrt{N}$
    request.push($r_{j,\mathrm{h.id}}$)
  else
    request.push(h.ePI)
  end if
end for
return request

the intricacies in those two works are specifically designed to improve the server protocol, i.e., the server does not have to send back all $\sqrt{N}$ entries but can instead send back just their XOR sum. Since we anyway need the server to send back all $\sqrt{N}$ entries, we design a new scheme that is simpler.

We next describe our protocol in detail.

Offline. During the offline phase, the client samples $M = \alpha\sqrt{N}$ hints where α is configurable parameter For each hint, the client uses a PRF to sample one (pseudo-)random index independently and uniformly from each partition. These indices form the *hint set*. Let $r_{j,k} = \mathsf{PRF}(j \parallel k) \mod \sqrt{N}$ be the index from the j^{th} partition sampled for the k^{th} hint. Then, the client computes the parity of entries at indices in the *hint set* as $P_k = \bigoplus_{j=0}^{\sqrt{N}-1} \mathsf{DB}[r_{j,k} + j\sqrt{N}]$. *Hint entries* are the database entries located at the indices in the *hint set*.

The k^{th} hint consists of the hint id k, parity P_k, and an *extra index*. The *extra index* allows the client to overwrite one index in the *hint set*, and its purpose will be explained later. The *extra index* is maintained as a combination of *extra partition*, which is the partition containing the *extra index*, and *extra partition index*, which is the position of the *extra index* in the extra partition.

In addition to sampling M hints, the client also samples M backup hints. Looking ahead, backup hints are used to replenish a hint after each query. The client stores all the hints and backup hints in its local storage. Details of the offline phase can be found in algorithm 1.

Online. During the online phase, the client scans the hints one after another, looking for a hint whose *hint set* contains the query index q. The partition containing the query index is $s = \lfloor \frac{q}{\sqrt{N}} \rfloor$. Specifically, for hint k, the client checks if either the extra index or $r_{s,\mathrm{hints[k].id}}$ matches the query index q. The first such hint is selected. The details of the hint selection procedure are in algorithm 2.

Next, the selected hint is used to make a request to the server. The *request* consists of all the indices in the *hint set* except that the query index is replaced with the index from the query partition in the *hint set* of a backup hint (called the

Algorithm 4 The online algorithm for `PIR-PRIMARY`

Input: q - Query index
State: n - Next backup hint position, hints - Hints, bHints - Backup hints

Request
$s \leftarrow \lfloor \frac{q}{\sqrt{N}} \rfloor$
$t \leftarrow q \mod \sqrt{N}$
$(h,\ x) \leftarrow \mathsf{selectHint}(q, \mathrm{hints})$ ▷ Run algorithm 2
if $h = \perp$ **then**
  return $\perp$
end if
request $\leftarrow \mathsf{getRequestIndices}(h, \mathrm{bHints}[n].\mathrm{id}, s)$ ▷ Run algorithm 3
Send request to the server

Response
Receive response consisting of database entries at request indices from the server

Answer
$P = 0$
for $k \in [0, \sqrt{N} - 1]$ **do**
  if $k \neq s$ **then**
    $P \leftarrow P \oplus \mathrm{response}[k]$
  end if
end for
$\mathrm{DB_q} \leftarrow P \oplus \mathrm{h.parity}$

Update
$\mathrm{hints}[x] \leftarrow \mathrm{bHints}[n]$
$\mathrm{hints}[x].\mathrm{parity} \leftarrow \mathrm{hints}[x].\mathrm{parity} \oplus \mathrm{response}[s] \oplus \mathrm{DB_q}$
$\mathrm{hints}[x].\mathrm{eP} \leftarrow s$
$\mathrm{hints}[x].\mathrm{ePI} \leftarrow t$
$n \leftarrow n + 1$
return $\mathrm{DB_q}$

switch index). We call the database entry at the *switch index* as the *switch entry*. This switch is essential for privacy, as always including the query index in the request is clearly insecure.

Notice that the *switch index* was picked randomly and independently from the other indices in the *request*. Therefore, all the indices in the *request* are picked randomly and independently from each other and the query index. Hence, the server gains no information about the query index from the *request*. Construction of the *request* is described in the **Request** section of algorithm 4 abd illustrated in fig. 1

The client sends the *request* to the server and receives a *response* consisting of the entries at the *request* indices. The client computes the parity P of all the returned entries except the *switch entry*. Since the hint parity $hintP$ is the parity of all the entries in the *hint set*, the client recovers the queried entry $\mathsf{DB_q} = P \oplus hintP$. These steps are described in the **Reponse** and **Answer** section of algorithm 4.

Now, the client replaces the used hint with the next

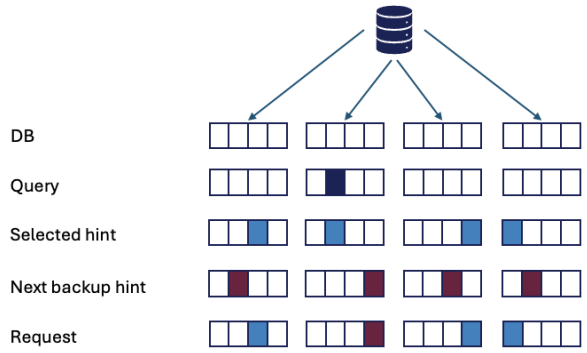

Figure 1: An illustration of how a PIR-PRIMARY *request* is constructed for a database with $N = 16$. The colored cells in each row depict the indices selected for the corresponding purpose.

Algorithm 5 The offline algorithm for BALANCED-PIR

Input: α, β - Storage parameters, γ - Frequency parameter
State: c - Cache, u - Query counter

if $u \equiv 0 \mod \alpha\sqrt{N}$ **then** ▷ Offline of PIR-PRIMARY
    Run algorithm 1 with $M = \alpha\sqrt{N}$
    Clear c
end if
if $u \equiv 0 \mod \gamma\alpha\sqrt{N}$ **then** ▷ Offline of PIR-BACKUP
    Run the offline algorithm in section 3.2 with $M = (\beta + 1)\sqrt{N}$
    $u \leftarrow 0$
end if

backup hint, which we call the *replenished hint*. The query index must be a part of the *hint set* of the replenished hint to ensure that the hint distribution remains identical before and after the query, which is crucial for privacy. This is accomplished by setting the extra index to be the query index in the replenished hint. The *extra index* overwrites the index selected by the PRF from the corresponding partition. To facilitate the overwrite, the parity of the replenished hint needs to be adjusted by adding the queried entry and removing the *switch entry*. Note that the *switch index* was a part of the *request* and hence the *switch entry* is in the *response*. Since the client knows both the queried entry and the *switch entry*, it can adjust the parity without further interaction with the server. Details regarding hint replenishment are described in the **Update** section of algorithm 4.

3.4. Putting it together

We now have two PIR schemes: PIR-PRIMARY requires large client storage to guarantee correctness but performs less computation, and PIR-BACKUP uses less client storage but requires more computation. Our construction

Algorithm 6 The online algorithm for BALANCED-PIR

Input: q - Query Index, α, β - Storage parameters, γ - Frequency parameter
State: cache - Cache, u - Query counter

if q in cache.keys **then**
    Make a *dummy request*
    return cache$[q]$
end if
$a \leftarrow$ Run algorithm 4 with q ▷ Make PIR-PRIMARY query
if $a \neq \perp$ **then**
    cache$[q] \leftarrow a$
    return a
else ▷ Make PIR-BACKUP query as PIR-PRIMARY failed
    $b \leftarrow$ Run the online algorithm in section 3.2 with q
    cache$[q] \leftarrow b$
    return b
end if
$u \leftarrow u + 1$
if $u \equiv 0 \mod \alpha\sqrt{N}$ **then**
    Run offline algorithm of BALANCED-PIR in algorithm 5
end if

of BALANCED-PIR combines the two PIR schemes in a secure manner while simultaneously ensuring efficiency.

Recall that α and β are storage parameters of PIR-PRIMARY and PIR-BACKUP, respectively. Additionally, γ is an integer parameter that determines the frequency of PIR-BACKUP offline phases: we will have one PIR-BACKUP offline phase per γ PIR-PRIMARY offline phases. Details on the selection of α, β and γ can be found in section 3.6.

During every PIR-PRIMARY offline phase, the client samples $\alpha\sqrt{N}$ hints for PIR-PRIMARY and during every PIR-BACKUP offline phase, the client samples hints according to a hint table with $(\beta + 1)\sqrt{N}$ columns. Moreover, the offline phase of PIR-PRIMARY is executed every $\alpha\sqrt{N}$ queries and the offline phase of PIR-BACKUP is executed every $\gamma\alpha\sqrt{N}$ queries. The offline algorithm of BALANCED-PIR is in algorithm 5.

In the online phase, every time the client has to retrieve a database entry, it attempts to find a hint for PIR-PRIMARY and use PIR-PRIMARY to make its query. If the client fails to find a suitable hint, it falls back to using PIR-BACKUP. The online algorithm of BALANCED-PIR is in algorithm 6.

The client caches the retrieved entries, and if a query is made for any of the cached entries, the client makes a *dummy request* (a *request* consisting of one random index from each partition) and returns the entry from the cache. This ensures query indices are unique, which is important in determining the client storage in our proof in section B.

3.5. Correctness and Privacy Analysis

Lemma 1. *In expectation, at most 1 in e^α queries of* BALANCED-PIR *uses* PIR-BACKUP*.*

Proof. In each query of BALANCED-PIR, the client first tries to use PIR-PRIMARY by finding a suitable hint. Each hint of PIR-PRIMARY consists of $\sqrt{N}$ random indices. Therefore, the probability that the query index is not selected by a hint is $1 - \frac{1}{\sqrt{N}}$. Since the client has $\alpha\sqrt{N}$ hints sampled, the probability that none of the hints select the query index is $(1 - \frac{1}{\sqrt{N}})^{\alpha\sqrt{N}}$. This value can be upper bounded in the following manner using a standard inequality

$$(1 - \frac{1}{\sqrt{N}})^{\alpha\sqrt{N}} < \frac{1}{e^\alpha}$$

Since the client uses PIR-BACKUP when it fails to find any PIR-PRIMARY hint, in expectation, the client uses PIR-BACKUP at most once every e^α queries. □

Theorem 2 (Correctness)**.** *For any sequence of queries between a client and a server using* BALANCED-PIR*, the client always retrieves all the entries at the query indices, except with negligible probability.*

Proof. In each query of BALANCED-PIR, the client first tries to use PIR-PRIMARY to execute the query. If the client is unable to find a suitable hint, the client uses PIR-BACKUP. PIR-BACKUP always succeeds as long as its hints have not been depleted, as shown in the work of Wang and Ren [2]. Therefore, to prove the correctness of the scheme, it suffices to show that the client is always able to find a PIR-BACKUP hint whenever it needs one.

Recall that α is the PIR-PRIMARY storage parameter, β is the PIR-BACKUP storage parameter, and γ is the PIR-BACKUP offline frequency parameter, that is, the client executes the offline phase of PIR-BACKUP every $m = \gamma\alpha\sqrt{N}$ queries. We will upper bound the probability that the client fails to find a PIR-BACKUP hint as a function of α, β, and γ.

Let the random variable X_i be an indicator random variable defined as follows.

$$X_i = \begin{cases} 1, & \text{Client uses PIR-BACKUP for query } i \\ 0, & \text{Client uses PIR-PRIMARY for query } i \end{cases} \quad (1)$$

The random variable $X = \sum_{i=1}^{m} X_i$ captures the number of times PIR-BACKUP is executed during the m queries.

In section B, we show that the indicator random variables $X_1, X_2, X_3, \ldots, X_m$ are negatively correlated. Hence, we can apply the Chernoff bound on X as shown in the work of Doerr [35]. Using lemma 1, we have that the mean of the random variable X is $\mu = \frac{m}{e^\alpha} = \frac{\gamma\alpha\sqrt{N}}{e^\alpha}$. We set $(1+\delta)\mu = \beta\sqrt{N}$ for some $\delta > 0$. The probability that the client fails to find a PIR-BACKUP hint for any query is bounded by the Multiplicative Chernoff bound on the upper tail of X

$$\Pr[X > \beta\sqrt{N}] = \Pr[X \geq (1+\delta)\mu] \leq e^{\left(-\frac{\delta^2\mu}{2+\delta}\right)} \quad (2)$$

where $\mu = \frac{\gamma\alpha\sqrt{N}}{e^\alpha}$ and $\delta = \frac{\beta\sqrt{N}}{\mu} - 1$.

We will select α, β, and γ such that $e^{\left(-\frac{\delta^2\mu}{2+\delta}\right)}$ is negligible, so BALANCED-PIR satisfies correctness. A practical set of parameters that satisfy the constraints is given in section 3.6. □

In order to prove the privacy of BALANCED-PIR, we first define a variant of PIR-PRIMARY called PIR-PRIMARY-w/DUMMY. In this variant, for each query, if the client is unable to find a suitable hint in PIR-PRIMARY, the client makes a dummy *request* to the server. A *dummy request* consists of one random index chosen from every partition. Additionally, the scheme uses random functions instead of pseudorandom functions.

Lemma 3. *PIR-PRIMARY-w/DUMMY satisfies the privacy definition from section 2.2.*

The privacy proof of PIR-PRIMARY-w/DUMMY follows an identical pattern to the privacy proof of Ren et al. [1]. Hence, we will give a sketch of the proof here and refer the reader to lemma 8 in the appendix for full details.

Proof sketch. For any PIR scheme to satisfy privacy according to the definition of section 2.2, the set of *requests* made by the client should not reveal any information about the query indices.

First query. Let us consider the first query. The client selects a hint whose *hint set* contains the query index along with randomly sampled indices from other partitions. The *request* consists of the indices in the *hint set*, except that the query index is replaced with the *switch index*, which is sampled independently and uniformly at random for the backup hint. Therefore, the set of indices in the *request* are sampled randomly from their partitions and are independent of the query index. Hence, the first *request* reveals no information about the query index.

Subsequent queries. To extend this argument to a sequence of queries, we need to show that the hint distribution, conditioned on the adversary's view, remains the same before and after each query. Then, our argument for the first query can be applied to all subsequent queries.

At a high level, the distribution of the hints conditioned on the adversary's view remains unchanged due to the fact that the used hint is replaced by another hint with the same distribution. Specifically, both the used hint and the replenished hint contain the queried index plus random indices from other partitions. This is accomplished by replacing each index revealed to the server in the used hint with a random index from the same partition in the replenished hint. Furthermore, all the other hints remain the same and hence have the same distribution. Therefore, the privacy argument for the first query can be extended to the subsequent queries.

Hence, PIR-PRIMARY-w/DUMMY is secure. □

Theorem 4 (Privacy)**.** *BALANCED-PIR satisfies the privacy definition from section 2.2.*

Proof. We show that BALANCED-PIR is secure using a hybrid argument. Consider the following set of hybrids

describing the scheme executed by the client in the adaptive security game defined in section 2.2.

- H_0: The client uses the BALANCED-PIR protocol as described in algorithm 6.
- H_1: Same as H_0 except that the client uses random functions instead of pseudorandom functions.
- H_2: The client uses PIR-PRIMARY-w/DUMMY.
- H_3: The client simply sends a *dummy request* for each query.

The adversary's view in H_0 and H_1 are computationally indistinguishable due to the security guarantee of pseudorandom functions. The adversary's view in H_2 and H_3 are identical due to lemma 3.

The difference between H_1 and H_2 is that, on failing to find a suitable hint for PIR-PRIMARY the client uses PIR-BACKUP in H_1 and makes a dummy request in H_2. Due to the privacy of PIR-BACKUP as shown in Wang and Ren [2], the distribution of a PIR-BACKUP request is identical to the distribution of a dummy request. Thus, the adversary's view is identical in H_1 and H_2.

Therefore, the adversary's view in BALANCED-PIR (H_0) is computationally indistinguishable from the adversary's view in H_3, which is independent of the query indices. Thereby, BALANCED-PIR satisfies privacy. □

3.6. Efficiency

We describe how the storage parameters α, β and frequency parameter γ are chosen such that the amortized costs for both online and offline phases are minimized.

Online efficiency. Let the online computational cost of PIR-BACKUP be ν times worse than the online computational cost of PIR-PRIMARY. From lemma 1, we expect 1 in e^{α} BALANCED-PIR queries to use PIR-BACKUP. Hence, we choose α such that $e^{\alpha} > \nu$ to ensure that the amortized online computational cost of BALANCED-PIR is within a small factor of the online computational cost of PIR-PRIMARY.

Offline efficiency. Let the offline computational cost of PIR-BACKUP be σ times worse than the offline computational cost of PIR-PRIMARY. To ensure the amortized offline cost of BALANCED-PIR is within a small factor of the offline cost of PIR-PRIMARY, we set the PIR-BACKUP frequency parameter γ to be greater than σ.

Correctness. The chosen values of α, β and γ must satisfy the correctness constraint in the proof of theorem 2.

Practical setting. For database entry sizes up to 128 bytes, we observe through experiments that PIR-BACKUP's online phase is roughly $\nu = 700$ times slower than that of PIR-PRIMARY, and that PIR-BACKUP's offline phase is about $\sigma = 100$ times slower than that of PIR-PRIMARY, in a practical setting. Additionally, we set the correctness failure probability at 2^{-40}. Following the considerations described above, we set $\alpha = 7$, $\beta = 2.32$, and $\gamma = 300$.

4. Verifiable BALANCED-PIR

Recall that a verifiable stateful PIR scheme (defined in section 2.3) has the following procedures: i) the server commits to and publishes a *database digest*, ii) the client verifies the committed database against the published *database digest*, iii) during each query, the client verifies that the *answer* matches the database entry at the query index of the committed database, and iv) the client protests against the server and aborts its execution if it detects malicious behavior by the server.

4.1. High-level Ideas

We first augment PIR-PRIMARY and PIR-BACKUP to be verifiable. Then, we combine Verifiable PIR-PRIMARY and Verifiable PIR-BACKUP in a similar manner as in BALANCED-PIR.

The *database digest* in our scheme is simply a hash of all the entries. Due to the collision resistance of the hash, the server cannot find two distinct databases that have the same *database digest*. During the offline phase, the client computes the *database digest* locally while streaming the database. The client verifies that the computed *database digest* matches the published *database digest*. Additionally, for every hint the client samples, the client computes and stores a hash of all the entries selected by that hint. We call it the *hint hash*. During the online phase, for each query, the client verifies the *answer* by recomputing the hash of all the entries selected by a hint and checks if it matches the stored *hint hash*.

Selective-failure attacks. One main challenge in designing a verifiable PIR scheme is to achieve resilience against *selective-failure attacks* [36] while allowing the client to protest against the server and abort its execution soon after detecting malicious server behaviors. If not designed properly, the timing of the client's protest action may leak information about the client query indices.

In both Verifiable PIR-PRIMARY and Verifiable PIR-BACKUP, every *request* contains a set of database indices, and the *response* contains the entries at the *request* indices. In our schemes, the client checks all the entries in the *response*. Most of these checks are carried out during each online query. However part of the checks have to be deferred to the subsequent offline phase due to the nature of our PIR schemes. Therefore, if the client in our scheme detects malicious behavior during an online query, it protests against the server and aborts in the next offline phase; before then, it should keep making dummy queries. This way, the timing of the protest is fully predictable by the malicious server and leaks no information about the query indices.

While the above high-level ideas are applicable to both schemes, there are nuances that need to be taken care of, which we address in the following subsections.

4.2. Verifiable PIR-PRIMARY

As described in the high-level idea, at the end of the offline phase, the client has the *hint hash* of all hints.

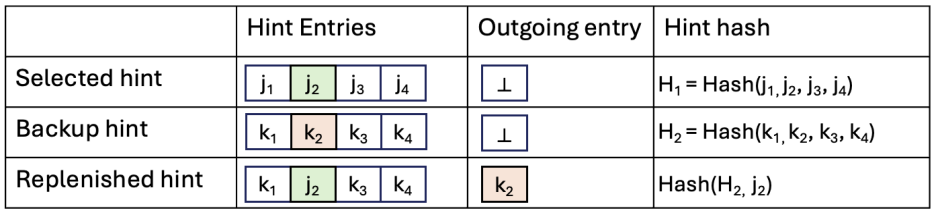

Figure 2: An illustration of hint replenishment during a query in Verifiable PIR-PRIMARY. The queried entry is j_2 and the *switch entry* is k_2. The replenished hint contains the same entries set as the backup hint, except that j_2 replaces k_2. The hint hash of the replenished hint is updated using the queried entry j_2 as depicted, and k_2 is stored as the *outgoing entry* with the replenished hint.

Recall the online protocol of PIR-PRIMARY. In Verifiable PIR-PRIMARY, we modify the hint replenishment procedure and add a verification procedure for each query.

We first give the intuition behind our hint replenishment strategy by describing the verification procedure for a special case. Then, we give the intuition behind our verification strategy.

When the *extra index* is not set in the hint, the client can perform the verification in a straightforward manner, as described in the general paradigm. That is, the client simply recomputes a hash of all the *hint entries* and verifies if it matches the stored *hint hash*. This verifies the integrity of the entries at the *hint set* and hence ensures the integrity of the *answer*.

At the end of each query, the backup hint replaces the used hint to become the replenished hint. The query index is added to the *hint set* of the replenished hint as the *extra index*, and the *switch index* is removed from the *hint set* of the replenished hint. Naturally, the *hint hash* of the replenished hint is updated using the entry at the *extra index*. Also note that the (now removed) *switch entry* was used to compute the *hint hash* of the replenished hint, so it will be needed for verification. Thus, we store the *switch entry* along with the replenished hint. These modifications ensure that all *hint entries* of the replenished hint and the *switch entry* can be verified correctly using the *hint hash*, if and when the replenished hint is used.

We now describe the details of Verifiable PIR-PRIMARY.

Offline. During the offline phase, the client verifies the committed database as described in the general paradigm. The client computes for every hint a *hint hash* as the hash of all the *hint entries*. The offline algorithm of Verifiable PIR-PRIMARY is described in algorithm 7. Key differences from the offline phase of PIR-PRIMARY are highlighted in yellow. Details regarding committed database verification are omitted in algorithm 7 for ease of exposition.

Online. In the hint replenishment procedure after each query, the client stores the *switch entry* alongside the replenished hint. The *switch entry* becomes the *outgoing entry* of the replenished hint. The *hint hash* of the replenished hint is updated as the hash of the *hint hash* and the entry at *extra index* (which is the queried entry). This process is depicted in fig. 2.

The verification procedure is as follows. At the end of each query, the client receives from the server all the *hint entries* of the selected hint and, if the extra index is set, the client has the *outgoing entry* stored locally. The client recomputes the hash of these entries and verifies if it matches with the *hint hash*. Details regarding the online phase of Verifiable PIR-PRIMARY can be found in algorithm 8. Key differences from the online phase of PIR-PRIMARY are highlighted in yellow.

Resilience against selective-failure attacks. At the end of an online phase, there may be replenished hints that have not been used. The *outgoing entries* of these replenished hints have not been verified. In the subsequent offline phase, while the client streams the database, the client verifies the integrity of these entries. Finally, if either the verification during any of the queries fails or the verification in the offline phase fails, the client protests against the server at the end of the offline phase.

4.3. Verifiable PIR-BACKUP

Offline. During the offline phase, the client verifies the committed database as described in the general paradigm. In PIR-BACKUP the client samples a hint table, and for each column in the table, the client stores the combined parity of entries at indices in the column. Additionally, in Verifiable PIR-BACKUP, the client also computes a *hint hash* as the hash of entries at the indices in each column of the hint table.

Online. Recall the high level overview of the online phase of PIR-BACKUP presented in section 3.2. We first describe how the *hint hash* of columns are updated after each query, and then our verification procedure.

At the end of each query, as each index from the hint column is moved to a pseudorandom column within its row, the *hint hash* of the destination column is updated as the hash of the current *hint hash* of the destination column and the entry at the index being added to the column.

Verification of the *answer* during each query is carried out as follows. After the client has retrieved all entries at indices in the hint column, the client computes a hash of these entries and verifies that it matches the *hint hash*. Importantly, the client must first compute the order in which indices are added to the hint column and then compute the hash of entries at the indices in the determined order.

Algorithm 7 The offline algorithm for Verifiable PIR-PRIMARY

Input: α - Hint storage factor
State: n - Next backup hint position, hints - Hints, bHints - Backup Hints, v_o - Online verification status, v_p - Offline verification status

$M \leftarrow \alpha\sqrt{N}$
sHints $\leftarrow []$
for $k \in [0, 2M-1]$ **do**
  $\text{hint}_\text{k} \leftarrow$ { id: k, parity: 0, eP: $\perp$, ePI: $\perp$, oEntry: $\perp$, hash: 0 }
  sHints.push(hint_k)
end for
for $j \in [0, \sqrt{N}-1]$ **do**
  Stream partition $\mathsf{DB}[j\sqrt{N} : (j+1)\sqrt{N}-1]$ from the server
  $v_{p_j} \leftarrow$ Verify oEntry from partition j in the hints from the previous online phase
  if v_{p_j} = fail **then**
    $v_p \leftarrow$ fail
  end if
  for $k \in [0, 2M-1]$ **do**
    $r_{j,k} \leftarrow \mathsf{PRF}(j \parallel k) \mod \sqrt{N}$
    $x \leftarrow \mathsf{DB}[r_{j,k} + j\sqrt{N}]$
    sHints[k].parity $\leftarrow$ sHints[k].parity $\oplus\ x$
    sHints[k].hash $\leftarrow$ Hash(sHints[k].hash, x)
  end for
end for
if v_p = fail or v_o = fail **then**
  return $\perp_v$
end if
hints $\leftarrow$ sHints$[0, M-1]$
bHints $\leftarrow$ shints$[M, 2M-1]$
$n \leftarrow 0$

Resilience against selective-failure attacks. Recall that in each PIR-BACKUP *request*, the cells marked as $\perp$ in the hint column and the query index are switched to random indices from their respective partitions. The entries at the random indices are not used in the computation of the *hint hash* of the column and thereby cannot be verified during each query. Therefore, the client defers the verification of these entries to the subsequent offline phase.

Specifically, the client maintains a hash of all the random entries in each row, called the *row hash*, where the random indices are sampled using a PRF. During the subsequent offline phase, the client verifies each *row hash* as it streams the database. If either the verification during any query fails or if the verification of any *row hash* fails, the client protests against the server.

4.4. Putting it together

We combine Verifiable PIR-PRIMARY and Verifiable PIR-BACKUP to construct Verifiable BALANCED-PIR in the same manner in which PIR-PRIMARY and PIR-BACKUP are combined in BALANCED-PIR.

During every Verifiable PIR-PRIMARY offline phase, the client executes the offline verification of both Verifiable PIR-PRIMARY and Verifiable PIR-BACKUP. (But the offline hint building of PIR-BACKUP still happens once per γ PIR-PRIMARY offline phases.) If the verification fails, the

Algorithm 8 The online algorithm for Verifiable PIR-PRIMARY

Input: q - Query index
State: n - Next backup hint position, hints - Hints, bHints - Backup hints, v_o - Online verification status

Request
if v_o = fail **then**
  Make a *dummy request*
  return
end if
Execute steps in **Request** of algorithm 4

Reponse
Execute steps in **Reponse** of algorithm 4

Answer
Execute steps in **Answer** of algorithm 4

Verify
$d \leftarrow 0$
for $j \in [0, \sqrt{N}-1]$ **do**
  if h.eP $\neq \perp$ and $j = h$.eP **then**
    $d \leftarrow \text{Hash}(d, h.\text{oEntry})$
  else if $j \neq s$ **then**
    $d \leftarrow \text{Hash}(d, \text{response}[j])$
  else
    $d \leftarrow \text{Hash}(d, \text{DB}_\text{q})$
  end if
end for
if h.eP $= s$ **then**
  $d \leftarrow \text{Hash}(d, \text{DB}_\text{q})$
else if h.eP $\neq \perp$ **then**
  $d \leftarrow \text{Hash}(d, \text{response}[h.\text{eP}])$
end if
if $d \neq h$.hash **then**
  $v_o \leftarrow$ fail
end if

Update
hints$[x] \leftarrow$ bHints$[n]$
hints$[x]$.parity $\leftarrow$ hints$[x]$.parity $\oplus$ response$[s] \oplus \text{DB}_\text{q}$
hints$[x]$.eP $\leftarrow s$
hints$[x]$.ePI $\leftarrow t$
hints$[x]$.oEntry $\leftarrow$ response$[s]$
hints$[x]$.hash $\leftarrow \text{Hash}(\text{hints}[x].\text{hash}, \text{DB}_\text{q})$
$n \leftarrow n+1$
return DB_q

client protests and aborts. If the online verification of any query fails, the client makes dummy *requests* (one random index from each partition) until the next PIR-PRIMARY offline phase is reached, and then protests and aborts.

4.5. Completeness, Soundness, and Privacy

Theorem 5 (Completeness)**.** *Verifiable BALANCED-PIR satisfies completeness as defined in section 2.3.*

Proof. The completeness of Verifiable PIR-PRIMARY is straightforward from the construction of the scheme and the correctness of PIR-PRIMARY. The completeness of Verifiable PIR-BACKUP will also be straightforward provided the client can compute the order in which indices get added to the hint column of each query. We show this is indeed the case in lemma 13 of the appendix to complete the proof of completeness of Verifiable BALANCED-PIR. □

The proof of lemma 13 is deferred to the appendix since this requires understanding some of the inner workings of the scheme of Wang and Ren [2], which we provide in section C. We note that for all the other constructions and proofs in this work, the scheme of Wang and Ren can be treated as a black box.

Theorem 6 (Soundness)**.** *Verifiable BALANCED-PIR satisfies soundness as defined in section 2.3.*

Proof. In both Verifiable PIR-PRIMARY and Verifiable PIR-BACKUP, the queried entry is a part of the input used to compute the *hint hash*, so the soundness follows from the collision resistance (section 2.4) of the hash function. □

Theorem 7 (Privacy)**.** *Verifiable BALANCED-PIR is secure under the privacy definition in section 2.3.*

Proof. Verifiable BALANCED-PIR does not alter the request format and distribution of BALANCED-PIR, so the *requests* in Verifiable BALANCED-PIR leaks no information about the query indices following theorem 4.

We need to show that if the client protests, the timing of the protest leaks no information about the query indices.

From the construction of Verifiable BALANCED-PIR, the client checks if every *response* entry matches its corresponding *request* index, partly during the offline phase and partly during the online phase. Then, the client protests and aborts in the offline phase. If any online check fails, the client makes *dummy requests* (which follow the distribution of *requests* as shown in theorem 4) and then aborts during the offline phase. Therefore, the timing of the protest is predictable from the *requests* and *responses* and is independent of the query indices.

Since neither the distribution of *requests* nor the timing of protest leaks information about the query indices, Verifiable BALANCED-PIR is secure. □

5. Evaluation

5.1. Implementation Details

Our BALANCED-PIR scheme and verifiable BALANCED-PIR scheme have been implemented using C++. The code is available at BALANCED-PIR. The implementation of PIR-PRIMARY and Verifiable PIR-PRIMARY is close to 500 lines of code and the implementation of PIR-BACKUP and Verifiable PIR-BACKUP is close to 800 lines of code.

We use the AES implementation from CryptoPP to instantiate the PRF or collision-resistant hash function. Additionally, PIR-BACKUP requires a small-domain PRP. We implement the construction of Morris and Rogaway [31].

5.2. Experimental Setup

Baseline. We compare BALANCED-PIR with the following single-server stateful PIR schemes:

- SimplePIR by Henzinger et al [37]: In this scheme, the server computation is linear in the size of the database. Unlike our schemes and the other schemes in the baselines, the offline phase of SimplePIR is executed only once. We use their Go implementation.[1]
- The scheme of Ren et al. [1]: The amortized communication and amortized computation of their scheme are both $O(\sqrt{N})$, whereas the client storage is $O(\lambda\sqrt{N})$ with λ set to 80 in practice. We use their C++ implementation. [2]
  For a fair comparison with BALANCED-PIR, we also consider a different parametrization of their scheme that uses smaller storage but blows up computation and communication.
- The scheme by Wang and Ren (WR PIR) [2]: This scheme has a client storage of $O(\sqrt{N})$ and the online server computation of $O(\sqrt{N})$. However, the scheme requires client evaluations of an expensive cryptographic primitive called small-domain PRP, which blows up the computational cost. Since no implementation of the scheme is available, we implement the scheme in C++.

We compare Verifiable BALANCED-PIR with the following verifiable stateful PIR scheme:

- VeriSimplePIR by de Castro and Lee [3]: This scheme builds on a verifiable PIR scheme of Colombo et al. [32] by removing the honest-digest assumption (see section 6). We use their C++ implementation.[3]

Execution environment. We run all our experiments with a single thread on an AWS m5.8x large instance, which operates on a 3.1 GHz Intel Xeon processor equipped with 128 GB RAM.

1. https://github.com/ahenzinger/simplepir
2. https://github.com/renling/S3PIR
3. https://github.com/leodec/VeriSimplePIR

TABLE 2: Comparison of single-server stateful PIR schemes.

	Database Parameters	Client Storage (MB)	Offline Comm. (MB)	Offline Compute (s)	Online Comm. (KB)	Online Compute (ms)	Amortized per query Comm. (KB)	Amortized per query Compute (ms)
SimplePIR [37]		10.6	10.6	1.38	20	7.43	20	7.43
WR PIR [2]	2^{20} 8-byte entries	0.02	8	29.7	10	77.23	18	78.44
Ren et al. [1]	8 MB in total	2.5	8	3.93	2.14	0.13	2.34	0.22
Ren et al. [1] normalized		0.2	8	7.89	34	1	35.6	1.77
BALANCED-PIR		0.2	8	0.2	10	0.17	11.1	0.2
SimplePIR [37]		20.9	20.9	4.8	40	14	40	14
WR PIR [2]	2^{20} 32-byte entries	0.07	32	29.74	34	85.79	66	87
Ren et al. [1]	32 MB in total	6.25	32	4	2.19	0.14	2.99	0.25
Ren et al. [1] normalized		0.55	32	8.34	34.1	1.06	40.5	1.87
BALANCED-PIR		0.6	32	0.23	34	0.19	38.6	0.22
SimplePIR [37]		42.3	42.3	35.8	84	39	84	39
WR PIR [2]	2^{24} 8-byte entries	0.08	128	494	40	356	72	361
Ren et al. [1]	128 MB in total	10	128	64.4	8.52	0.51	9.32	0.90
Ren et al. [1] normalized		0.78	128	126	136	4.11	142	7.18
BALANCED-PIR		0.8	128	3.29	40	0.75	44.6	0.87
SimplePIR [37]		86.8	86.8	154	168	103	168	103
WR PIR [2]	2^{24} 32-byte entries	0.27	512	495	136	356	264	362
Ren et al. [1]	512 MB in total	25	512	65	8.56	0.62	11.76	1
Ren et al. [1] normalized		2.19	512	136	136	4.33	167	7.66
BALANCED-PIR		2.42	512	3.8	136	0.87	154	1
SimplePIR [37]		173	173.4	623	338	319	338	319
WR PIR [2]	2^{28} 8-byte entries	0.31	2048	8227	160	1640	288	1661
Ren et al. [1]	2048 MB in total	40	2048	989	34	2.4	37.2	3.9
Ren et al. [1] normalized		3.12	2048	2009	544	17.2	570	29.4
BALANCED-PIR		3.18	2048	53.58	160	3.3	178	3.77
SimplePIR [37]		353	353	2788	688	1123	688	1123
WR PIR [2]	2^{28} 32-byte entries	1.06	8192	8249	544	1644	1056	1665
Ren et al. [1]	8192 MB in total	100	8192	1146	34.1	2.7	46.9	4.5
Ren et al. [1] normalized		8.75	8192	2154	544	17.7	646	30.8
BALANCED-PIR		9.67	8192	67.15	544	3.85	617	4.43

TABLE 3: Comparison of verifiable single-server stateful PIR schemes.

	Database Parameters	Client Storage (MB)	Offline Comm. (MB)	Offline Compute (s)	Online Comm. (KB)	Online Compute (ms)	Amortized per query Comm. (KB)	Amortized per query Compute (ms)
VeriSimplePIR [3]	2^{20} 8-byte entries	32	71.5	51.6	32	22	32	22
Verifiable BALANCED-PIR	8 MB in total	0.27	8	5.16	10	0.61	11.3	1.45
VeriSimplePIR [3]	2^{24} 8-byte entries	128	290	796	128	136	128	136
Verifiable BALANCED-PIR	128 MB in total	1.09	128	86.5	40	2.57	45.3	6.09
VeriSimplePIR [3]	2^{28} 8-byte entries	512	1178	12553	512	1336	512	1336
Verifiable BALANCED-PIR	2048 MB in total	4.37	2048	1326	160	9.96	181	23.46

Experimental setup. We run BALANCED-PIR, Verifiable BALANCED-PIR, and all the baselines on databases with 2^{20}, 2^{24} and 2^{28} entries. The experiments on BALANCED-PIR and its baselines are run with both 8-byte and 32-byte entries. The experiments on Verifiable BALANCED-PIR and its baseline are run only on 8-byte entries since the execution of VeriSimplePIR failed for 32-byte entries (and we expect the comparison to be similar for larger entry sizes.)

5.3. Evaluation Results

Single-server stateful PIR scheme. In table 2 we compare various stateful single-server PIR schemes with BALANCED-PIR. We present the communication and computation costs of both the offline phase and the online phase. We also compute and present the amortized communication and computation costs of each scheme as the offline cost divided by the number of maximum queries possible during each online phase, plus the online cost.

For each metric, we highlight the scheme that performs the best on the metric in green and the schemes that are two or more orders worse than the best scheme on that metric in red. Notice that BALANCED-PIR has no metric colored red. We highlight only the first row as the same pattern applies to all rows.

SimplePIR requires $35\times$ to $50\times$ larger client storage than our BALANCED-PIR scheme. The offline phase of SimplePIR is executed only once and hence does not contribute to the amortized cost. But the amortized computation of their scheme is $37\times$ to $63\times$ worse than that of BALANCED-PIR due to their linear computation. The

amortized communication cost of SimplePIR is comparable to that of BALANCED−PIR. We note that the amortized costs of SimplePIR do not include the offline cost since the offline phase is executed only once. This allows for a fair comparison with the other schemes.

Unlike SimplePIR, all the other PIR schemes presented in table 2 require periodic executions of the offline phase. The offline phase of BALANCED−PIR requires the least amount of computation among all the schemes. Compared to BALANCED−PIR, the offline computation of WR PIR is $120\times$ to $148\times$ more expensive; the offline computation of Ren et al. is $16\times$ to $23\times$ more expensive if we normalize the client storage (and is still $32\times$ to $40\times$ more expensive, even with more client storage.)

The WR PIR scheme uses the least amount of client storage. BALANCED−PIR needs $8.5\times$ to $10\times$ of the client storage of the WR scheme. However, the amortized computation of WR PIR is $375\times$ to $492\times$ more expensive than BALANCED−PIR.

The original scheme of Ren et al. needs $10.5\times$ to $12.5\times$ more client storage than BALANCED−PIR. Using the extra storage, their scheme can achieve $4.5\times$ to $12.5\times$ advantage in terms of amortized communication. If we normalize their scheme to use a similar amount of client storage as our BALANCED−PIR scheme, then our scheme will be $7\times$ to $8.5\times$ in amortized computation and $1.05\times$ to $3.2\times$ better in amortized communication, when the entry size ranges between 8 to 32 bytes.

We emphasize that as the entry size increases, the amortized communication of BALANCED−PIR increases faster than Ren et al. normalized. For example, if the entry size is 128 bytes (not shown in the table), then BALANCED−PIR would be $2.5\times$ worse in amortized communication while amortized computation is still $7\times$ better and the offline computation is $30\times$ better.

Verifiable single-server stateful PIR schemes. In comparison with VeriSimplePIR, our Verifiable BALANCED−PIR scheme is $117\times$ better in client storage, $15\times$ to $56\times$ better in amortized computation, and $2.9\times$ better in amortized communication. The client storage of VeriSimplePIR is significantly larger because their scheme builds on the semi-honest SimplePIR [37], which already requires large client storage, and adds substantial additional client storage for verification.

6. Related Work

Single-server stateful PIR schemes. Stateful private information retrieval in the single-server setting was first presented in the work of Patel et al. [38]. The scheme still requires $O(N)$ online computation, same as stateless schemes. SimplePIR and FrodoPIR [39], [37] are two other stateful PIR schemes that use $O(N)$ online computation. They have the advantage that the offline phase is executed only once.

The PIR scheme by Corrigan-Gibbs et al. [25] was the first single-server stateful PIR scheme with amortized sublinear communication and computation. Several schemes [26], [40] use an advanced cryptographic primitive called privately puncturable PRFs to improve amortized communication. However, there are no known practical instantiations of privately puncturable PRFs, and these schemes are only of theoretical interest.

The works of Zhou et al. [27] and Ren et al. [1] are state-of-the-art practical schemes. They achieved $O(\sqrt{N})$ amortized communication and $O(\sqrt{N})$ amortized computation. Both schemes require $O(\lambda\sqrt{N})$ client storage where λ is a security parameter for correctness and is typically set to be 80.

A recent PIR scheme presented by Wang and Ren [2] requires $O(\sqrt{N})$ client storage and $O(\sqrt{N})$ online server computation, significantly reducing the amount of client storage required and meeting a lower bound by Yeo [41]. However, the amortized computation of the scheme is impractical since the client needs to evaluate $\sqrt{N}$ small-domain PRPs during each query.

Ghoshal et al. [30] further improves communication to $O(N^{\frac{1}{4}})$ per query. However, the improvement in communication comes at the cost of significant blowups to client storage and offline computation, especially for smaller entry sizes. Fisch et al. [29] gives a method to achieve sublinear offline bandwidth, but a powerful server (e.g., hundreds of thousands of GPUs) is required for the end-to-end preprocessing time to be faster than prior work.

Verifiable single-server stateful PIR schemes. Colombo et al. [32] is the first to propose and achieve what we consider the natural definition of verifiability in PIR schemes: the server commits to a database beforehand, and the retrieved entry matches the queried entry in the committed database. (Earlier works on verifiable PIR [42], [43], [44] consider different or weaker notions of verifiability.) Colombo et al. [32] gave two schemes in the single-server setting. The more practical one is based on SimplePIR [37]. One drawback of their work is that they make an *honest-digest assumption*, i.e., the commitment (digest) of the database is assumed to be computed faithfully.

De Castro and Lee [3] gave a verifiable version of SimplePIR without relying on the honest-digest assumption. This is the state-of-the-art verifiable single-server PIR scheme and our baseline. We note that Verifiable BALANCED−PIR performs better on all metrics than both versions of Verifiable SimplePIR above, as well as the original (unverifiable) SimplePIR.

The other single-server construction in Colombo et al. [32] is based on the Decisional Diffie–Hellman (DDH) assumption. This scheme involves prohibitive computation: it takes the client one hour to retrieve a single bit from a 1 GB database. Dietz and Tessaro [33] remove the honest-digest assumption from the DDH-based scheme, but still require large server computation.

7. Conclusion

We have presented BALANCED−PIR, a single-server stateful PIR scheme with moderate client storage and

efficient amortized computation. Unlike prior works, BALANCED-PIR has effectively balanced computation and client storage. BALANCED-PIR inherits some of the drawbacks of earlier works, such as periodic executions of the offline phase and $O(\sqrt{N})$ request size. We also presented Verifiable BALANCED-PIR, which is the first verifiable stateful PIR scheme with sublinear amortized computation and small client storage.

Acknowledgments. This work is funded in part by the National Science Foundation Award #2246386 and the Google Research Scholar Program. This work used AWS through the CloudBank project [45], which is supported by National Science Foundation grant #1925001.

References

[1] L. Ren, M. H. Mughees, and I. Sun, "Simple and practical amortized sublinear private information retrieval using dummy subsets," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 1420–1433.

[2] Z. Wang and L. Ren, "Single-server client preprocessing pir with tight space-time trade-off," *Cryptology ePrint Archive*, 2024.

[3] L. de Castro and K. Lee, "{VeriSimplePIR}: Verifiability in {SimplePIR} at no online cost for honest servers," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5931–5948.

[4] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *Journal of the ACM (JACM)*, vol. 45, no. 6, pp. 965–981, 1998.

[5] P. Mittal, F. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg, "Pir-tor: Scalable anonymous communication using private information retrieval," in *20th USENIX security symposium*, 2011.

[6] A. H. Kwon, D. Lazar, S. Devadas, and B. Ford, "Riffle: An efficient communication system with strong anonymity," 2015.

[7] S. Angel and S. Setty, "Unobservable communication over fully untrusted infrastructure," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 551–569.

[8] S. B. Mane, S. T. Sawant, and P. K. Sinha, "Using private information retrieval protocol for an e-commerce application," in *Proceedings of the CUBE International Information Technology Conference*, ser. CUBE '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 659–663. [Online]. Available: https://doi.org/10.1145/2381716.2381842

[9] Y. Huang and I. Goldberg, "Outsourced private information retrieval," ser. WPES '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 119–130. [Online]. Available: https://doi.org/10.1145/2517840.2517854

[10] J. Bringer, H. Chabanne, D. Pointcheval, and Q. Tang, "Extended private information retrieval and its application in biometrics authentications," in *International Conference on Cryptology and Network Security*. Springer, 2007, pp. 175–193.

[11] M. Vries, "Private information retrieval applied to biometric verification," Master's thesis, University of Twente, 2022.

[12] A. Juels, "Targeted advertising... and privacy too," in *Cryptographers' Track at the RSA Conference*. Springer, 2001, pp. 408–424.

[13] M. Backes, A. Kate, M. Maffei, and K. Pecina, "Obliviad: Provably secure and practical online behavioral advertising," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 257–271.

[14] M. Green, W. Ladd, and I. Miers, "A protocol for privately reporting ad impressions at scale," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 1591–1601.

[15] S. Servan-Schreiber, K. Hogan, and S. Devadas, "Adveil: A private targeted advertising ecosystem," *Cryptology ePrint Archive*, 2021.

[16] T. Gupta, N. Crooks, W. Mulhern, S. Setty, L. Alvisi, and M. Walfish, "Scalable and private media consumption with popcorn," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 91–107.

[17] C. A. Melchor, J. Barrier, L. Fousse, and M.-O. Killijian, "Xpir: Private information retrieval for everyone," *Proceedings on Privacy Enhancing Technologies*, pp. 155–174, 2016.

[18] S. Angel, H. Chen, K. Laine, and S. Setty, "Pir with compressed queries and amortized query processing," in *2018 IEEE symposium on security and privacy (SP)*. IEEE, 2018, pp. 962–979.

[19] C. Gentry and S. Halevi, "Compressible fhe with applications to pir," in *Theory of cryptography conference*. Springer, 2019, pp. 438–464.

[20] J. Park and M. Tibouchi, "Shecs-pir: somewhat homomorphic encryption-based compact and scalable private information retrieval," in *European symposium on research in computer security*. Springer, 2020, pp. 86–106.

[21] M. H. Mughees, H. Chen, and L. Ren, "Onionpir: Response efficient single-server pir," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2292–2306.

[22] S. J. Menon and D. J. Wu, "Spiral: Fast, high-rate single-server pir via fhe composition," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 930–947.

[23] H. Corrigan-Gibbs and D. Kogan, "Private information retrieval with sublinear online time," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2020, pp. 44–75.

[24] A. Lazzaretti and C. Papamanthou, "Treepir: Sublinear-time and polylog-bandwidth private information retrieval from ddh," in *Annual International Cryptology Conference*. Springer, 2023, pp. 284–314.

[25] H. Corrigan-Gibbs, A. Henzinger, and D. Kogan, "Single-server private information retrieval with sublinear amortized time," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2022, pp. 3–33.

[26] A. Lazzaretti and C. Papamanthou, "Near-optimal private information retrieval with preprocessing," in *Theory of Cryptography Conference*. Springer, 2023, pp. 406–435.

[27] M. Zhou, A. Park, W. Zheng, and E. Shi, "Piano: Extremely simple, single-server pir with sublinear server computation," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2023, pp. 55–55.

[28] A. Lazzaretti and C. Papamanthou, "Single pass {Client-Preprocessing} private information retrieval," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5967–5984.

[29] B. Fisch, A. Lazzaretti, Z. Liu, and C. Papamanthou, "Thorpir: single server pir via homomorphic thorp shuffles," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 1448–1462.

[30] A. Ghoshal, M. Zhou, and E. Shi, "Efficient pre-processing pir without public-key cryptography," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2024, pp. 210–240.

[31] B. Morris and P. Rogaway, "Sometimes-recurse shuffle: almost-random permutations in logarithmic expected time," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2014, pp. 311–326.

[32] S. Colombo, K. Nikitin, H. Corrigan-Gibbs, D. J. Wu, and B. Ford, "Authenticated private information retrieval," in *32nd USENIX security symposium (USENIX Security 23)*, 2023, pp. 3835–3851.

[33] M. Dietz and S. Tessaro, "Fully malicious authenticated pir," in *Annual International Cryptology Conference*. Springer, 2024, pp. 113–147.

[34] B. Cohen and B. Laurie, "Aes-hash," *NIST: Modes of Operation for Symmetric Key Block Ciphers*, 2001.

[35] B. Doerr, "Probabilistic tools for the analysis of randomized optimization heuristics," *Theory of evolutionary computation: Recent developments in discrete optimization*, pp. 1–87, 2020.

[36] E. Kushilevitz and R. Ostrovsky, "Replication is not needed: Single database, computationally-private information retrieval," in *Proceedings 38th annual symposium on foundations of computer science*. IEEE, 1997, pp. 364–373.

[37] A. Henzinger, M. M. Hong, H. Corrigan-Gibbs, S. Meiklejohn, and V. Vaikuntanathan, "One server for the price of two: Simple and fast single-server private information retrieval," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 3889–3905.

[38] S. Patel, G. Persiano, and K. Yeo, "Private stateful information retrieval," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1002–1019.

[39] A. Davidson, G. Pestana, and S. Celi, "Frodopir: Simple, scalable, single-server private information retrieval," *Proceedings on Privacy Enhancing Technologies*, 2023.

[40] M. Zhou, W.-K. Lin, Y. Tselekounis, and E. Shi, "Optimal single-server private information retrieval," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2023, pp. 395–425.

[41] K. Yeo, "Lower bounds for (batch) pir with private preprocessing," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2023, pp. 518–550.

[42] S. Ben-David, Y. T. Kalai, and O. Paneth, "Verifiable private information retrieval," in *Theory of Cryptography Conference*. Springer, 2022, pp. 3–32.

[43] X. Wang and L. Zhao, "Verifiable single-server private information retrieval," in *Information and Communications Security: 20th International Conference, ICICS 2018, Lille, France, October 29-31, 2018, Proceedings*. Springer, 2018, pp. 478–493.

[44] L. Zhao, X. Wang, and X. Huang, "Verifiable single-server private information retrieval from lwe with binary errors," *Information Sciences*, vol. 546, pp. 897–923, 2021.

[45] M. Norman, V. Kellen, S. Smallen, B. DeMeulle, S. Strande, E. Lazowska, N. Alterman, R. Fatland, S. Stone, A. Tan *et al.*, "Cloudbank: Managed services to simplify cloud access for computer science research and education," in *Practice and Experience in Advanced Research Computing 2021: Evolution Across All Dimensions*, 2021, pp. 1–4.

[46] D. P. Dubhashi and D. Ranjan, "Balls and bins: A study in negative dependence," *BRICS Report Series*, vol. 3, no. 25, 1996.

Appendix A. Privacy of `PIR-PRIMARY-w/DUMMY`

Lemma 8. *`PIR-PRIMARY-w/DUMMY` satisfies the privacy definition from section 2.2 (full proof).*

The privacy proof of `PIR-PRIMARY-w/DUMMY` follows an identical pattern to the privacy proof of Ren et al. [1].

Proof. For any PIR scheme to satisfy privacy according to the definition of section 2.2, the set of *requests* made by the client should not reveal any information about the query indices.

First query. See the proof for lemma 3.

Subsequent queries. To extend the above argument to a sequence of queries, we need to show that the hint distribution remains the same before and after each query, even when conditioned on the view of the adversary.

Let the random variable $H^i_{j,k}$ represent the index selected by the j-th hint from the k-th partition before the i-th query. Then,

$$\mathbf{H^i} = \begin{bmatrix} H^i_{0,0} & H^i_{0,1} & \dots & H^i_{0,\sqrt{N}-1} \\ H^i_{1,0} & H^i_{1,1} & \dots & H^i_{1,\sqrt{N}-1} \\ \vdots & \vdots & \ddots & \vdots \\ H^i_{M-1,0} & H^i_{M-1,1} & \dots & H^i_{M-1,\sqrt{N}-1} \end{bmatrix}$$

specifies the distribution of the set of hints before the i-th query. We define $\mathcal{Y}$ as the distribution of each hint and Y as the number of possibilities of each hint.

Similarly, we define the random variable $\tilde{\mathbf{H}}^{\mathbf{i}}$ which is the hint distribution conditioned on the view of the adversary up to and not including query i.

We will show that the hint distribution conditioned on the view of the adversary remains the same by using induction.

Base case: Since the indices in the *hint set* of each hint are picked independently and uniformly from their partitions, each row in $\mathbf{H^1}$ is i.i.d. and follows $\mathcal{Y}$. Therefore, $\Pr[\mathbf{H^1} = h] = \frac{1}{Y^M}$. Moreover, since the view of the adversary is empty, $\Pr[\mathbf{H^1} = h] = \Pr[\tilde{\mathbf{H}}^{\mathbf{1}} = h] = \frac{1}{Y^M}$.

Case i: $\tilde{\mathbf{H}}^{\mathbf{i}}$ and $\mathbf{H^1}$ are distributed identically.

Case $i+1$: During query i, the client attempts to find a suitable hint that contains the query index. If the client doesn't find any suitable hint, the client makes a *dummy request*, and hence, it is straightforward to see that $\tilde{\mathbf{H}}^{i+1}$ is distributed identically to $\tilde{\mathbf{H}}^{i}$.

Now we consider the case where the client finds a suitable hint. Let hint t be selected for the query and let the query index be in partition s. Then, all the indices in the *hint set* of hint t are sent to the server except the query index. Instead, the index from partition s of the next backup hint is sent to the server. Let the view of the *request* for the i-th query be denoted by the random variable $\mathbf{R}$. We denote the set of all possible *requests* as $\mathbf{S}$.

The probability that the random variable $\tilde{\mathbf{H}}^{i+1}$ takes a value h is as follows:

$$\begin{aligned} \Pr[\tilde{\mathbf{H}}^{i+1} = h] &= \Sigma_{r \in \mathbf{S}} \Pr[\tilde{\mathbf{H}}^{i+1} = h \mid \mathbf{R} = r] \cdot \Pr[\mathbf{R} = r] \\ &= \Sigma_{r \in \mathbf{S}} \frac{\Pr[\tilde{\mathbf{H}}^{i+1} = h, \mathbf{R} = r]}{\Pr[\mathbf{R} = r]} \cdot \Pr[\mathbf{R} = r] \\ &= \Sigma_{r \in \mathbf{S}} \Pr[\tilde{\mathbf{H}}^{i+1} = h, \mathbf{R} = r] \end{aligned}$$

For the event $\tilde{\mathbf{H}}^{i+1} = h, \mathbf{R} = r$ to occur, we need the following: i) all rows other than row t of h must match $\tilde{H}^i$, ii) indices from row t of h, other than the index from partition s, must match the corresponding indices in the next backup hint, iii) index from partition s of row t in h must match the query index, iv) row t of $\tilde{H}^i$ must match r except the index in partition s and v) index in partition s of r must match the index in partition s of the next backup hint. Additionally, the size of the set $\mathbf{S}$ is also Y. Therefore, using

the fact that $\tilde{\mathbf{H}}^i$ is identically distributed to $\mathbf{H}^1$, the above expression becomes:

$$\Pr[\tilde{\mathbf{H}}^{i+1} = h] =$$
$$Y \cdot (\frac{1}{Y^{M-1}} \cdot \frac{1}{\sqrt{N}^{\sqrt{N}-1}} \cdot \frac{1}{\sqrt{N}} \cdot \frac{1}{\sqrt{N}^{\sqrt{N}-1}} \cdot \frac{1}{\sqrt{N}})$$

Using the fact that $Y = \frac{1}{\sqrt{N}^{\sqrt{N}}}$, we have that:

$$\Pr[\tilde{\mathbf{H}}^{i+1} = h] = \frac{1}{Y^M}$$

which is the same as $\Pr[\mathbf{H}^1 = h]$.

Since the hint distribution conditioned on the view of the adversary remains the same, the security analysis for the first query can be extended to subsequent queries. Hence, PIR-PRIMARY-w/DUMMY is secure. □

Appendix B. Correctness of BALANCED-PIR

Recall the details on the correctness of BALANCED-PIR in theorem 2. The proof requires that the random variables $X_1, X_2, \ldots, X_m$ are negatively correlated. In this section, we prove that this is indeed the case.

Specifically, we can show that X_1, X_2, X_3, ... ,X_m are 1-negatively correlated if for all $\mathrm{S} \subseteq [1, m]$, the following holds:

$$\Pr[\forall i \in \mathrm{S} : \mathrm{X_i} = 1] \leq \Pi_{i \in S} \Pr[\mathrm{X_i} = 1] \tag{3}$$

Balls and bins view. The indices in a partition and the set of hints can be thought of as an instance of balls and bins [46]. Specifically, the indices in the partition correspond to bins. Each hint contributes one ball to the bin corresponding to the index selected by the hint from the partition. (Each partition corresponds to a separate instance of balls and bins and each hint contributes one ball to each balls-and-bins instance.)

The locations of balls keep changing as hints get replenished. Let BB_i denote the balls and bins view corresponding to the i^{th} partition. During i-th query, if a hint containing the query index q_i is found, that hint is used and replenished with another hint. The replenished hint selects q_i from the query partition and random indices from other partitions. In the balls-and-bins view, the hint stays in the same bin in BB_i but randomly relocates in all other instances of the balls and bins.

In the following two lemmas, consider a set of the random variables $X_1, X_2, X_3, \ldots, X_l$ as defined in eq. (1) corresponding to a set of l queries with query indices $q_1, q_2, q_3, \ldots, q_l$.

Lemma 9. *If the query indices $q_1, q_2, q_3, \ldots, q_l$ are from different partitions, then $X_1, X_2, X_3, \ldots, X_l$ are mutually independent.*

Proof. Let us consider j-th query in the balls and bins view. If bin q_j in BB_j is empty, PIR-PRIMARY is not used for this query and all the instances of balls and bins remain unchanged. On the other hand, if the bin q_j contains a ball, then the balls corresponding to the hint in all other instances of the balls and bins problem are relocated to a random bin. Therefore, the distribution of balls after query j conditioned on the outcome of query j is the same as the distribution of the balls right before query j.

Therefore, translating back to the hints view, the existence of a usable hint for query j does not influence whether any hints are available for further queries.

Using the above fact repeatedly, we have that

$$\Pr[X_1, X_2, \ldots, X_l] = \Pr[X_1] \cdot \Pr[X_2, X_3 \ldots, X_l]$$
$$= \Pr[X_1] \cdot \Pr[X_2] \Pr[X_3 \ldots, X_l] = \ldots = \Pi_{i=1}^{l} \Pr[X_i]$$

Hence, $X_1, X_2, X_3, \ldots, X_l$ are mutually independent. □

Lemma 10. *If the query indices $q_1, q_2, q_3, \ldots, q_l$ are from the same partition, then $X_1, X_2, X_3, \ldots, X_l$ are negatively correlated.*

Proof. Without loss of generality, let us assume that all the indices are from the first partition. Additionally, let the number of PIR-PRIMARY hints sampled be M

For all $i \in [1, l]$, $j \in [1, M]$, $k \in [0, \sqrt{N} - 1]$, let us define the set of random variables $B^i_{j,k}$ as follows

$$B^i_{j,k} = \begin{cases} 1, & \text{Hint j selects index k right before query i} \\ 0, & \text{Hint j does not select index k right before query i} \end{cases} \tag{4}$$

We also define the following set of random variables for all $j \in [1, M]$, $k \in [0, \sqrt{N} - 1]$.

$$Y^i_k = \Sigma_{j=1}^{j=m} B^i_{j,k} \tag{5}$$

Specifically, Y^i_k indicates the number of hints that select index k before query i.

We first want to prove that $Y^1_{q_1}, Y^2_{q_2}, \ldots, Y^l_{q_l}$ are negatively correlated. We call the set of random variables $B^1_{i,q_1}, B^2_{i,q_2}, \ldots, B^l_{i,q_l}$ as the contribution of the i-th hint to $Y^1_{q_1}, Y^2_{q_2}, \ldots, Y^l_{q_l}$ respectively. We first prove that the contribution of each hint is negatively correlated. Then, it follows that $Y^1_{q_1}, Y^2_{q_2}, \ldots, Y^l_{q_l}$ are negatively correlated as each of these random variables is a sum of contributions from all the hints, where the contribution of hints are independent of one another.

Let us look at the contribution of the first hint to each of $Y^1_{q_1}, Y^2_{q_2}, \ldots, Y^l_{q_l}$. We first assume that the hint remains unchanged through the l queries and then consider the general case where the hint could be replenished at multiple points

Assuming that the first hint remains unchanged during the l queries, we have that:

$$B^2_{1,q_2} = B^1_{1,q_2}, B^3_{1,q_1} = B^1_{1,q_3}, \ldots, B^l_{1,q_l} = B^1_{1,q_l} \tag{6}$$

This is because the index selected by the hint from the partition remains the same. The zero-one lemma from the

work of Dubhashi and Ranjan [46], states that a set of indicator random variables that have a sum of at most 1 are negatively correlated. Therefore, using the zero-one lemma, the random variables $B^1_{1,q_1}, \mathrm{B}^1_{1,\mathrm{q}_2}, \ldots, B^1_{1,q_l}$ are negatively correlated. Hence, using eq. (6), the contribution of the first hint to $Y^1_{q_1}, Y^2_{q_2}, \ldots, Y^l_{q_l}$, which are the random variables $B^1_{1,q_1}, \mathrm{B}^2_{1,\mathrm{q}_2}, \ldots, B^l_{1,q_l}$ respectively, are negatively correlated.

However, the first hint could be replenished at multiple points while other queries are executed in between any two queries we consider. For instance, if there is a replenishment after query i, the succeeding random variables after i, $B^{i+1}_{1,q_{i+1}}, \ldots, B^l_{1,q_l}$ are independent of the preceding variables $B^1_{1,q_1}, ..., B^{i-1}_{1,q_{i-1}}, B^i_{1,q_i}$. This is because the index selected by the hint after query i from the partition is independent of the index selected by the hint before query i. Extending this logic to multiple replenishments and using the discussion in the previous paragraph, the random variables in the contribution of the first hint between any two replenishments are negatively correlated and are independent of all the other random variables in the contribution. Therefore, the set of random variables $B^1_{1,q_1}, B^1_{1,q_2}, \ldots, B^1_{1,q_l}$ are negatively correlated.

The above argument for the first hint can be extended to the contribution of other hints to $Y^1_{q_1}, Y^2_{q_2}, \ldots, Y^l_{q_l}$. Since hints are sampled independently of each other, $Y^1_{q_1}, Y^2_{q_2}, \ldots, Y^l_{q_l}$, which are each a sum of negatively correlated contributions from the hints, are negatively correlated.

Notice that

$$X_i = \begin{cases} 1, & Y^i_{q_i} = 0 \\ 0, & Y^i_{q_i} \neq 0 \end{cases} \tag{7}$$

For a set of negatively correlated random variables $A_1, A_2, \ldots, A_n$ and any set of real values $a_1, a_2, \ldots, a_n$, the marginal probability bounds shown in the work of Dubhashi and Ranjan [46] states that:

$$\Pr[A_i \leq a_i, i \in [1,n]] \leq \Pi_{i\in[1,n]} \Pr[A_i \leq a_i] \tag{8}$$

Using the above result, $\forall S \subseteq [1,l]$, we have that

$$\Pr[\forall i \in \mathrm{S} : \mathrm{Y^i_{q_i}} \leq 0] \leq \Pi_{i\in S} \Pr[\mathrm{Y^i_{q_i}} \leq 0] \tag{9}$$

Therefore $\forall S \subseteq [1,l]$, we have that,

$$\Pr[\forall i \in \mathrm{S} : \mathrm{X_i} = 1] \leq \Pi_{i\in S} \Pr[\mathrm{X_i} = 1] \tag{10}$$

The above expression is precisely the condition for 1-negative correlation as defined here eq. (3). □

Lemma 11. *Consider the set of the random variables $X_1, X_2, X_3, \ldots, X_m$, as defined in eq.* (1)*, corresponding to a set of m queries with query indices $q_1, q_2, q_3, \ldots, q_m$. Then, $X_1, X_2, X_3, \ldots, X_m$ are negatively correlated.*

Proof. For all $S \subseteq [1,m]$. We need to prove that

$$\Pr[\forall i \in S : X_i = 1] \leq \Pi_{i\in S} \Pr[X_i = 1] \tag{11}$$

Let P_i be a set containing the indices in the i^{th} partition. Using lemma 9, we can write the LHS as

$$\Pr[\forall i \in S : X_i = 1] = \Pi_{i=1}^{\sqrt{N}} \Pr[\forall j \in S, q_j \in P_i : X_j = 1] \tag{12}$$

Using lemma 10, we have that for any i,

$$\Pi_{i=1}^{\sqrt{N}} \Pr[\forall j \in S, q_j \in P_i : X_j = 1] \leq \Pi_{\forall j\in S, q_j \in P_i} \Pr[X_j = 1] \tag{13}$$

Combining eq. (12) and eq. (13), we have that

$$\Pr[\forall i \in S : X_i = 1] \leq \Pi_{i\in S} \Pr[X_i = 1] \tag{14}$$

Hence, the indicator random variables $X_1, X_2, X_3, \ldots, X_m$ are 1-negatively correlated.

Appendix C. Overview of the Work of Wang and Ren [2]

Consider the high-level overview of WR PIR presented in section 3.2. The client tracks the contents of the hint table using the small-domain PRP keys and a data structure called the Relocation Data Structure. Using the Relocation Data Structure, the client uses an algorithm called Locate to determine the column containing a specific index and an algorithm called Access to determine the indices in a specific column of the hint table, and an algorithm called

We provide a high-level overview of the Relocation Data Structure and the algorithms Access and Locate below.

Relocation Data Structure. We first provide the intuition behind the construction of the data structure, followed by a description of the data structure.

Consider a specific row of the hint table whose permutation is called P (stored as a small-domain PRP key as defined in section 2.4). Each row in the hint table is equivalent to a graph G with nodes and edges defined as follows. The i-th node in the graph corresponds to the i-th cell in the row (or the i-th column in the row). Every node in G has an associated value. Specifically, the initial value associated with the i-th node is $P^{-1}(i)$. If the c-th column is used for the t-th query, the indices in the column are relocated to the column containing the t-th $\perp$. Edges are added to G to denote the relocation. Specifically, if the c-th column is used for the t-th query, an edge is added between node c and the node $P(\sqrt{N}+t)$ (column containing the t-th $\perp$). Notice that the state of the row in the hint table can be determined if the client tracks the edges of the graph G.

Thereby, the edges of graph G are stored in an array called Hist. Specifically, if the c-th column is used for the t-th query, the t-th entry in Hist is set to be c. Moreover, the inverse of this array is stored in a map called Hist^{-1}.

Additionally, the array Hist and the map Hist^{-1} are identical across all rows of the hint table and thereby need to be stored by the client only once.

Locate. Given an index i, the Locate algorithm determines the current column in which index i is located. In the graph

view of the row containing index i, initially, node $P(i)$ has the value i. As edges are added to a chain containing this node, the value i gets relocated to the end node in the chain. Therefore, the Locate algorithm starts the traversal at node $P(i)$ and traverses along out-edges until a node without any out-edge is reached. The out-edges at each node can be determined by using the Hist^{-1} data structure and permutation P. Finally, the column number corresponding to the final node is the column containing index i.

Access. Given a column c, the Access algorithm determines the value in column c of the row. In the graph view, this is the initial value at the start node of the chain whose end node is c. The Access algorithm starts at node c and traverses the graph G along in-edges. Upon reaching a node n with no in-edge, the algorithm outputs $P^{-1}(n)$, i.e., the initial value at node n. The in-edges can be determined by using Hist data structure.

Lemma 12. *The query number at which an index i is relocated to column c of the hint table can be determined by the client.*

Proof. Consider the graph view G of the row containing index i. At the end of the offline phase, the index i is associated with node $P^{-1}(i)$, and node c contains index i in the current state of the hint table. The node $P^{-1}(i)$ has no in-edge (Fact 3.7, [2]) and the set of edges connecting node c to node $P^{-1}(i)$ forms a chain S (Corollary 3.10, [2]). Index i relocates to the end node of the chain as and when edges are added. Thereby, the query number at which the final edge is added to S determines when i is relocated to column c. □

Appendix D. Completeness of Verifiable PIR-BACKUP

Lemma 13. *In Verifiable PIR-BACKUP, the client can determine the order in which indices are added to the hint column.*

Proof. From lemma 12, the client can determine the query number at which each index is added to the hint column. Therefore, the client can determine the correct order in which indices are added to the hint column. □