A Proposal for Assisted Private Information Retrieval

Natnatee Dokmai¹, L. Jean Camp¹, and Ryan Henry²

¹Indiana University, Bloomington
²University of Calgary

Abstract

Private Information Retrieval (PIR) addresses the cryptographic problem of hiding sensitive database queries from database operators. In practice, PIR schemes suffer from either high computational costs or from restrictive requirements difficult to justify in practical settings. In this work, we introduce Assisted Private Information Retrieval (APIR), a new PIR problem for keyword-value databases which generalizes and relaxes the database consistency assumption in multi-server PIR. Leveraging the decentralized nature of Domain Name Service (DNS), APIR is able to address a privacy issue inherent to encrypted DNS proposals such as DNS-over-HTTPS (DoH) by preventing DNS operators from collecting sensitive data. We propose a construction of Synchronized APIR, an efficient hybrid APIR scheme between black-box single-server PIR and non-black-box multi-server PIR. We apply Synchronized APIR to a proof-of-concept protocol for private DNS query, and demonstrate that APIR is able to outperform the baseline single-server PIR protocol after the initial one-time cost.

1 Introduction

One often overlooked privacy aspect of information access over a network is the monitoring of information queries by database-operating servers. To illuminate this concern, let us consider a scenario: a user wishes to download a sensitive file from a website, but the act of access to that particular file is incriminating. At this point, the user has a few options. She can use an encrypted channel like TLS to prevent eavesdropping, but this does not prevent the leakage of the query to the web server; or she can use anonymizing technology like VPN or Tor to hide her identity, but this does not prevent the web server from being notified and collecting statistics about file access.

A case in point is the Domain Name System (DNS). DNS privacy and security are the core argument for encrypted DNS such as DNS-over-HTTPS (DoH) and DNS-over-TLS (DoT). However, the present encrypted DNS proposals do not necessarily increase consumers’ privacy. The most notable privacy risk is increased data concentration at DNS operators. Given the exploitation of users’ data by data collectors and online services in the recent years, it is not far-fetched to claim that concentration of data is a privacy violation.

The family of cryptographic protocols that proposes to address this issue is private information retrieval (PIR). The core privacy definition of PIR requires that any two different PIR queries appear indistinguishable from the point of view of the attacker, thus preventing database operators from collecting sensitive information about the queries. There have been many proposed PIR schemes in recent years, yet they all suffer from similar practical issues. The PIR schemes that require one server to operate, or single-server PIR, are applicable to practical scenarios; yet, they often rely on additively homomorphic encryption schemes which are computationally expensive and incur high communication costs in practice. The PIR schemes that require multiple servers to operate, or multi-server PIR, are orders of magnitude cheaper than single-server PIR to deploy, yet they have conflicting requirements that 1) all the servers hold some form of the identical copy of the database, and 2) some servers do not collude. In practice, it is difficult to imagine a scenario in which both requirements would hold simultaneously. In the DNS application for instance, it can be argued that DNS servers administered by independent organizations are unlikely to collude. However, to require that they hold identical copies cache tables and zone files seem impossible. Further, it makes the application less useful e.g. it does not allow the same domain to point to different local IP addresses, as in the case of scalable cloud services.

1.1 Our contributions

In this work, we introduce assisted private information retrieval (APIR). APIR generalizes multi-server PIR in the following ways, while still maintaining the non-collusion assumption:

1. The databases are keyword-value maps instead of indexed vectors.
2. Client can query the main server with assistance from assisting servers without requiring the assisting servers to hold and identical copy of the database.

We present Synchronized APIR, an APIR scheme which allows the client to synchronize the “view” of the databases across the servers prior to making queries. Synchronized APIR is a hybrid protocol between a black-box single-server PIR scheme and a non-black-box multi-server PIR scheme similar to CGKS [7]. The hybridization takes advantage of the cheaper multi-server PIR scheme to “assist” in lowering the costs of the single-server PIR scheme between the client and main server; the level of assistance is determined by the amount of overlap between main server’s and assisting servers’ databases. Because Synchronized APIR makes use of a black-box single-server scheme, improvements in single-server PIR in the future will also lead to improvements in Synchronized APIR. We provide a formal analysis and an implementation of Synchronized APIR in Rust using SealPIR [9] as the underlying single-server scheme.

We apply Synchronized APIR to demonstrate a proof-of-concept private DNS query application, specifically to query nameserver (NS) records, among DNS cache servers. Then, we evaluate the application with simulated datasets based on realistic assumptions about DNS queries and cache behavior.

The results show that despite the higher initial one-time cost, private DNS query via Synchronized APIR is able to outperform SealPIR, the baseline single-server PIR, either in communication cost by a factor of 5 or in computational cost by a factor of 8, given the most optimal popularity distribution of DNS queries.

2 Related Works

Our work is partly inspired by Fanti et al. [9], who propose an information-theoretic multi-server PIR scheme over unsynchronized databases. In this setting, the PIR servers hold different copies of the same indexed database with some database values missing. The scheme proceeds by first allowing the client to synchronize the database “view” to identify the missing values. Then, the client makes a query in a way that avoids the missing values, while also hide which values are missing from the servers. This setting is applicable to peer-to-peer (P2P) file-sharing, where P2P users share and download parts of the same file or database in small, indexed fragments.

We view Fanti et al.’s scheme as a solution to a subset of the APIR problem with two additional limitations: 1) the database is strictly indexed, and 2) the client is not allowed to query any values with missingness. (Indeed, Synchronized APIR also directly solves this problem.) However, we remark that the second limitation may be counter-productive to privacy in practice. Sensitive data is often the rarer one and the one that goes missing; this introduces a contradiction to Fanti et al.’s scheme where the more sensitive the data is, the more likely it is unavailable. Clients therefore must resort to a less privacy-preserving service to be able to access the data. APIR insists that all database values are retrievable for this reason, and Synchronized APIR solves this problem by making a separate single-server PIR query to the values with missingness.

3 Preliminaries

3.1 Single-Server PIR

Single-server PIR, or sPIR for short, is a PIR protocol that requires only one database-operating server to interact with the client to query information. sPIR is associated with computational PIR (CPIR) because of the computational assumptions usually required to instantiate this scheme. One example of an sSPIR scheme is the trivial PIR, where the client downloads the entire database from the server and picks out the value she wants to query locally. This technically satisfies the privacy requirements of PIR because the server is unable to learn which value the client wishes to query. However, this results in a high communication, which is prohibitive in practice. Therefore, any non-trivial PIR schemes must aim to satisfy the privacy requirements as well as keep the communication cost lower than the trivial PIR.

We formally define sPIR below in Definition 3.1. A typical flow of an sSPIR protocol proceeds as follows. 1) The client generates the public-secret key pair with SGEN and gives the public key to the server. 2) Once the client has decided to query the i-th item in the database, an sSPIR query is generated with SQUERY using i. The query is sent to the server. 3) The server generates a reply from the query and database using SREPLY. The reply is returned to the client. 4) The client decodes the reply with the secret key using SDECODE to obtain the answer.

Definition 3.1 (sSPIR Scheme). Given a Server operating an indexed database \( V = (v_0, \ldots, v_{m-1}) \) for all \( i \in \{0, \ldots, m-1\} \). An sSPIR scheme is a tuple \( \Pi^{SPIR} = (\text{SGEN}, \text{SQUERY}, \text{SREPLY}, \text{SDECODE}) \) defined by

- \( \text{SGEN}(1^\lambda) \rightarrow (\text{spk}, \text{ssk}) \) where \( \lambda \) is the security parameter, \( \text{spk} \) the public key, and \( \text{ssk} \) the secret key.
- \( \text{SQUERY}(\text{spk}, i, m) \rightarrow \text{sq} \) generates query \( \text{sq} \) from index \( i \in \{0, \ldots, m-1\} \) targeting item \( v_i \).
- \( \text{SREPLY}(\text{spk}, V, \text{sq}) \rightarrow \text{sr} \) generates reply \( \text{sr} \) for query \( \text{sq} \) from database \( V \).
The formal definition of sPIR correctness is provided in Definition A.1. Intuitively, an sPIR scheme is correct if the answer is the same as the target value in the query.

The formal definition of sPIR privacy is provided in Definition A.2. Intuitively, an sPIR scheme is privacy-preserving if the adversary is unable to distinguish between a query encoding index $i$ and a query encoding index $j$, even if the adversary can choose $i$ and $j$ and has oracle access to SQUERY.

Indeed, there are many existing schemes in literature that can satisfy these definitions of sPIR, notably because an additively homomorphic encryption scheme implies an sPIR scheme [14]. An additively homomorphic encryption scheme can be used as a building block as follows: 1) For a given index $i$, the query is encoded as the $i$-th standard basis vector $e_i$; 2) the client encrypts $e_i$, component-wise using the homomorphic encryption scheme; 3) the server homomorphically evaluates the inner product, treating the database values as scalar constants so that the result is a linear combination; 4) the client decrypts the result. Note that this is merely a simplified construction. There have many techniques introduced to lower computational and communication cost in practice, including recursive PIR [3], which treats the database as two-dimensional and accesses it by row and column index; and ciphertext packing [4], which packs multiple query indices into a single ciphertext.

Although this instantiation of sPIR seems simple enough, in practice the computational cost can be too high—sometimes to the point where it can be faster for the client to download the entire database [13]. In recent years, the development has been moving toward lattice-based cryptography [2,4,11], whose efficiency has enabled the PIR schemes to become more practical.

In this work, we use SealPIR [4], one of the state-of-the-art sPIR schemes, as a building block for our scheme. In addition to the relatively low computational cost utilizing the aforementioned techniques, SealPIR offers a significant advantage over its predecessors via its query compression technique, which cuts down the size of a query by a large factor.

### 3.2 Multi-Server PIR

Multi-server PIR, or mPIR for short, is a PIR scheme that requires multiple database-operating servers to interact with the client to query information. mPIR is a broader category of information-theoretic PIR (IT-PIR) in literature, in that all IT-PIR schemes are multi-server, but some mPIR schemes can be a hybrid between IT-PIR and CPIR. Existing mPIR schemes rely on the following two assumptions: 1) databases must be consistent, meaning that each server holds the same “ground-truth” copy of the database which may be preprocessed for the scheme; and 2) no more than a given number of servers can collude, meaning that they cannot share the client’s private information other than what is instructed. One of the well-known mPIR schemes is the CGKS scheme [7]. Below, we provide a 3-server example of CGKS, although CGKS can naturally extended to an unbounded number of servers.

Suppose there are three database-operating servers, $S_1$, $S_2$, $S_3$, each holding a consistent database $V = (a, b, c, d, f) \in \text{GF}(2)^5$; and suppose the client $C$ wishes to retrieve the item at index 2 i.e. $c$, then she must generate 3 queries, one for each server, following the steps below.

1. Sample $q_1 \in \text{GF}(2)^5$ and $q_2 \in \text{GF}(2)^5$ uniformly at random. Assume from now on that $q_1 = (0, 0, 1, 1, 0)$ and $q_2 = (0, 1, 0, 0, 0)$.
2. Compute $q_3 \leftarrow q_1 \oplus q_2$. Therefore $q_3 = (0, 1, 1, 1, 0)$.
3. Encode index 2 as $e_3 = (0, 0, 1, 0, 0)$ (or index 3 as $e_{i+1}$ in general).
4. XOR $q_3$ with $e_3 = (0, 0, 1, 0, 0)$, namely $q_3 \leftarrow q_3 \oplus e_3$. Therefore $q_3 = (0, 1, 0, 1, 0)$.

Next, $C$ sends each query $q_i$ to server $S_i$, who then generates a reply $r_i$ by computing a dot product $r_i \leftarrow q_i \cdot V$. As a result, we have $r_1 = c \oplus d$, $r_2 = b$, and $r_3 = b \oplus d$. $S_i$ returns reply $r_i$ to $C$. Finally, $C$ decodes the replies to obtain the answer by computing $r_1 \oplus r_2 \oplus r_3 = c$.

The CGKS scheme generalizes to any number of servers with databases of any size. However, it is worth remarking that CGKS scheme as stated above is only non-trivial (i.e. communication cost lower than downloading the entire database) if the database is not exponentially lopsided (i.e., if the length of each record is super-logarithmic in the number of records). Otherwise, the query size is asymptotically equivalent to database size.

Correctness and privacy is defined in the same manner as sPIR. The privacy definition includes the notion of collusion threshold in addition: that a group of $t$ or fewer colluding servers cannot distinguish between two queries. This property is called $t$-collusion resistance. CGKS scheme is $(n-1)$-collusion-resistant where $n$ is the number of servers. This is because any $n-1$ queries are statistically indistinguishable for a uniformly random string of the same length.

In our construction that follows, we will use as a building block a variation of CGKS that does not require consistent databases and uses a PRG to generate queries in order to reduce the communication cost.

* $\text{SDecode}(ssk, sr) \rightarrow sa$ decodes reply sr for answer sa.
4 Assisted PIR

The goal of this section is to provide a generic definition of Assisted PIR (APIR) including that of privacy, correctness, and communication efficiency. We will show how to construct our APIR scheme called “Synchronized APIR” in the upcoming section.

APIR comprises three groups of participating parties with distinct roles:

<table>
<thead>
<tr>
<th>Notations</th>
</tr>
</thead>
<tbody>
<tr>
<td>* C, client</td>
</tr>
<tr>
<td>* S₀, main server</td>
</tr>
<tr>
<td>* S₁ − Sₙ, n assisting servers</td>
</tr>
</tbody>
</table>

C wishes to retrieve the value associated with key \( k \) from keyword-value database \( DB₀ \), operated by \( S₀ \), with assistance from \( S₁, \ldots, Sₙ \) who independently operate keyword-value databases \( DB₁, \ldots, DBₙ \), respectively. \( DB₁, \ldots, DBₙ \) do not need to be the exact duplicates of \( DB₀ \), although they may have some common keyword-value pairs. Note that this is in contrast to sPIR and mPIR in Section 3 where databases are assumed to be index-based and consistent. APIR is formally defined below.

<table>
<thead>
<tr>
<th>Notations</th>
</tr>
</thead>
<tbody>
<tr>
<td>* ID := ( {0, \ldots, n} ), the set of server ID’s</td>
</tr>
<tr>
<td>* AID := ( {1, \ldots, n} ), the set of assisting server ID’s</td>
</tr>
<tr>
<td>* ( (a_i)<em>I := (a</em>{i_1}, \ldots, a_{i_n}) ); ( i_j \in I ), a sequence ordered by the index set ( I ). We sometimes use ( (a_i)_{i \in I} ) instead for clarity.</td>
</tr>
<tr>
<td>* Keys(db) the set of all keywords in database db</td>
</tr>
</tbody>
</table>

**Definition 4.1 (APIR Scheme).** Given a set of \( n + 1 \) database-operating servers and define \( ID := \{0, \ldots, n\} \) the set of server ID’s. An APIR scheme is a tuple \( \Pi^{APIR} = (ServGen, CliGen, Query, Reply, Decode) \) defined by

- \( ServGen(id, DB_id) \rightarrow par_{id} \), where \( id \in ID \) and \( par_{id} \) is the database parameter for \( DB_id \).
- \( CliGen(1^\lambda, t, (par_{id})_ID) \rightarrow (pk, sk) \), where \( \lambda \) is the security parameter, \( t \) the collusion threshold where \( 1 \leq t \leq n \), \( pk \) is the public key, and \( sk \) the secret key.
- \( Query(pk, sk, k) \rightarrow (q_{id})_ID \) generates query \( q_{id} \) for Server id from query keyword \( k \in Keys(DB_0) \) targeting database value \( DB_0(k) \).
- \( Reply(id, pk, DB_id, q_{id}) \rightarrow r_{id} \) generates reply \( r_{id} \) for query \( q_{id} \) from database \( DB_id \), where \( id \in ID \).
- \( Decode(sk, (r_{id})_ID) \rightarrow a \) decodes replies \( (r_{id})_ID \) for answer \( a \).

The flow of the scheme is similar to what we have previously seen in sPIR and mPIR in Section 3. 1) The servers generate database parameters using \( ServGen \) and send them to client. 2) The client generates a public-secret key pair from the database and security parameters using \( CliGen \). 3) Client chooses a query keyword and generates queries using \( Query \); each server gets its own query. 4) The servers responds to the query with their own database using \( Reply \). And finally, 5) the client aggregates all the replies and decodes to obtain the answer with \( Decode \) using the secret key.

The correctness of APIR is defined according to what is contained in \( DB_0 \). If the client query with keyword \( k \), then \( DB_0(k) \) is defined to be the correct answer even though there may be other \( DB_id \) such that \( DB_id(k) \neq DB_0(k) \); formally

**Definition 4.2 (APIR Correctness).** Following Definition 4.1, scheme \( \Pi^{APIR} \) is APIR-correct for any set \( ID \), databases \( DB_id \) for all \( id \in ID \), and \( k \in Keys(DB_0) \) if

- \( \forall id \in ID : par_{id} \leftarrow ServGen(id, DB_id) \)
- \( (pk, sk) \leftarrow CliGen(1^\lambda, t, (par_{id})_ID) \)
- \( (q_{id})_ID \leftarrow Query(pk, sk, k) \)
- \( \forall id \in ID : r_{id} \leftarrow Reply(id, pk, DB_id, q_{id}) \)
- \( a \leftarrow Decode(sk, (r_{id})_ID) \)

then \( a = DB_0(k) \).

The privacy definition of APIR is similar to that of mPIR in that we want to capture both the indistinguishability of queries when collusion does not exceed a certain threshold. Intuitively, an APIR scheme is privacy preserving if the attacker, who compromises a set of servers, cannot distinguish between two queries generated from query keywords \( k_0, k_1 \) of the attacker’s own choice. Moreover, we want to capture a real-world scenario in which the client is not informed of which servers are compromised, or if they are compromised at all (and thus the role of the oracle \( O \) in the definition below to relay the compromised queries to the attacker without the client’s knowledge). We formally define this using a game-based definition below.

**Definition 4.3 ((\(\lambda, t\))-APIR Privacy).** Define a APIR privacy experiment \( \text{Priv}_{A, \Pi^{APIR}, (1^\lambda)} \) for an APIR scheme \( \Pi^{APIR} \) according to Definition 4.1 and adversary \( A \) below.

1. \( A \) chooses well-formed database parameters \( (par_{id})_ID \) and outputs \( (par_{id})_ID \). \( A \) chooses a collusion set \( C \subset ID \) such that \( |C| \leq t \) and sends \( C \) to the oracle \( O \).
2. The parameters are generated by \((pk, sk) \leftarrow \text{Cligen}(\lambda, t, (par_{id}))\) and \(pk\) is given to \(A\).

3. \(A\) is given oracle access to \textsc{query} in the following way: \(A\) chooses and outputs \(k \in \mathcal{K}(par)\) where \(\mathcal{K}(par)\) is the query keyword space determined by a well-formed \(par\). Queries are generated by \((q_{id})_0 \leftarrow \text{query}(pk, sk, k)\) and \((q_{id})_{id}\) is given to \(O\). \(O\) gives \((q_{id})_C\) to \(A\).

4. \(A\) chooses \(k_0^*, k_1^* \in \mathcal{K}(par)\) and outputs \((k_0^*, k_1^*)\). A uniformly random bit is sampled \(b \leftarrow \{0, 1\}\). Queries are generated by \((q_{id})_b \leftarrow \text{query}(pk, sk, k_b^*)\) and \((q_{id})_{id}\) is given to \(O\). \(O\) gives \((q_{id})_C\) to \(A\).

5. \(A\) is given more oracle access to \textsc{query}.

6. \(A\) outputs \(b^* \in \{0, 1\}\). The output of the experiment is defined to be 1 if \(b^* = b\) and 0 otherwise.

\(\Pi^{\text{pir}}\) is \((\lambda, t)\)-APIR privacy-preserving for all PPT adversary \(A\) if there exists a negligible function \(\text{negl}\) such that

\[
\Pr[\text{PrivA}_{A, \Pi^{\text{pir}}, t}(1^\lambda) = 1] \leq \frac{1}{2} + \text{negl}(\lambda)
\]

5 Our Scheme: Synchronized APIR

In this section, we present our construction of APIR called \textit{Synchronized APIR}. Synchronized APIR allows the client to synchronize the global “view” of the databases before querying for data, resulting in a high one-time communication cost to initialize the scheme but low amortized computational and communication cost to query. Further, Synchronized APIR makes use of a black-box mPIR scheme described in Section 3.1 in a hybrid PIR, meaning that any improvements to mPIR schemes result in improvements to Synchronized APIR. For the rest of this section, we will introduce the concept of Synchronized APIR in Section 5.1, then follow up with the full description in Section 5.2. Finally, we describe how we implement Synchronized APIR in Section 5.3.

5.1 Concept

Let us consider an example of keyword-value databases in Figure 1, step 1, and suppose that each database is independently operated by a PIR server in a network. A client with a given query keyword \(k\) wishes to retrieve the value stored in database \(DB_0\) associated with keyword \(k\), while hiding \(k\) from a malicious party controlling the servers. How could this be achieved? In the traditional mPIR setting, databases are required to be consistent for correctness. However, because this is not the case in Figure 1, step 1, mPIR is not clearly achievable. Instead, the client must resort to the costly sPIR on \(DB_0\) and disregard \(DB_1, DB_2\) and \(DB_3\).

To circumvent this issue, Synchronized APIR makes use of some of the keyword-value pairs in \(DB_1, DB_2, DB_3\) that are consistent with \(DB_0\) to “assist” in reducing the cost of sPIR on \(DB_0\) via a hybrid PIR. To demonstrate this concept, we will walk through the example in Figure 1 below.

**Step 1**: For the purpose of this demonstration, we assume that the client can fully observe \(DB_0, \ldots, DB_3\) (how exactly this is done will be explained in the upcoming section). The client first notices that there are keyword-value pairs in \(DB_1, DB_2\), \(DB_3\) inconsistent with \(DB_0\), indicated in light grey. The client prefers the “correct” versions of the keyword-value pairs according to \(DB_0\), so the inconsistent pairs in \(DB_1, DB_2, DB_3\) are disregarded.

**Step 2**: To lower the cost of Synchronized APIR, the participating parties want to apply traditional mPIR to “assist” wherever possible. This requires that the client first determines how many servers can collude without leaking \(k\), i.e. the \(t\) collusion threshold similarly to the one in CGKS in Section 3.2. Let suppose that the client decides that \(t = 2\). Recall from Section 3.2 that a given threshold \(t\) requires at least \(t + 1\) duplicates of the same database values across the databases; so the client must identify the keyword-value pairs with at least 3 duplicates to pass the threshold requirement.

\((a, 100), (b, 200), (c, 300), \) and \((d, 400)\) have at least 3 duplicates \((a, 100)\) has 4, so the one in \(DB_3\) is redundant and disregarded). Neither of \((e, 500), (f, 600),\) or \((g, 700)\) meets the requirement, so they are disregarded. The pairs that do not pass the threshold are indicated in dark grey.

**Step 3**: The client splits the pairs that pass the threshold from those that do not. This reveals, on the top rows in the green box, the pairs which can be retrieved with mPIR, and, on the remaining bottom rows in the purple box, the pairs which can only be retrieved with sPIR.

**Step 4**: The client and servers engage in a hybrid PIR protocol. The client makes separate PIR queries for mPIR and sPIR from key \(k\). All the servers process the mPIR queries, while only the server operating \(DB_0\) processes the sPIR query.

There are some important details we have omitted here for conceptual simplicity. In the following section, we will expand on the concept of Synchronized APIR to answer these questions:

- How can the client “synchronize” the databases according to step 1-3 without needing to download them in full, and at a low communication cost?
• How can the client construct a coherent mPIR query in Synchronized APIR when the databases are consistent with DB, not passing t = 2?

• How can the communication cost of query and reply be optimized?

### 5.2 Protocol Description

In order to query keyword k, Client C works with main server S₀ and assisting servers S₁, ..., Sₙ through three phases, as illustrated in Figure 2: **Synchronization, Setup**, and **Query**. Synchronization and Setup must be completed once at the start, while a new Query session can be repeated for every new query keyword C wishes to query.

In the following sections, we will describe step 1 - 3 of Synchronized APIR as shown in Figure 2 in details.

#### 5.2.1 Synchronization Phase

During Synchronization phase, C obtains catalogs from all servers, and then informs them of the synchronized mPIR keywords that pass the threshold. The goal of this phase is to allow C to map out all the keyword-value pairs across the databases without having to download them all in full. We refer to step 1 - 3 in Figure 2 and (1) - (3) in 3 as a demonstration to describe Synchronization phase below.

<table>
<thead>
<tr>
<th>Notations</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Map(·), a keyword-value map : suppose that M = Map(S), then (k, v) ∈ S iff M[k] = v.</td>
</tr>
<tr>
<td>• ∅, an empty set</td>
</tr>
</tbody>
</table>

**Input**

- DBid
- t

**Output**

- Catid,Mid
- Cat₀, (Cat₀)ₐid, (Mid)ₐid, S

**Synchronization Protocol**

1. C obtains the full catalog Cat₀ from S₀ and catalog intersections Cat₁,...,Catₙ from S₁,...,Sₙ by following the steps below:
   A. For each id ∈ IDₐid, S₀ generates catalog
      Catₐid ← Map {{(k, h) | (k, v) ∈ DBidₐid, h = H(v)}
      where H(·) is a universal hash function with short hash values, chosen at random by C from the universal family. (A shorter hash length will reduce the communication cost but increases the probability of collision.) This results in the catalogs that are smaller than the databases in size but capable of representing the uniqueness of database values. In Figure 3, we choose H(v) = v/10 to demonstrate the point that an appropriate hash function should be able to produce short unique hash values. In general when the patterns of database values are not predetermined, universal hash functions should be applied to reduce hash collisions.
   B. C downloads Cat₀ from S₀. For each id ∈ AID, C engages in catalog intersection protocol CatINTERSECTION with S₀, where C obtains catalog intersections Catₐid = Catₐid (Cat₀) at the end. The goal of CatINTERSECTION is to transmit catalog intersections at a cost lower than sending full catalogs. CatINTERSECTION is described in Section 5.2.5. Figure 3 demonstrates how keyword-value pairs inconsistent with DB₀ in DB₁,...,DBₙ and with Cat₀ in Cat₁,...,Catₙ are eliminated via catalog intersection. With the full catalog and catalog intersections, C now has complete information of the keyword-value pairs consistent with DB₀.

2. C tags the catalogs to categorize keyword-value pairs for either mPIR or sPIR:
   ((Mid)ₐid, S) ← TAG (Cat₀, (Cat₀)ₐid, t)
   where t is the collusion threshold i.e. the number of colluding servers up to which C can tolerate without
leaking query keywords; \( M_d \) is the mPIR keyword set i.e. set of keywords determined to be queried via mPIR from \( S_d \); \( S \) is the sPIR keyword set i.e. set of keywords determined to be queried via sPIR from \( S_0 \). \( \text{TAG} \) is detailed in Algorithm 1 below. In Figure 3 \( C \) has decided that \( t = 2 \), so the pairs that pass the threshold for mPIR must have \( t + 1 = 3 \) duplicates: 1 duplicate in \( \text{Cat}_0 \) and 2 duplicates in \( \text{Cat}_1', \text{Cat}_2', \text{Cat}_3' \). \( (a, 10), (b, 20), (c, 30) \), and \( (d, 40) \) all pass the threshold \((a, 10) \) in \( \text{Cat}_3' \) is redundant and disregarded). Neither of \( (e, 50), (f, 60), \) or \( (g, 70) \) pass the threshold, so \( C \) disregards them. This results in \( M_0, \ldots, M_4 \) for the set of keywords that pass the threshold for mPIR, and the rest in \( S \) for sPIR.

Algorithm 1 Catalog Tagging. In line 3–8 if there are at least \( t \) duplicates of a given keyword, then \( t \) of them are tagged for mPIR and stored in \( M_d \) \((t + 1 \) in total, including those in \( M_0 \)). In line 10 the rest of the keywords are tagged for sPIR. In line 6 \( C_k \) can be chosen at random or with a specific optimization strategy.

\[
\begin{align*}
1: & \quad \text{procedure TAG}(\text{Cat}_0, (\text{Cat}_d)_\text{AID} : t) \\
2: & \quad \forall d \in \text{ID} : M_d \leftarrow \phi \\
3: & \quad \text{for } k \in \text{Keys} (\text{Cat}_0) \text{ do} \\
4: & \quad \quad G \leftarrow \{d \in \text{ID} | k \in \text{Keys} (\text{Cat}_d')\} \\
5: & \quad \quad \text{if } |G| \geq t \text{ then} \\
6: & \quad \quad \quad \text{choose } C_k \subseteq G \text{ such that } |C_k| = t \\
7: & \quad \quad \quad \forall d \in C_k \cup \{0\} : M_d \leftarrow M_d \cup \{k\} \\
8: & \quad \quad \text{end if} \\
9: & \quad \text{end for} \\
10: & \quad S \leftarrow \text{Keys} (\text{Cat}_0) \setminus M_0 \\
11: & \quad \text{return } ((M_d)_\text{ID}, S) \\
12: & \end{procedure}
\]

For each \( d \in \text{ID} \), \( C \) and \( S_d \) engage in protocol KEYSYNCHRONIZE, where \( S_d \) obtains \( M_d \) at the end. The goal of KEYSYNCHRONIZE is to transmit \( M_d \) at a cost lower than sending them in full. KEYSYNCHRONIZE is described in Section 5.2.2.

5.2.2 Setup Phase

Let \( l = \text{poly}(\lambda) \) denote the total number of queries that \( C \) will be making during Query phase \((l \) is not necessarily predetermined at this point). We call each query during Query phase a Query session, each denoted with session ID \( \text{sid} \in \{1, \ldots l\} \).

Notations

- \( \text{sid} \in \{1, \ldots l\} \), session ID. Starting at \( \text{sid} = 1 \), \( \text{sid} \) increases by \( 1 \) for each new iteration of Query. Let \( l \) denote the number of sessions and \( [l] := \{1, \ldots l\} \) the set of all session ID’s.
- \( \text{index}(a, A) := |\{b \in A | b < a\}| \), index of \( a \) in set \( A \).

During Setup, \( C \) sends each of \( S_0, \ldots, S_n \) a random PRG seed which will be used to generate random mPIR queries for all Query sessions \( \text{sid} \in [l] \). We follow steps 3 and 5 in Figure 2 and 5 in 3 to describe Setup phase below.

Input

\[
\begin{align*}
& \quad |M_d| \\
& \quad l, (M_d)_{\text{AID}} \\
& \quad \text{from } S_d, \forall \text{id} \in \text{AID} \\
& \quad \text{from } C \\
\end{align*}
\]

Output

\[
\begin{align*}
& \quad \text{spk} \\
& \quad \text{to } S_0 \\
& \quad (m_0^{\text{sid}})_{\text{sid} \in [l]} \\
& \quad \text{to } S_d, \forall \text{id} \in \text{AID}
\end{align*}
\]
Figure 3: Synchronization. An example of databases in a Synchronized APIR setting with 3 assisting servers that undergo Synchronization phase.

\[ \text{Setup Protocol} \]

\[ \text{Algorithm 2 mPIR Token Generation.} \]

\[ \text{5.2.3 Special Case: mPIR-Only Query Phase} \]

Before proceeding to describing the full protocol of Query phase, it would be instructive to walk through the special case in which all keyword-value pairs can be retrieved with mPIR i.e. \( S = \emptyset \) in Algorithm 1. The goal is to show how all the mPIR components fit together during Query phase to provide correct answers, without having to deal with the complexity of the hybrid PIR. We shall follow the steps in Figure 1 for this walk-through.

Suppose that at this point in time \( C \) and \( S_0, \ldots, S_n \) have already completed Synchronization in Section 5.2.1 and Setup in Section 5.2.2. We will now pick up step (6) and (7)* → (9)* (* to denote the steps for this special case) in Figure 4 from here.

\[ \text{Input} \]

\[ \text{Output} \]
Figure 4: mPIR-Only Query Phase Workflow. Workflow of mPIR components in during Query phase to demonstrate how mPIR keyword-value pairs can be queried with the client’s query keyword.

\[ M_0, tok^{sid}, k^{sid} \]

\[ \text{Output} \]
\[ \overset{\to}{a^{sid}} \]
\[ \text{to} \]
\[ C \]

mPIR-Only Query Protocol for Session \( sid \)

(6) For each assisting servers \( id \in \text{AID} \):

A. \( S_{id} \) generates a reply from the query

\[ m_{id}^{sid} \leftarrow \text{MREPLY} \left( M_{id}, DB_{id}, m_{id}^{sid} \right) \]

and sends \( m_{id}^{sid} \) to \( C \). MREPLY is described in Algorithm 3 below. Figure 4 step (6) demonstrates the mPIR reply generation process: the values of \( DB_{id} \) are filtered with \( M_{id} \) by key; and the dot product in \( GF(2) \) between the filtered values and random query bits produces the reply. This process is identical for \( S_2 \) and \( S_3 \).

Algorithm 3 mPIR Reply Generation. The dot product in line 3 is defined in \( GF(2) \).

1. \textbf{procedure} MREPLY(\( M_{id}, DB_{id}, m_{id}^{sid} \))
2. \[ V \leftarrow (DB_{id}[k])_{k \in M_{id}} \]
3. \[ m_{id}^{sid} \leftarrow m_{id}^{sid} \cdot V \]
4. \textbf{return} \( m_{id}^{sid} \)
5. \textbf{end procedure}

Note that this step can be completed offline as it does not require a query keyword \( k^{sid} \) by \( C \). This means that during online time (that is, step (7) onwards), \( C \) only needs to interact with \( S_0 \), which significantly reduces the latency.

(7) * Once \( C \) has decided on a mPIR query keyword \( k^{sid} \in M_0 \) to query, she generates a mPIR query for \( S_0 \)

\[ m_{0}^{sid} \leftarrow \text{MQUERY} \left( M_0, tok^{sid}, k^{sid} \right) \]

where MQUERY is described in Algorithm 4 below. Intuitively, as visually demonstrated in Figure 4 MQUERY flips one bit of the token \( tok^{sid} \) at the index position of \( k^{sid} \) in \( M_0 \) to produce the query \( m_{0}^{sid} \).

Finally, \( C \) sends \( m_{0}^{sid} \) to \( S_0 \) to query.

Algorithm 4 mPIR Query Generation.

1. \textbf{procedure} MQUERY(\( M_0, tok, k \))
2. \[ m_{0} \leftarrow tok \]
3. \textbf{if} \( k \in M_0 \) \textbf{then}
4. \[ i \leftarrow \text{index}(k, M_0) \]
5. \[ m_{0}[i] \leftarrow m_{0}[i] \oplus 1 \]
6. \textbf{end if}
7. \textbf{return} \( m_{0} \)
8. \textbf{end procedure}

(8) * Upon receiving \( m_{0}^{sid} \), \( S_0 \) processes the query in the same way queries are processed in step (6), except that here \( m_{0}^{sid} \) is given by \( C \),

\[ m_{0}^{sid} \leftarrow \text{MREPLY} \left( m_{0}^{sid}, M_0, DB_0 \right) \]
\( S_0 \) returns \( m_{0id} \) to \( C \).

\( \circ \) Now that \( C \) has received all \( m_{0id}, \ldots, m_{nids} \), she can decode them for the answer by a simple XOR

\[
a^\text{sid} \leftarrow \bigoplus_{id \in ID} m_{id}^\text{sid}
\]

where \( a^\text{sid} = DB[k^\text{sid}] \) as a result.

To see why it is the case that \( a^\text{sid} = DB[k^\text{sid}] \), recall from step (5) and (7) * in Figure 4. If we “peel off” \( m_{0id}, \ldots, m_{nids} \) from \( m_{0id}^\text{sid} \) via XOR, the result is exactly the encoding of \( k^\text{sid} \) by design. Because all keyword-value pairs of \( DB_1, \ldots, DB_n \) are consistent with that of \( DB_0 \), in step (9) * all other non-target values are peeled off except for the target value.

The privacy of the \( k^\text{sid} \) is intact if no more than \( t \) Servers collude. This is because up to \( t \) Servers can observe up to \( t \) of \( m_{0id}^\text{sid} \)'s which are pseudorandom as intended by the collusion threshold during \( \text{TAG} \).

5.2.4 Query Phase

In this section, we will expand on the special case in the last section to describe the full Query phase when \( S \neq \phi \) in Algorithm 1. This enables the client to retrieve any values in \( DB_0 \) without leaking to the servers whether the query keyword is in sPIR or mPIR. Below, we follow Figure 5 to describe step (6) - (9). We refer to step (6), (7) * - (9) * from Section 5.2.3

Input

\( DB_0, M_0, \text{spk} \) from \( S_0 \)

\( DBid, M_id, m_{id}^\text{sid} \) from \( S_id, \forall id \in \text{AID} \)

\( M_0, S, \text{spk}, \text{ssk}, \text{tok}^\text{sid}, k^\text{sid} \) from \( C \)

Output

\( a_{sid} \) to \( C \)

Query Protocol for Session \( sid \)

\( \circ \) This is the same as step (6) in Section 5.2.3

\( \circ \) \( C \) decides on a query keyword \( k^\text{sid} \in M_0 \cup S \) and generates a hybrid-PIR query for \( S_0 \)

\[
(m_{0id}^\text{sid}, sq^\text{sid}) \leftarrow \text{QUERY}(M_0, S, \text{spk}, tok^\text{sid}, k^\text{sid})
\]

where \( sq^\text{sid} \) is the sPIR query and \( m_{0id}^\text{sid} \) mPIR query. \( \text{QUERY} \) is described in Algorithm 5 below. \( C \) sends \( (m_{0id}, sq^\text{sid}) \) to \( S_0 \).

The hybridization of the mPIR-sPIR query is visually demonstrated in Figure 6. First, step (7) * is followed to generate \( m_{0id}^\text{sid} \). (Note that if \( k^\text{sid} \not\in M_0 \), then \( m_{0id}^\text{sid} \) is a “blank” token as per Algorithm 4.)

Next, if \( k^\text{sid} \in S \), then \( k^\text{sid} \) is encoded by its index position in \( S \) as an input to \( S\text{QUERY} \); otherwise if \( k^\text{sid} \in M_0 \), then it is encoded as the last index position as an input to \( S\text{QUERY} \). In this example, because \( k^\text{sid} = c \in M_0 \), it is encoded as 3, the 0-based index of the last position.

### Notations

- \( \perp \), an empty string/value

#### Algorithm 5 Query Generation.

1. procedure \( \text{QUERY}(M_0, S, \text{spk}, tok, k) \)
2. \( m_{0id} \leftarrow \text{MQUERY}(M_0, tok, k) \)
3. \( sq \leftarrow \perp \)
4. if \( S \neq \phi \) then
5. \( \circ \) if \( k \in S \) then
6. \( i \leftarrow \text{index}(k, S) \)
7. \( sq \leftarrow S\text{QUERY} \langle \text{spk}, i, |S| + 1 \rangle \)
8. else
9. \( sq \leftarrow S\text{QUERY} \langle \text{spk}, |S|, |S| + 1 \rangle \)
10. end if
11. end if
12. return \( (m_{0id}, sq) \)
13. end procedure

\( \circ \) \( S_0 \) receives the query \( (m_{0id}^\text{sid}, sq^\text{sid}) \) from \( C \) and generates the reply

\[
r^\text{sid} \leftarrow \text{REPLY}(M_0, DB_0, m_{0id}^\text{sid}, sq^\text{sid})
\]

where \( r^\text{sid} \) is the resulting hybrid PIR reply. \( \text{REPLY} \) is described in Algorithm 6 below. \( S_0 \) returns \( r^\text{sid} \) to \( C \).

#### Algorithm 6 Reply Generation.

1. procedure \( \text{REPLY}(M_0, DB_0, m_{0id}, sq) \)
2. \( m_{0id} \leftarrow \text{MREPLY}(M_0, DB_0, m_{0id}) \)
3. \( S \leftarrow \text{Keys}(DB_0) \backslash M_0 \)
4. \( r \leftarrow \perp \)
5. if \( S \neq \phi \) then
6. \( V \leftarrow \langle DB_0[k] \rangle_{k \in S} \)
7. \( r \leftarrow S\text{REPLY} \langle \text{spk}, V || m_{0id}, sq \rangle \)
8. else
9. \( r \leftarrow m_{0id} \)
10. end if
11. return \( r \)
12. end procedure

Figure 5 demonstrates the hybridized reply generation process. First, the mPIR reply is generated according to step (8), resulting in \( m_{0id}^\text{sid} \). (Note that
if $mr_{0}^{id}$ is not a blank query, then $mr_{0}^{id}$ is a genuine mPIR reply; otherwise, it is a random value. This fact, however, is unknown to $S_{0}$.) Next, the values of $DB_{0}$ is filtered with $S$ by key, resulting in $V$. $mr_{0}^{id}$ is appended to $V$ as the last item, i.e. $V||mr_{0}^{id}$. The sPIR query $sq^{sid}$ and $V||mr_{0}^{id}$ are input to the sPIR reply generation algorithm $SReply$ to produce a reply.

At this point, it becomes clear why the sPIR query $sq$ is generated the way it is as described in step 7. When $k^{sid} \in S$, $sq^{sid}$ is generated to target values in $V$, ignoring $mr_{0}^{id}$ which is the last item. However, when $k \in M_{0}$, $sq^{sid}$ is generated to target the last value, which is where $mr_{0}^{id}$ is located. The resulting reply $r^{sid}$ therefore encodes either a targeted value in $V$ when $k^{sid} \in S$ or $mr_{0}^{id}$ when $k^{sid} \in M_{0}$.

$C$ receives the reply $r^{sid}$ from $S_{0}$ and decodes it using the same query keyword $k^{sid}$ from step 7 to obtain the answer

$$a^{sid} \leftarrow \text{DECODE} \left(S, ssk, k^{sid}, (mr_{id}^{sid})_{id \in AID}, r^{sid} \right)$$

DECODE is described in Algorithm 7 below.

Algorithm 7 Decoding.

1: procedure $\text{DECODE}(S, ssk, k, (mr_{id}^{sid})_{id \in AID}, r)$
2: \hspace{1em} if $k \in S$ then
3: \hspace{2em} $a \leftarrow S\text{DECODE}(ssk, r)$
4: \hspace{1em} else
5: \hspace{2em} if $S \neq \emptyset$ then
6: \hspace{3em} $mr_{0} \leftarrow S\text{DECODE}(ssk, r)$
7: \hspace{2em} else
8: \hspace{3em} $mr_{0} \leftarrow r$
9: \hspace{2em} end if
10: \hspace{2em} $a \leftarrow \oplus_{id \in ID} mr_{id}$
11: \hspace{1em} end if
12: \hspace{1em} return $a$
13: end procedure

First, the reply $r^{sid}$ from $S_{0}$ is decoded using secret key $ssk$ in the sPIR decoding algorithm $S\text{DECODE}$. If $k^{sid} \in S$, then $C$ already has the final answer; otherwise, $C$ obtains $mr_{0}^{id}$ and follows step 9*. Here, $C$ XOR all $mr_{0}^{id}, \ldots, mr_{3}^{id}$ to obtain the answer.

For privacy, it is important that regardless of whether $k^{sid} \in S$, $mr_{1}^{id}, \ldots, mr_{n}^{id}$ must be downloaded by $C$ from $S_{1}, \ldots, S_{n}$, respectively. If this step is skipped, then $S_{1}, \ldots, S_{n}$ can infer that $k^{sid} \in S$.

Figure 5: Query Phase Workflow Workflow of hybrid components in during Query phase to demonstrate how keyword-value pairs in $DB_{0}$ can be queried with the client’s query key.
5.2.5 Catalog Intersection

Catalog Intersection protocol is a part of Synchronization phase to reduce the communication cost of transferring catalog intersections. The idea is as follows: given that \( C \) has already obtained \( \text{Cat}_0 \), then \( \text{Cat}'_\text{id} = \text{Cat}_\text{id} \cap \text{Cat}_0 \) can be compressed with a hash function (we call these hashes *digests*) for transporting. We describe Catalog Intersection protocol below.

**Input**

\[ \begin{align*}
\triangleright & \quad \text{Cat}_\text{id} \quad \text{from} \quad S_{\text{id}}, \forall \text{id} \in \text{AID} \\
\triangleright & \quad \text{Cat}_0 \quad \text{from} \quad C
\end{align*} \]

**Output**

\[ \begin{align*}
\triangleright & \quad \left( \text{Cat}'_{\text{id}} \right)_{\text{AID}}, \left( \text{Dig}_{\text{id}} \right)_{\text{AID}} \quad \text{to} \quad C
\end{align*} \]

**Catalog Intersection Protocol**

1. For each \( \text{id} \in \text{AID}, S_{\text{id}} \) generates digests from \( \text{Cat}_\text{id} \)

\[ \text{Dig}_{\text{id}} \leftarrow \{ G((k, h)) \mid (k, h) \in \text{Cat}_\text{id} \} \]

where \( G(\cdot) \) is a universal hash function with short hash values, chosen at random from the universal family by \( C \). (A shorter hash length will reduce the communication cost but increase the probability of collision.) A digest is therefore a hashed representation of a keyword-hash pair in the catalogs.

2. \( C \) downloads all the digest sets \( \left( \text{Dig}_{\text{id}} \right)_{\text{AID}} \) from \( S_1, \ldots, S_n \). For each \( \text{id} \in \text{AID}, C \) remaps digest sets into catalog intersections as follows:

\[ \text{Cat}'_{\text{id}} \leftarrow \text{Map}\{ (k, h) \in \text{Cat}_0 \mid G((k, h)) \in \text{Dig}_{\text{id}} \} \]

5.2.6 Keyword Synchronization

Keyword Synchronization protocol is a part of Synchronization phase to reduce the communication cost of transferring mPIR keyword sets \( M_{\text{id}} \) from \( C \) to \( S_{\text{id}} \). The idea is to map each keyword in \( M_{\text{id}} \) to its corresponding index in \( \text{Dig}_{\text{id}} \) for transport, which can then be remapped to \( M_{\text{id}} \) on the server side. We describe Keyword Synchronization protocol below.

**Input**

\[ \begin{align*}
\triangleright & \quad \text{Dig}_{\text{id}}, \text{Cat}_\text{id} \quad \text{from} \quad S_{\text{id}}, \forall \text{id} \in \text{ID} \\
\triangleright & \quad (M_{\text{id}})_{\text{id}}, (\text{Dig}_{\text{id}})_{\text{id}} \quad \text{from} \quad C
\end{align*} \]

**Output**

\[ \begin{align*}
\triangleright & \quad M_{\text{id}} \quad \text{to} \quad S_{\text{id}}, \forall \text{id} \in \text{ID}
\end{align*} \]

**Keyword Synchronization Protocol**

1. For all \( \text{id} \in \text{ID}, C \) converts \( M_{\text{id}} \) to the set of indices corresponding to the ordering of \( \text{Keys}(\text{Cat}_0) \) for main server and of \( \text{Dig}_{\text{id}} \) for assisting servers:

\[ I_0 \leftarrow \text{KEYToINDEX}_0 (M_0, \text{Cat}_0) \]

\[ \forall \text{id} \in \text{AID} : I_{\text{id}} \leftarrow \text{KEYToINDEX}_\text{id} (M_{\text{id}}, \text{Dig}_{\text{id}}, \text{Cat}_\text{id}) \]

**KEYToINDEX** is described in Algorithm 8 below. \( C \) then sends \( I_{\text{id}} \) to \( S_{\text{id}} \). \( I_{\text{id}} \) is a small, indexed representation of \( M_{\text{id}} \) which can be transported at a low communication cost.

**Algorithm 8 Key-to-Index Mapping**

1: procedure \( \text{KEYToINDEX}_0 (M_0, \text{Cat}_0) \)
2: \[ I_0 \leftarrow \{ i \mid k \in M_0, i = \text{index}(k, \text{Keys} (\text{Cat}_0)) \} \]
3: return \( I_0 \)
4: end procedure
5: procedure \( \text{KEYToINDEX}_\text{id} (M_{\text{id}}, \text{Dig}_{\text{id}}, \text{Cat}_\text{id}) \)
6: \[ D \leftarrow \{ G((k, h)) \mid k \in M_{\text{id}}, h = \text{Cat}_\text{id}[k] \} \]
7: \[ I_{\text{id}} \leftarrow \{ i \mid d \in D, i = \text{index}(d, \text{Dig}_{\text{id}}) \} \]
8: return \( I_{\text{id}} \)
9: end procedure

2. For each \( \text{id} \in \text{ID}, S_{\text{id}} \) receives \( I_{\text{id}} \) and remaps it back to \( M_{\text{id}} \)

\[ M_0 \leftarrow \text{INDEXToKEY}_0 (I_0, \text{Cat}_0) \]

\[ \forall \text{id} \in \text{AID} : M_{\text{id}} \leftarrow \text{INDEXToKEY}_\text{id} (I_{\text{id}}, \text{Dig}_{\text{id}}, \text{Cat}_\text{id}) \]

**INDEXToKEY** is described in Algorithm 9 below.

**Algorithm 9 Index-to-Key Mapping**

1: procedure \( \text{INDEXToKEY}_0 (I_0, \text{Cat}_0) \)
2: \[ K \leftarrow \text{Keys} (\text{Cat}_0) \]
3: \[ M_0 \leftarrow \{ k \in K \mid i \in I_0, \text{index}(k, K) = i \} \]
4: return \( M_0 \)
5: end procedure
6: procedure \( \text{INDEXToKEY}_\text{id} (I_{\text{id}}, \text{Dig}_{\text{id}}, \text{Cat}_\text{id}) \)
7: \[ D \leftarrow \{ d \in \text{Dig}_{\text{id}} \mid i \in I_{\text{id}}, \text{index}(d, \text{Dig}_{\text{id}}) = i \} \]
8: \[ M_{\text{id}} \leftarrow \{ k \mid (k, h) \in \text{Cat}_\text{id}, G((k, h)) \in D \} \]
9: return \( M_{\text{id}} \)
10: end procedure

5.2.7 Catalog Intersection and Keyword Synchronization in Relation to Set Reconciliation

Set reconciliation is an active research area to find the symmetric difference between two sets from two remote parties over a network at a low communication cost. Notably, Eppstein et al. \[8\] propose a practical scheme whose...
communication cost is asymptotically linear in the size of the symmetric difference. One therefore could wonder if these schemes would apply to our Catalog Intersection and Keyword Synchronization problem. This is because Catalog Intersection and Keyword Synchronization are essentially about finding remote set intersections, which is implied by set symmetric differences. We argue that our solution as presented above solve this problem more efficiently by killing two birds with one stone: instead of apply two separate instances of set reconciliation to Catalog Intersection and Keyword Synchronization (and therefore paying the cost twice), we use the ordering information provided by digest sets Dig_{id} to reduce keyword sets M_{id} to sets of indices, and therefore reduce the upload cost by the client (which is especially important in an asymmetric network connection). It is not clear how this cost-saving technique can be applied to existing set reconciliation schemes in literature.

5.2.8 Keyword Compression

When query keywords are long, this can result a high communication cost during Synchronization phase since all keywords in catalogs must be transported to the client. For this reason, we aim to reduce this cost by representing each query keyword k by its hash H(k), where H(·) is a universal hash function with short hash values, chosen from the universal family by the client. When querying keyword k, the client computes the hash and uses it as a keyword to query instead.

Despite the cost reduction, this can introduce a new problem when the keyword space is large, which can either lead to long hash values or high collision (i.e. high false-positive rate). Our suggested workaround for this problem is the keep to hash values short (and therefore high collision rate) but let the servers modify the databases lead to long hash values or high false-positive rate. Our suggested workaround for this problem when the keyword space is large, which can either

5.3 Implementation

Synchronized APIR is implemented in Rust, where keyword compression in Section 5.2.8 is implemented by default. The universal hash functions H, G, and H are instantiated with Google’s CityHash (https://github.com/google/cityhash), each seeded with a random number by the client. The variable-length PRG is instantiated with the stream cipher ChaCha12 [7], which provides 256-bit security. sPIR is instantiated with SealPIR using SEAL v3.2.0 (https://github.com/ndokmai/sealpir-rust). We modify the SealPIR library to permit extra operations as required by Synchronized APIR. By default, this version of the library sets the degree of ciphertext polynomial to 2,048, and the size of the coefficients to 54 bits, which provides 128-bit security [6]. sPIR instantiation with OnionPIR [11] will be provided in future work.

The open-source implementation of Synchronized APIR is available at https://github.com/ndokmai/assisted-pir

6 Analysis

In this section, we provide theorems and proof sketches for Synchronized APIR correctness and privacy. In addition, we provide an analysis for a loose upper bound for the probability of failure of Synchronized APIR in the event that there are hash collisions. We defer full proofs and an analysis with a tighter bound to future work.

6.1 Correctness

We will prove the correctness of Synchronized APIR by first assuming that the universal hash functions in the scheme provide no hash collisions. Then, we provide an analysis for the probability of no hash collisions, which implies an upper bound for the probability of failure of Synchronized APIR.

Theorem 1 (Synchronized APIR Correctness). Suppose an sPIR-correct scheme IP^{APIR} and that the universal hash functions H, G, and H provide no collisions. Following Definition 4.2, the Synchronized APIR scheme IP^{SynAPIR} is correct for any server set ID = {0, 1, ..., n} and AID = {1, 2, ..., n}, collision threshold 1 ≤ t ≤ n, databases DB_{id} for all id ∈ ID, l Query sessions, and k_{sid} ∈ Keys(DB_0) for all sid ∈ [l]. That is, if

- Synchronization Protocol takes inputs (DB_{id})_{ID}, t and outputs (Cat_{id})_{ID}, (Cat_{lid})_{AID}, (M_{id})_{ID}, S
- Setup Protocol takes inputs l, (M_{id})_{ID} and outputs spk, ssk, (m_{sid})_{id ∈ [l], id ∈ AID}, (tk_{sid})_{id ∈ [l]}
- For all sid ∈ [l], Query Protocol for session sid takes inputs (DB_{id})_{ID}, (M_{id})_{ID}, S, (m_{sid})_{id ∈ AID}, spk, ssk, tk_{sid}, k_{sid} and outputs a_{sid} then for all sid ∈ [l], DB_0[k_{sid}] = a_{sid}.

Proof sketch. First, by assuming that the universal hash functions provide no collisions, we can claim that
• $H(k)$ perfectly represents keyword $k$, so $k$ can be efficiently replaced with $H(k)$.

• $H(v)$ perfectly represents value $v$, so catalog $Cat_id$ perfectly represents database $DB_{id}$. Therefore tagging the catalogs is the same as tagging the databases themselves.

• $G((k, h))$ perfectly represents $(k, h)$, so Catalog Intersection Protocol and Keyword Synchronization Protocol are trivially correct.

Next, we consider the properties of the tags $(M_id)_{ID}$. By construction of Tag (Algorithm 1), for each keyword $k \in Keys(DB_0)$,

1. either $k \in M_0$ or $k \in S$ but not both

2. if $k \in M_0$ and $k \in M_id$ for any $id \in AID$, then $DB_0[k] = DB_{id}[k]

3. $\bigcup_{k \in M_0} \bigcup_{id \in C_k} \{ (id, k) \} = \bigcup_{id \in AID} \bigcup_{k \in M_id} \{ (id, k) \}

Property 3. is a consequence of the fact that by construction, $M_id = \{ k \in M_0 \mid id \in C_k \}$ and $C_k = \{ id \in AID \mid k \in M_id \}$.

If $k^{sid} \in S$, then by construction of QUERY (Algorithm 5), $sq^{sid}$ targets item $\text{index}(k^{sid}, S)$ in the filtered database $DB_0[k]_{k \in S}$ in REPLY (Algorithm 6), which is exactly $DB_0[k^{sid}]$. By sPIR correctness of II^mPIR, we conclude that $a^{sid} = DB_0[k^{sid}]$.

If $k^{sid} \in M_0$, we consider how the mPIR token $tok^{sid}$ and queries $mq^{sid}_{id}$ are generated. Consider each bit of $tok^{sid}$ by construction in MTOKEN (Algorithm 2): there exists $k \in M_0$ such that $\text{token}_{id}^{sid}[\text{index}(k, M_0)] = \bigoplus_{id \in C_k} \text{mq}_{id}^{sid}[\text{index}(k, M_0)]$

where $C_k$ is a set of id’s such that $DB_0[k] = DB_{id}[k]$ by property 2. stated above. This implies

$\text{token}_{id}^{sid}[\text{index}(k, M_0)] \cdot DB_0[k]$

$= \bigoplus_{id \in C_k} \text{mq}_{id}^{sid}[\text{index}(k, M_id)] \cdot DB_0[k]$

$= \bigoplus_{id \in C_k} \text{mq}_{id}^{sid}[\text{index}(k, M_id)] \cdot DB_{id}[k]$

And therefore, by the equation above and property 3.,

$\text{token}_{id}^{sid}(DB_0[k])_{k \in M_0}$

$= \bigoplus_{k \in M_0} \text{token}_{id}^{sid}[\text{index}(k, M_0)] \cdot DB_0[k]$

$= \bigoplus_{id \in AID} \bigoplus_{k \in M_id} \text{mq}_{id}^{sid}[\text{index}(k, M_id)] \cdot DB_{id}[k]$

$= \bigoplus_{id \in AID} \text{mq}_{id}^{sid} \cdot (DB_{id}[k])_{k \in M_id}$

Next, because

$\text{mq}_{0}^{sid}[\text{index}(k^{sid}, M_0)] = \text{token}_{id}^{sid}[\text{index}(k^{sid}, M_0)] \oplus 1$

by construction of MQUERY (Algorithm 4), we have

$\bigoplus_{id \in ID} \text{mq}_{id}^{sid} \cdot (DB_{id}[k])_{k \in M_id}$

$= \bigoplus_{id \in ID} \text{mq}_{id}^{sid} \cdot (DB_{id}[k])_{k \in M_id}$

$= \bigoplus_{id \in ID} \text{mq}_{id}^{sid} \cdot (DB_{id}[k])_{k \in M_id} \oplus (1 \cdot DB_0[k^{sid}])$

$= \bigoplus_{id \in AID} \text{mq}_{id}^{sid} \cdot (DB_{id}[k])_{k \in M_id}$

$= DB_0[k^{sid}]$

Thus proven the correctness of mPIR-Only Query Protocol.

To show that the general case in Query Protocol is correct, we observe that when $k^{sid} \in M_0$, $sq^{sid}$ targets targets item $|S|$ by construction of QUERY (Algorithm 5). That is, $sq^{sid}$ targets targets $mr^{sid}_{id}$ by construction of REPLY (Algorithm 6). By sPIR correctness of II^mPIR, we conclude that in DECODE (Algorithm 7), $SDECODE(ssk, r^{sid}) = mr^{sid}_{0}$. And finally,

$a^{sid} = \bigoplus_{id \in ID} mr^{sid}_{id}$

$= \bigoplus_{id \in ID} \text{mq}_{id}^{sid} \cdot (DB_{id}[k])_{k \in M_id}$

$= DB_0[k^{sid}]$

by correctness of mPIR-Only Query Protocol.

Next, we analyze the probability that there is no hash collision when the universal hash functions are used in the context of Synchronized APIR.

**Theorem 2** (Synchronized APIR Hash Non-collision). Let $H, G, H$ be random universal hash functions from universal families whose hash values are of length $l_1, l_2, l_1$, respectively. For any $ID = \{0, \ldots, n \}$ and databases $DB_{id}, id \in ID$, let $U := \bigcup_{id \in ID} DB_{id}$. We define the following non-collision events:

1. For any $k \in M_0$ and $id \in AID$, $DB_0[k] = DB_{id}[k]$, so $U \subseteq DB_{id}$.

2. For any $k \in M_0$ and $id \in AID$, $DB_0[k] = DB_{id}[k]$, so $U \subseteq DB_{id}$.

3. For any $k \in M_0$ and $id \in AID$, $DB_0[k] = DB_{id}[k]$, so $U \subseteq DB_{id}$.
• $E_1(U) := \forall k, k' \in \text{Keys}(U) : k \neq k' \implies H(k) \neq H(k')$

• $E_2(U) := \forall v, v' \in \text{Values}(U) : v \neq v' \implies H(v) \neq H(v')$

• $E_3(U) := \forall (k, v), (k', v') \in U : G((H(k), H(v))) \neq G((H(k'), H(v'))) \implies (H(k), H(v)) \neq (H(k'), H(v'))$

Let $p(m, d) := e^{-\frac{(m-d) \cdot d}{d}}$, then

$$\Pr[E_1(U), E_2(U), E_3(U)] = \Pr[E_1(U)] \approx p(|\text{Keys}(U)| \cdot 2^{hc}) \cdot p(|\text{Values}(U)| \cdot 2^{hn})$$

Proof Sketch. First we consider $E_1(U)$ and $E_2(U)$, which describe the “birthday” problem where no two individuals share the same birthday. For $E_1(U)$ the number of birthdays is $2^{hc}$ and the number of individuals is $|\text{Keys}(U)|$; likewise for $E_2(U)$, the number of birthdays is $2^{hn}$ and the number of individuals is $|\text{Values}(U)|$. By [12], we have

$$\Pr[E_1(U)] \approx p(|\text{Keys}(U)| \cdot 2^{hc})$$

$$\Pr[E_2(U)] \approx p(|\text{Values}(U)| \cdot 2^{hn})$$

Given that $E_1(U)$ and $E_2(U)$ occur, $E_3(U)$ also describes the same birthday problem with $2^{hc}$ birthdays and $|U|$ individuals. Therefore,

$$\Pr[E_3(U) \mid E_1(U), E_2(U)] \approx p(|U| \cdot 2^{hc})$$

Finally, since $E_1(U)$ and $E_2(U)$ are independent events, we have

$$\Pr[E_1(U), E_2(U), E_3(U)] = \Pr[E_3(U) \mid E_1(U), E_2(U)] \Pr[E_1(U), E_2(U)] = p(|U| \cdot 2^{hc}) \cdot p(|\text{Keys}(U)| \cdot 2^{hc}) \cdot p(|\text{Values}(U)| \cdot 2^{hn})$$

What does Theorem 1 imply for Theorem 2? By definition, we know that if events $E_1(U), E_2(U), E_3(U)$ take place, then Theorem 2 is true with probability 1. That is,

$$\Pr[\Pi^{\text{SynAPIR}} \text{ is correct} \mid E_1(U), E_2(U), E_3(U)] = 1$$

So,

$$\Pr[\Pi^{\text{SynAPIR}} \text{ is correct}] \geq \Pr[\Pi^{\text{SynAPIR}} \text{ is correct} \mid E_1(U), E_2(U), E_3(U)] \cdot \Pr[E_1(U), E_2(U), E_3(U)] = \Pr[E_1(U), E_2(U), E_3(U)]$$

And therefore

$$\Pr[\Pi^{\text{SynAPIR}} \text{ is incorrect}] \leq 1 - \Pr[E_1(U), E_2(U), E_3(U)]$$

We note that the probability of incorrectness is not per query but per query keyword; that is, the same keyword either succeeds or fails every time from hash collisions. This implies that the amount of failure does not scale with query traffic loads but with the total number of query keywords made.

The upper bound here can be improved because $E_1(U), E_2(U), E_3(U)$ are not necessary conditions for the correctness of Synchronized APIR, because there are certain hash collisions that simply do not affect the correctness of the scheme. For example, in the catalogs, it is not an issue if $v \neq v'$ but $H(v) = H(v')$ as long as pairs $(k, H(v))$ and $(k', H(v'))$ are in the catalogs and $k \neq k'$. In future work, we will provide an analysis with a tighter bound taking into account all of these nuances.

6.2 Privacy

To prove the privacy of Synchronized APIR, we first provide the privacy definition of Synchronized APIR which directly corresponds to the privacy definition of APIR in Definition 1.3. Then, we show that mPIR queries are in fact pseudorandom given the variable-length PRG, even if some of the PRG seeds are predetermined. Leveraging this fact and the privacy of sPIR scheme, we show that Synchronized APIR is privacy-preserving.

First, let us provide the privacy definition of Synchronized APIR below. Intuitively, Synchronized APIR is privacy-preserving if the attacker is unable to distinguish between queries generated by keyword $k_0$ and $k_1$ during one session, even if the attacker has oracle access to all other query sessions and is provided with some of the PRG seeds used to generate the mPIR queries.

Definition 6.1 ($\lambda, t$)-Synchronized APIR Privacy. Following Definition 1.3, we define the privacy experiment $\text{Priv}_{\lambda, \Pi^{\text{SynAPIR}}}(1^\lambda)$ for the Synchronized APIR scheme given a $\lambda$-sPIR privacy-preserving scheme $\Pi^{\text{PIR}}$ by Definition 1.2 and variable-length PRG $\text{PRG}(s, t) \rightarrow r$ below.

1. $A$ chooses the number of Query sessions $l = \text{poly}(\lambda)$, well-formed catalog $\text{Catalog}_0$ and digests $(\text{Dig}_{\text{id}})_{\text{ID}}$, and outputs $(l, \text{Catalog}_0, (\text{Dig}_{\text{id}})_{\text{ID}})$. $A$ chooses a collision set $C \subset \text{ID}$ such that $|C| \leq t$ and sends $C$ to the oracle $O$.

2. The following steps are followed to generate public and secret parameters:

   a) **Synchronization:** Step 2 of Catalog Intersection Protocol in Section 5.2.5 is followed with
inputs \( \text{Cat}_0, (\text{Dig}_{\text{id}})_{\text{AID}} \) to produce \( (\text{Cat}'_{\text{id}})_{\text{AID}} \). Catalogs are tagged by
\[
((M_{\text{id}})_{\text{ID}}, S) \leftarrow \text{TAG}(\text{Cat}_0, (\text{Cat}'_{\text{id}})_{\text{AID}}, t)
\]
as per step (2) of Synchronization Protocol in Section 5.2.1.

(b) **Setup:** Setup Protocol in Section 5.2.2 is followed with inputs \((M_{\text{id}})_{\text{ID}}\) and \(l\) and outputs \((\text{spk}, \text{ssk}, \left(\text{tok}^{\text{id}}\right)_{[l]})\) (the \(\text{mq}\) output is ignored). The PRG seeds \((s_{\text{id}})_{\text{AID}}\) generated during this step is also saved.

\((\text{spk}, (M_{\text{id}})_{\text{ID}}, S)\) is given to \(\mathcal{A}\) as public parameters.

3. Initialize session ID \(\text{id} = 1\). \(\mathcal{A}\) is given oracle access to \(\text{QUERY}\) in the following way:
   
   (a) At \(\text{id} = 1\), \((s_{\text{id}})_{\text{AID}}\) is given to \(\mathcal{O}\), and \(\mathcal{O}\) gives \((s_{\text{id}})_{C \setminus \{0\}}\) to \(\mathcal{A}\) (recall that \(s_{\text{id}}\) is a PRG seed).
   
   (b) \(\mathcal{A}\) chooses and outputs \(k^{\text{id}} \in \text{Keys}(\text{Cat}_0)\).
   
   (c) A query is generated by
   
   \[
   (\text{mq}^{\text{id}}, \text{sq}^{\text{id}}) \leftarrow \text{QUERY}\left(M_0, S, \text{spk}, \left(\text{tok}^{\text{id}}\right)_{[l]}\right)
   \]

   following step (7) of Query Protocol in Section 5.2.4 \((\text{mq}^{\text{id}}, \text{sq}^{\text{id}})\) is given to \(\mathcal{O}\).

   (d) If \(0 \in C\), \(\mathcal{O}\) gives \((\text{mq}_0^{\text{id}}, \text{sq}_0^{\text{id}})\) to \(\mathcal{A}\); otherwise, \(\mathcal{O}\) gives an empty value \(\perp\) to \(\mathcal{A}\).

   (e) \(\text{id} \leftarrow \text{id} + 1\)

4. During some session \(\text{id} = \text{id}^* \leq l\),
   
   (a) \(\mathcal{A}\) chooses \(k_0^{\text{id}^*}, k_1^{\text{id}^*} \in \text{Keys}(\text{Cat}_0)\) and outputs \((k_0^{\text{id}^*}, k_1^{\text{id}^*})\).
   
   (b) A uniformly random bit is sampled \(b \leftarrow \{0, 1\}\).
   
   (c) A query is generated
   
   \[
   (\text{mq}^{\text{id}^*}_b, \text{sq}^{\text{id}^*}_b) \leftarrow \text{QUERY}\left(M_0, S, \text{spk}, \left(\text{tok}^{\text{id}^*}\right)_{k_b^{\text{id}^*}}\right)
   \]

   and \((\text{mq}^{\text{id}^*}_{\bar{b}}, \text{sq}^{\text{id}^*}_{\bar{b}})\) is given to \(\mathcal{O}\).

   (d) If \(0 \in C\), \(\mathcal{O}\) gives \((\text{mq}^{\text{id}^*}_0, \text{sq}^{\text{id}^*}_0)\) to \(\mathcal{A}\); otherwise, \(\mathcal{O}\) gives an empty value \(\perp\) to \(\mathcal{A}\).

   (e) \(\text{id} \leftarrow \text{id}^* + 1\)

5. \(\mathcal{A}\) is given more oracle access to \(\text{QUERY}\) until \(\text{id} = l\).

6. \(\mathcal{A}\) outputs \(b^* \in \{0, 1\}\). The output of the experiment is defined to be 1 if \(b^* = b\) and 0 otherwise.

**Theorem 3 (mPIR Query Pseudorandomness).** Suppose a variable-length PRG \(\text{PRG}(s, \ell) \rightarrow r\) where \(|s| = \lambda\) and \(|r| = \ell = \text{poly}(\lambda)\). For any \(l := \{0, \ldots, n\}\), \(\mathcal{AID} := \{1, \ldots, n\}, 1 \leq t \leq n\), well-formed catalog \(\text{Cat}_0\) and catalog intersections \(\text{Cat}'_{\text{id}}\), \(\text{id} \in \mathcal{AID}\), \(l\) Query sessions, \(\text{sid} \in [l]\), \(k_{\text{id}}^{\text{sid}} \in \text{Keys}(\text{Cat}_0)\), \(A \subset \mathcal{AID}\) such that \(|A| < t\), and \((s_{\text{id}})_{\mathcal{A}} \in \{0, 1\}^{1, \lambda^{\lceil\mathcal{A}\rceil}}\); define

- \(\forall \text{id} \in \mathcal{AID} \setminus A : s_{\text{id}} \leftarrow s \{0, 1\}^{\lambda}\)
- \((M_{\text{id}})_{\text{AID}} : S) = \text{TAG}(\text{Cat}_0, (\text{Cat}'_{\text{id}})_{\text{id} \in \mathcal{AID}})
- \forall \text{id} \in \mathcal{AID} : (\text{mq}_{\text{id}}^{\text{sid}})_{\text{id} \in [l]} = \text{PRG}(s_{\text{id}}, l \cdot (M_{\text{id}}))
- \forall \text{sid} \in [l] : \text{tok}^{\text{id}} = \text{MTOKEN}((M_{\text{id}})_{\text{ID}}, (\text{mq}^{\text{id}})_{\mathcal{AID}})
- \forall \text{sid} \in [l] : \text{mq}^{\text{id}}_{\text{ID}} = \text{MQUERY}(M_0, \text{tok}^{\text{id}}, k_{\text{id}}^{\text{id}})
- \text{aux} = ((M_{\text{id}})_{\text{ID}}, A, (s_{\text{id}})_{A}, (k_{\text{id}}^{\text{id}})_{[l]})

Then for all PPT distinguisher \(D\), there exists a negligible function \(\text{negl}\) such that

\[
\Pr[D(r, \text{aux}) = 1] - \Pr[D((\text{mq}^{\text{id}})_+[l], \text{aux}) = 1] \leq \text{negl}(\lambda)
\]

where \(r \leftarrow s \{0, 1\}^{\lceil M_0 \rceil}\) is sampled uniformly at random.

**Proof sketch.** We will use a hybrid argument to prove the theorem as follows.

**Part 1:** For any \(B \subset \mathcal{AID}\) such that \(A \subseteq B\) and \(|B| = t - 1\), we will slightly modify the definition of \(\text{mq}^{\text{id}}_{\text{ID}}\) above called \(\text{mq}^{\text{id}}_{\text{ID}}\) where if \(\text{id} \in \mathcal{AID} \setminus B\), then for all \(\text{sid} \in [l]\), \(\text{mq}^{\text{id}}_{\text{id}} \leftarrow s \{0, 1\}^{\lceil M_0 \rceil}\) is sampled uniformly at random; everything else remains unchanged i.e. \(\forall \text{id} \in B, \forall \text{sid} \in [l] : \text{mq}^{\text{id}}_{\text{id}} = \text{mq}^{\text{id}}_{\text{id}}\). This results in the new \(\text{mq}^{\text{id}}_{\text{id}}\), \(\text{sid} \in [l]\).

We claim that these two ensembles are perfectly indistinguishable

\[
\langle r, \text{aux} \rangle \equiv ((\text{mq}^{\text{id}}_{\text{ID}})_+[l], \text{aux})
\]

16
by observing the construction of \( m_{0|sid} \). Let

\[
\text{tok}^{\text{sid}} = \text{MTOKEN} \left( (M_{id})_{ID} \cdot \left( m_{id}^{\text{sid}} \right)_{id \in \text{AID}} \right)
\]

and consider each bit of \( \text{tok}^{\text{sid}} \): for each \( k \in M_0 \),

\[
\text{tok}^{\text{sid}}[\text{index}(k, M_0)] = \bigoplus_{id \in C_k} m_{id}^{\text{sid}}[\text{index}(k, M_{id})]
\]

\[
= \bigoplus_{id \in C_k \setminus B} m_{id}^{\text{sid}}[\text{index}(k, M_{id})]
\]

\[
\bigoplus_{id \in C_k \cap B} m_{id}^{\text{sid}}[\text{index}(k, M_{id})]
\]

Since \( |C_k| = t \) by construction and \( |B| = t - 1 \), we know that \( C_k \setminus B \) is not empty. Because for all \( id \in C_k \setminus B \), \( m_{id}^{\text{sid}} \) is uniformly random by definition, we conclude that \( \bigoplus_{id \in C_k \setminus B} m_{id}^{\text{sid}}[\text{index}(k, M_{id})] \) is uniformly random, and so is \( \text{tok}^{\text{sid}}[\text{index}(k, M_0)] \) and \( \text{tok}^{\text{sid}} \) for all \( sid \in [l] \). Thus by construction of MQUERY, \( \text{tok}^{\text{sid}} \) is also uniformly random for all \( sid \in [l] \). This proves the prefect indistinguishability.

**Part 2:** Given the variable-length PRG, we claim that these two ensembles are computationally indistinguishable

\[
\langle (m_{0|sid}^{\text{id}})_{[l]}, \text{aux} \rangle \equiv \langle (m_{0|sid}^{\text{id}})_{[l]}, \text{aux} \rangle
\]

via a reduction proof.

Suppose there exists a PPT distinguisher \( D \) who can distinguish between the two ensembles, we will construct an adversary \( A \) using \( D \) as a subroutine to play the following game:

The challenger \( C \) is tasking \( A \) to distinguish between a uniformly random string

\[
r_0 \leftarrow \{0, 1\} \sum_{id \in \text{AID} \setminus B} l \cdot |M_{id}|
\]

where \( B \) is defined in Part 1, and a pseudorandom string

\[
r_1 \leftarrow \text{PRG}(s_{id}, l \cdot |M_{id}|)_{id \in \text{AID} \setminus B}
\]

where each \( s_{id} \leftarrow \{0, 1\}^* \) is sampled uniformly at random. Upon given \( r_b, b \in \{0, 1\} \), \( A \) defines

\[
\left( \left( m_{id}^{\text{sid}, *} \right)_{id \in [l]} \right)_{id \in \text{AID} \setminus B} := r_b
\]

and \( m_{id}^{\text{sid}, *} := m_{id}^{\text{sid}} \) for the rest of \( id \in B \) and \( sid \in [l] \). Next, \( A \) constructs \( m_{0|sid}^{\text{sid}, *} \) out of \( m_{id}^{\text{sid}, *} \), \( id \in \text{AID} \) i.e.

\[
\forall sid \in [l]: \text{tok}^{\text{sid}, *} = \text{MTOKEN} \left( (M_{id})_{ID} \cdot \left( m_{id}^{\text{sid}, *} \right)_{AID} \right)
\]

and inputs \( ((m_{0|sid}^{\text{sid}, *})_{[l]}, \text{aux}) \) to \( D \). If \( D \) determines the input is the first ensemble, then \( A \) outputs 0; otherwise \( A \) outputs 1. By variable-length PRG assumption, we conclude that the advantage of \( A \) of guessing the correct string is negligible, and so the two ensembles are indistinguishable.

**Part 3:** By “transitivity” of Part 1 and 2,

\[
\langle r, \text{aux} \rangle \equiv \langle (m_{0|sid}^{\text{id}})_{[l]}, \text{aux} \rangle \equiv \langle (m_{0|sid}^{\text{id}})_{[l]}, \text{aux} \rangle
\]

\[
\implies \langle r, \text{aux} \rangle \equiv \langle (m_{0|sid}^{\text{id}})_{[l]}, \text{aux} \rangle
\]

thus proven the theorem. \( \square \)

Now that we have proven that mPIR are pseudorandom given the variable-length PRG, we will use a hybrid argument to show that the hybridization between mPIR and sPIR (given that the sPIR scheme is privacy-preserving) in Synchronized APIR results in a privacy-preserving scheme, even of some of the PRG seeds are leaked to the attacker.

**Theorem 4 (Synchronized APIR Privacy).** Suppose \( \Pi_{\text{mPIR}} \) is \( \lambda \)-sPIR privacy-preserving according to Definition 6.2 and \( \text{PRG}(s, l) \rightarrow r \) is a variable-length PRG, then Synchronized APIR scheme \( \Pi_{\text{SynAPIR}} \) is \( (\lambda, t) \)-APIR privacy-preserving according to Definition 6.1.

**Proof sketch.** We first observe that if \( 0 \not\in C \) in \( \text{Priv}_{A, \Pi_{\text{SynAPIR}}, t} \), i.e. main server \( S_0 \) is not compromised, then \( A \) does not obtain any information related to \( b \), which implies no advantage in guessing \( b^* \); the scheme is therefore trivially privacy-preserving. For the rest of this proof, we hence only focus on the scenario in which \( A \) has chosen \( 0 \in C \).

We want to show that the view of \( A \) in \( \text{Priv}_{A, \Pi_{\text{SynAPIR}}, t} \) is computationally indistinguishable between when \( b = 0 \) and \( b = 1 \). Formally, define the view of \( A \) in the experiment as

\[
\text{out}_{\text{view}, A} := \langle (k_{\text{sid}}^{\text{sid}, *})_{sid \neq \text{sid}^*}, (k_0^{*}, k_1^{*}) \rangle, \text{auxout}\)
\]

for the outputs of \( A \), where

\[
\text{auxout} := (l, \text{Cat}_0, \text{Di}_{\text{sid}} \cdot \text{AID} \cdot C)
\]

and

\[
\text{in}_{\text{view}, A}(b) := \langle (s_{\text{sid}})_{C \setminus \{0\}}, (m_{0|sid}^{\text{sid}, *}, \text{sq}_{\text{sid}, *}^{\text{sid}, *})_{sid \neq \text{sid}^*}, (m_{0|sid}^{\text{sid}, *}, \text{sq}_{\text{sid}, *}^{\text{sid}, *})_{aux}, \text{auxin} \rangle
\]

for the inputs of \( A \), where

\[
\text{auxin} := (\text{spk}, (M_{id})_{ID} \cdot S)
\]
and finally
\[
\text{view}_A(b) := \langle \text{outview}_A, \text{inview}_A(b) \rangle
\]

We want to show that
\[
\text{view}_A(0) \subseteq \text{view}_A(1)
\]
with respect to the security parameter $\lambda$ through the follows steps:

1. Similarly to $\text{inview}_A(b)$, we define
\[
\tilde{\text{inview}}_A(b) := \langle (s_{id})_{C \setminus \{0\}}, (mq_{0}^{\text{sid}}, sq_{0}^{\text{sid}})_{\text{sid} \neq \text{sid}^*}, (mq_{0}^{\text{sid}^*}, \tilde{s}_q) \rangle, \text{auxin}
\]
where $\tilde{s}_q \leftarrow \text{SQUERY}(\text{spk}, 0)$, and
\[
\tilde{\text{view}}_A(b) := \langle \text{outview}_A, \tilde{\text{inview}}_A(b) \rangle
\]

By $\lambda$-sPIR privacy assumption, we claim that
\[
\text{view}_A(0) \subseteq \tilde{\text{view}}_A(0)
\]

2. By Theorem 3, we claim that
\[
\tilde{\text{view}}_A(0) \subseteq \tilde{\text{view}}_A(1)
\]

because mPIR queries are pseudorandom.

3. Similarly to step 1, by $\lambda$-sPIR privacy assumption we claim that
\[
\tilde{\text{view}}_A(1) \subseteq \text{view}_A(1)
\]

4. By “transitivity”, we claim that
\[
\text{view}_A(0) \subseteq \text{view}_A(0) \subseteq \tilde{\text{view}}_A(1) \subseteq \text{view}_A(1)
\]

Thus proven the theorem.

7 Private DNS Query

In this section, we showcase the application of Synchronized APIR to achieve private DNS query for NS (Name Server) records in DNS cache servers. Indeed, the applicability of Synchronized APIR is not limited to NS records, but NS records make a great example because they are relatively stable with the time-to-live (TTL) of 2 days based on the ICANN’s .com zone file, which we will discuss next.

We assume a hypothetical setting in which there are multiple DNS cache servers who keep separate tables for different top-level domains (TLD) and types of records. Each server gathers the query statistics per each domain from non-private DNS traffic in the past 24 hours, and updates the cache table with the records of most queried domains. The server then builds a Synchronized APIR database out of this cache table. The cache table is assumed to be much smaller than the total number of records for efficiency.

The Synchronized APIR client chooses a group of Synchronized APIR servers: one as the main server and the rest as assisting servers. The parties then participate in the Synchronized APIR scheme.

For the application demonstrated here, the client must informs the servers she wishes to query the .com NS table. In the table, each domain is assumed to link to the collection of all the associated NS records. This implies that in a successful PIR query, the client will receive all the NS records for the queried domain, and decides which one to use.

7.1 NS Record Dataset

We analyze the .com Generic Top Level Domain (gTLD) zone file provided by ICANN available per request on https://czds.icann.org, accessed on June 16, 2022, to gather statistics about NS records. The data analysis is for the purpose of simulating cache tables for the PIR servers, which will be described in the next section. The .com zone file includes all record types, but we filter it for NS records only. Each NS record contains 1) domain name, 2) time-to-live (TTL), 3) class, 4) type, 5) resource record length, and 6) NS domain name. Each domain name may be linked to multiple NS records, and each record are of variable length, depending on how long the NS domain name is. We summarize the statistics in Table 1.

In this table, we draw particular attention to “max. |records| per domain”. “|records| per domain” here means that if a domain name is linked to record 1, record 2, ..., record $n$, then |records| per domain for this domain is $|\text{record 1}| + |\text{record 2}| + \ldots + |\text{record n}|$. “max.” indicates the maximum of this value across all the domains in the zone file, which is 759 bytes. This implies that the Synchronized APIR database values need to be at least 759 bytes in length to be able to hold all the NS records linked to each domain name for all the domain names. We thus parameterize the database values to be 1,024 bytes in length as a conservative measure.

7.2 Data Simulation Method

Although we have access to NS records in the previous section, what remains unknown is how independent DNS cache servers would behave in the real world. Specifically,
we need to know what NS records each server is holding in the cache table at a given time to be able to use this table as a database in Synchronized APIR. To simulate the servers’ cache tables for this purpose, we therefore need to make assumptions about 1) the statistical distribution of domain name popularity, and 2) the frequency of DNS queries over a period of time.

For 1), we assume that the popularity of domain name in DNS queries follows a Zipfian distribution \cite{10,15}, with the total number of ranks being the total number of domain names with NS records (159 million domains according to Table 1).

For 2), we assume that a cache server collects DNS query statistics from the normal (i.e. non-private) DNS service over a period of time; after which, the server updates the cache table with the most queried domain names during the period. To assume the number of queries (which follow a Zipfian distribution) over a period of time, we consider two scales of DNS operation: local and regional. The scale of operation implies the scale of traffic loads. In our setting, local means a US city, whereas regional means the entire US.

To get a sense of what the scales look like in the real world, we obtain DNS query statistics from ICANN Managed Root Server (IMRS) accessible on stats.dns.icann.org on June 18, 2022 to obtain the average queries-per-second (QPS) statistics for NS queries in a 24-hour window. For the local scale, we obtain the QPS within the city of Chicago; for the regional setting, we obtain the QPS within the entire US. The average QPS in a 24-hour window for the local setting is 256.3, which accumulates to 22 million queries in the 24-hour cycle. The average QPS in a 24-hour window for the regional setting is 7032.3, which accumulates to 608 million queries in the 24-hour cycle.

We follow these assumptions and parameters to simulate 3 cache servers for Synchronized APIR for local and regional experiment: 1 main and 2 assisting servers with collusion threshold of \( t = 2 \). The cache table size, or \(|DB|\), for the local experiment is \( 2^{13} \sim 8 \) thousand domains, and for the regional experiment \( 2^{16} \sim 66 \) thousand domains. We base the cache table sizes roughly on Cisco’s Caching DNS Capacity and Performance Guidelines \cite{1}. To simulate a cache table, we sample domain name ranks from a Zipfian distribution as many times as the number of queries in 24 hours for each server, where the most \(|DB|\) popular ranks are kept in the cache table. The actual keywords and values in the cache tables are randomly generated; this does not affect the performance of the scheme since the scheme is agnostic of the actual content of database keywords and values.

### Zipfian parameters

The next question is what appropriate popularity index \( s \) of the Zipfian distribution to use in the simulation. Wang \cite{15} and Jung et al. \cite{10} found the popularity index to be 0.98 and 0.91, respectively, in a local DNS setting at the time of the studies. We surmise that the distribution of domain name popularity is always sensitive to time and geography, and therefore there is no single distribution that is universally true and accurate.

Instead, what we aim to demonstrate in our experiments is Synchronized APIR in the most optimal conditions. Since Synchronized APIR’s most expensive computational and communication cost corresponds to the number of sPIR items i.e. \(|S|\), we want to find a Zipfian popularity index \( s \) that minimizes \(|S|\) to demonstrate the most optimal conditions for both the local and regional setting.

To achieve this, we simulate the cache tables for 3 Synchronized APIR servers at varying \( s \) and cache sizes for the local and regional setting (\( 2^{12}, 2^{13}, 2^{14} \) domains for local, and \( 2^{15}, 2^{16}, 2^{17} \) domains for regional). The results are shown in Figure 6. Here, the optimality is represented by the percentage of sPIR items in the main cache table, i.e. \(|S|/|DB_0| \times 100\%\). The results indices that \( s = 1.0 \) is an optimal parameter. (Coincidentally, this is close to the 0.98 and 0.91 in Wang and Jung et al.’s study, respectively.) We therefore choose \( s = 1.0 \) to evaluate Synchronized APIR in the next section.

### 7.3 Evaluation Settings

We summarize the parameters used to evaluate Synchronized APIR in Table 2 and 3.

To evaluate the scheme, we use the implementation specified in Section 5.3. The computing environment is a desktop computer with an Intel i9-10900 CPU and 64 GB of RAM running Ubuntu 20.04. The client and servers are simulated locally in the same computing environment, where each party occupies one CPU core; the

<table>
<thead>
<tr>
<th>Statistics</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>#records</td>
<td>382M</td>
</tr>
<tr>
<td>#domains</td>
<td>159M</td>
</tr>
<tr>
<td>avg. #records per domain</td>
<td>2.4</td>
</tr>
<tr>
<td>avg.</td>
<td>domain name</td>
</tr>
<tr>
<td>avg.</td>
<td>records</td>
</tr>
<tr>
<td>max.</td>
<td>records</td>
</tr>
<tr>
<td>TTL</td>
<td>2 days</td>
</tr>
</tbody>
</table>

Table 1: Statistics for NS records in ICANN’s .com gTLD Zone File (M=10⁷). “#records” indicates the number of records. “#domains” indicates the number of domains. “[#records] per domain” specifies total the size in bytes of all records linked each domain, where each NS record is encoded as TTL|CLASS|TYPE|LEN|NSDOMAIN. The TTL’s are of the same value of 172800 across all the records.
remote communication is via TCP loopback connections.

We measure the performance in terms of communication and computational cost for a single PIR query. The communication cost is measured separately in the amount of data uploaded and downloaded by the client in bytes. The computational cost is measured in the amount of CPU time spent by each party in milli-seconds.

Synchronized APIR is evaluated in two experiments: regional and local. In each experiment, a comparison is made between 3 schemes:

1. Baseline SealPIR with query dimensionality \(d = 2\) i.e. recursive PIR
2. Synchronized APIR with \(d = 1\) SealPIR
3. Synchronized APIR with \(d = 2\) SealPIR

The Baseline \(d = 1\) SealPIR scheme is omitted because it repeatedly crashed during the regional experiment in our trial runs regardless of parameters. We suspect this is due to a limitation in the SealPIR library, which is unable to handle databases larger than a certain size for \(d = 1\).

In Baseline SealPIR experiment, the keyword compression technique is applied to the list of domain names in the cache table; the compressed (i.e. hashed) keywords are sent to the client as the catalog, which allows the client to translate keywords to index positions in the database. In Synchronized APIR, the keyword compression technique is applied per description in Section 5.2.8.

SealPIR is configured by default to provide 128-bit security (see Section 5.3). The log of plaintext modulus is determined the noise budget in the underlying SEAL fully homomorphic encryption scheme (higher means more noise budget and higher communication cost); this is adjusted by trial and error to ensure the experiments do not fail from the noise after multiple runs.

The sPIR percentage statistics is calculated from the simulated databases. The probability of failure i.e. the probability that there is a hash collusion as per Theorem 2 is calculated from the hash length and simulated databases.

### 7.4 Results

The results for the communication costs of the experiments are shown in Table 4 and computational costs in Table 5. We shall refer to the Baseline \(d = 2\) SealPIR scheme as the baseline for comparison.

According to Table 4 the client using Synchronized APIR transmits 4.6x the amount of data that of the base-
line in the local experiment and 4.7x in the regional experiment during Synchronization and Setup. The calculation does not include the cost of uploading the long-term SEAL public key, which can be mitigated through public-key infrastructure. During the Query phase of \( d = 1 \) and \( d = 2 \) Synchronized APIR, the client transmits \( \sim 0.2x \) and \( \sim 1.0x \) the amount of data per query that of the baseline, respectively, in both the local and regional experiment. We note that the client’s upload costs during Query appear irregular with respect to the number of sPIR items due to SealPIR’s query compression technique, which is able to pack many query bits in one large ciphertext; this results in big gaps in query size between small and large number of sPIR query bits.

Where Synchronized APIR clearly outperforms the baseline is in computational cost. According to Table 5, the computational costs of the assisting servers (\( S_1 \) and \( S_2 \)) are near negligible across the experiments. The computational costs during Synchronization and Setup are slightly cheaper for the main server (\( S_0 \)) in Synchronized APIR than in the baseline, and slightly more expensive the client in Synchronized APIR than in the baseline. However, the more interesting part is the recurring costs of the main server during Query. Here, the computational cost of the main server in \( d = 1 \) and \( d = 2 \) Synchronized APIR are 78% and 12% that of the baseline, respectively.

We summarize our results below.

1. Baseline \( d = 2 \) SealPIR offers low one-time communication cost but high recurring costs.
2. \( d = 1 \) Synchronized APIR offers 4.6x-4.7x one-time communication cost and \( \sim 0.2x \) recurring communication cost that of the baseline for the client, and 78% recurring computational cost that of the baseline for the main server.
3. \( d = 2 \) Synchronized APIR offers 4.6x-4.7x one-time communication cost and \( \sim 1.0x \) recurring communication cost that of the baseline, and 12% recurring computational cost that of the baseline for the main server.

7.5 Discussion

It is clear that Synchronized APIR provides significant advantage over SealPIR in a long run after the one-time

<table>
<thead>
<tr>
<th>Phases</th>
<th>SealPIR, ( d = 2 )</th>
<th>Sync-APIR, ( d = 1 )</th>
<th>Sync-APIR, ( d = 2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( S_0 )</td>
<td>( S_0 )</td>
<td>( S_0 )</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Download</td>
<td>44K</td>
<td>95K</td>
</tr>
<tr>
<td></td>
<td>Setup</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Upload (spk)</td>
<td>3.5M</td>
<td>3.5M</td>
</tr>
<tr>
<td></td>
<td>Upload (others)</td>
<td>-</td>
<td>769</td>
</tr>
<tr>
<td>Query (per session)</td>
<td>Upload</td>
<td>64K</td>
<td>34K</td>
</tr>
<tr>
<td></td>
<td>Download</td>
<td>257K</td>
<td>32K</td>
</tr>
</tbody>
</table>

Table 4: Client’s communication costs for one query in local and regional PIR experiment (in bytes, \( K = 2^{10} \), \( M = 2^{20} \)). “\( S_1 + S_2 \)” indicates the client’s upload/download costs to/from Server 1 and 2, combined. The upload costs during Setup are listed separately between the cost of SEAL public key and other public parameters.
<table>
<thead>
<tr>
<th>Phases</th>
<th>SealPIR, $d = 2$</th>
<th>Sync-APIR, $d = 1$</th>
<th>Sync-APIR, $d = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$C$</td>
<td>$S_0$</td>
<td>$C$</td>
</tr>
<tr>
<td><strong>Local</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synchronization</td>
<td>&lt; 1</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>Setup</td>
<td>107</td>
<td>150</td>
<td>98</td>
</tr>
<tr>
<td>Query (per session)</td>
<td>8</td>
<td>90</td>
<td>1</td>
</tr>
<tr>
<td><strong>Regional</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synchronization</td>
<td>7</td>
<td>31</td>
<td>184</td>
</tr>
<tr>
<td>Setup</td>
<td>106</td>
<td>653</td>
<td>221</td>
</tr>
<tr>
<td>Query (per session)</td>
<td>8</td>
<td>414</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 5: **Computational costs for one query in local and regional PIR experiment (in milli-seconds).** “$S_1$ or $S_2$” indicates the computational cost on $S_1$ or $S_2$ since they cost the same CPU time.

cost, but what does this means in terms of practicality? For a single non-private NS query to a DNS server, the communication cost would be less than 1 KB, and the computational cost negligible, allowing the operation to scale with minimal computational resources. In comparison, Synchronized APIR would be multiple orders of magnitude more expensive. This is not to mention the specificity of the DNS protocol and complexity of the current DNS infrastructure which would require significant adjustment for Synchronized APIR to fit in.

When putting the scheme in context as such, we reckon it would be more sensible to offer private DNS via Synchronized APIR as an optional, special service one needs to opt in to access (especially because our current conception requires the cache servers to collect cache statistics from non-private DNS service). The computational and communication costs of Synchronized APIR are not necessarily prohibitive because 1) the non-sensitive parts of Synchronization and Setup can be outsourced to a third party and shared across multiple clients, allowing the operation to scale, and 2) the recurring cost of a query is only as expensive as the best sPIR scheme available. OnionPIR [11], for example, can reduce the recurring download cost for the client by 25x in comparison to SealPIR at the same computational cost, although a downside is the recurring upload cost may double with a small database.

8 Conclusion

In this work, we introduce assisted PIR, a generalization to multi-server PIR that allows for database inconsistencies. We present the construction of Synchronized APIR, a hybrid APIR protocol between a black-box single-server PIR scheme and a multi-server PIR scheme which takes advantage of the overlap between inconsistent databases to reduce the costs. A formal analysis of Synchronized APIR is also provided.

We apply Synchronized APIR to demonstrate a proof-of-concept private DNS query application, specifically to query NS records, among DNS cache servers. Then, we evaluate the application with simulated datasets based on realistic assumptions about DNS queries and cache behavior.

The results show that despite the higher initial one-time cost, private DNS query via Synchronized APIR is able to outperform the baseline single-server PIR either in communication cost or in computational cost. Although the costs are high in comparison to non-private DNS, Synchronization APIR holds its potential in future developments of single-server PIR schemes from its black-box use of single-server PIR.

Acknowledgements

We thank Dr. Syed Mahbub Hafiz at University of California, Davis for comments and contributions during the development of this work.

This research was supported in part by the National Science Foundation awards CNS 156537 and the Comcast Innovation Fund.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government, the National Science Foundation, Comcast, Indiana University, nor University of Calgary.
References


A Appendix

A.1 sPIR Correctness and Privacy Definition

Definition A.1 (sPIR Correctness). Following Definition 3.1 scheme $\Pi^{sPIR}$ is correct for any database $V = (v_0, \ldots, v_{m-1})$ such that $m \geq 1$ and $i \in \{0, \ldots, m-1\}$ if

- $(spk, ssk) \leftarrow SGEN(1^\lambda)$
- $sq \leftarrow SQUERY(spk, i, m)$
- $sr \leftarrow SREPLY(spk, V, sq)$
- $sa \leftarrow SDECODE(ssk, sr)$

then $sa = v_i$.

Definition A.2 ($\lambda$-sPIR Privacy). Define an sPIR privacy experiment $\text{PrivS}_{\lambda, \Pi^{sPIR}}(1^\lambda)$ for sPIR scheme $\Pi^{sPIR}$ according to Definition 3.1 and adversary $A$ below.

1. $A$ chooses and outputs $m$.
2. The parameters are generated $(spk, ssk) \leftarrow SGEN(1^\lambda)$ and $spk$ is given to $A$.
3. $A$ is given oracle access to $SQUERY(spk, \cdot)$.
4. $A$ chooses $i_0, i_1 \in \{0, \ldots, m-1\}$ and outputs $(i_0^*, i_1^*)$. A uniformly random bit is chosen $b \leftarrow \{0, 1\}$. A query $sq^* \leftarrow SQUERY(spk, i_b^*)$ is generated and $sq^*$ is given to $A$. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS ’21, page 2292–2306, New York, NY, USA, 2021. Association for Computing Machinery.
5. \( \mathcal{A} \) is given more oracle access to \( \text{SQUERY}(\text{spk}, \cdot) \).

6. \( \mathcal{A} \) outputs \( b^* \in \{0, 1\} \). The output of the experiment is defined to be 1 if \( b^* = b \) and 0 otherwise.

\( \Pi^{\text{PIR}} \) is \( \lambda \)-sPIR privacy-preserving if for all PPT adversary \( \mathcal{A} \), there exists a negligible function \( \text{negl} \) such that

\[
\Pr[\text{Priv}_{\mathcal{A},\Pi^{\text{PIR}}}(1^\lambda) = 1] \leq \frac{1}{2} + \text{negl}(\lambda)
\]