Assisted Private Information Retrieval

Natnatee Dokmai[1], L. Jean Camp[1], and Ryan Henry[2]

[1]Indiana University, Bloomington
[2]Universiy 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 restrictive requirements that are difficult to apply in practical settings. In this work, we introduce *Assisted Private Information Retrieval* (APIR), a new PIR framework for keyword-value databases which generalizes multi-server PIR and relaxes its database consistency assumption. Leveraging the decentralized nature of Domain Name Service (DNS), APIR addresses a privacy issue inherent to recent encrypted DNS proposals such as DNS-over-HTTPS (DoH) by preventing DNS operators from collecting queried domain name data. We propose a construction of *Synchronized APIR*, an efficient APIR scheme as a hybrid 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 can 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 surveillance of users' queries by database-operating servers. To illuminate this concern, let us consider the following scenario: a user wishes to download a sensitive file from a website, but accessing that particular file is incriminating. To get around this problem, 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 patterns.

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 encrypted DNS proposals do not necessarily increase users' privacy. The most significant privacy risk is increased data concentration at DNS operators. Given the exploitation of users' data by data collectors and online services in recent years, it is not far-fetched to claim that the concentration of data is a privacy violation.

The family of cryptographic protocols that proposes to address this threat model is *Private Information Retrieval* (PIR). A core privacy definition of PIR requires that any two PIR queries appear indistinguishable from the attacker's point of view, thus preventing database operators from collecting sensitive information about the queries. Many PIR schemes have been proposed recently, yet they all suffer from similar practical issues. The family of PIR schemes that requires one server to operate, or *single-server* PIR, makes minimal assumptions about data management. Yet, they often rely on additively homomorphic encryption schemes, which are computationally expensive and incur high communication costs in practice. The family of PIR schemes that requires multiple servers to operate, or *multi-server* PIR, are orders of magnitude more efficient than single-server PIR to deploy. Yet, they have conflicting requirements that 1) all the servers hold an identical copy of the database in some form, and 2) some sets of servers do not collude by sharing user queries. In practice, it is difficult to imagine a scenario in which both requirements hold simultaneously. For instance, in the DNS application, it can be argued that DNS servers administered by independent organizations are unlikely to collude. However, requiring that they hold the same cache tables and zone files would be a significant overhaul to the DNS infrastructure, making it less scalable. Further, the decentralization of DNS is what makes it useful for localization and load-balancing to begin with.

1.1 Our contributions

In this work, we introduce a new PIR framework, *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. The servers can operate their copies of the databases that are different from one another, but correctness is defined by the copy operated by the server chosen to be the *main server*.

We present *Synchronized APIR*, an APIR scheme that allows the client to synchronize the "view" of the databases across the servers before 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 protocol [7]. The hybridization takes advantage of the more efficient multi-server PIR scheme to "assist" in lowering the costs of single-server PIR; the level of assistance is determined by the overlap between the main server's database and the rest of the servers' databases. Because Synchronized APIR uses 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 [4] 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 initial one-time cost, private DNS query via Synchronized APIR outperforms SealPIR, the baseline single-server PIR, either in communication cost by a factor of 5 or in computational cost by a factor of 8, assuming 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 operate identical indexed databases, but some database values can go missing in some copies. The scheme first allows the client to synchronize the database "view" to identify the missing values. Then, the client makes a query for an index whose value is not missing in *any* of the databases while hiding which values are missing from the servers. This setting applies to peer-to-peer (P2P) file-sharing, where P2P users share and download parts of the same file or database in small, indexed fragments.

Fanti *et al.*'s scheme can be viewed as a solution to a subset of APIR with two additional limitations: 1) the databases are strictly indexed, and 2) the client is not allowed to query any values that are missing in at least one of the databases. It is worth remarking that the second limitation may introduce a privacy risk in practice. Because sensitive data is often more challenging to access and is the first to go missing, this presents a dilemma to Fanti *et al.*'s scheme, where the more sensitive the data is, the more likely it is inaccessible in the scheme. This results in the clients being turned toward a less privacy-preserving service to be able to access the data. APIR insists that all database values must be 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 (sPIR) is the family of PIR schemes that requires the client to interact with only one database-operating server to query information. sPIR is closely related to *computational* PIR (CPIR) due to the computational assumptions usually required to instantiate it, although not all sPIR schemes are CPIR. One example of such an sPIR scheme is the *trivial* PIR, where the client downloads the entire database from the server and queries from her local copy. This technically satisfies the privacy requirements of PIR because the server cannot learn the client's query. However, this results in a prohibitively high communication cost in practice. Therefore, non-trivial PIR schemes must aim to satisfy the privacy requirements, keeping the communication cost lower than the trivial PIR.

We formally define sPIR in Definition 3.1 below. A typical flow of an sPIR 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 sPIR 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 (sPIR Scheme)**.** Given a server operating an indexed database $V = (v_0, \ldots, v_{m-1})$ for all $i \in \{0, \ldots, m-1\}$. An sPIR scheme is a tuple $\Pi^{\mathsf{sPIR}} = (\textsc{SGen}, \textsc{SQuery}, \textsc{SReply}, \textsc{SDecode})$ defined by

- $\text{SGen}(1^\lambda) \rightarrow (\mathsf{spk}, \mathsf{ssk})$ where λ is the security parameter, spk the public key, and ssk the secret key.
- $\text{SQuery}(\mathsf{spk}, i, m) \rightarrow \mathsf{sq}$ generates query sq from index $i \in \{0, \ldots, m-1\}$ targeting item v_i.
- $\text{SReply}(\mathsf{spk}, V, \mathsf{sq}) \rightarrow \mathsf{sr}$ generates reply sr for query sq from database V.
- $\text{SDecode}(\mathsf{ssk}, \mathsf{sr}) \rightarrow \mathsf{sa}$ decodes reply sr for answer sa.

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 cannot 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, many existing schemes in PIR literature 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 in a simplified sPIR construction as follows. 1) For a given 0-based index i, the query is encoded as the $(i+1)$-th standard basis vector e_{i+1}. 2) The client encrypts e_{i+1} 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 to obtain the answer.

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]. To reduce the computational and communication cost in practice, many techniques have been introduced, 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. 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 due to the techniques mentioned above, 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 (mPIR) is the family of PIR schemes that requires the client to interact with multiple database-operating servers to query information. mPIR is a broader category for *information-theoretic* PIR (IT-PIR) in literature, in that all IT-PIR schemes (except for the trivial PIR) are multi-server, but some mPIR schemes are 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.

We omit the technical abstractions for mPIR here since we intend to use mPIR in a non-block-box manner in our construction. Instead, for instructional purposes, we provide an example of one of the most well-known mPIR schemes, CGKS [7], on which our non-black-box construction is based. Below is a 3-server example of CGKS.

Suppose there are three database-operating servers, $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{S}_3$, each holding an identical, consistent indexed database $V = (a, b, c, d, f) \in \mathsf{GF}(2)^{\ell \cdot 5}$, where each of the 5 database values is of length ℓ. Suppose client $\mathcal{C}$ wishes to retrieve the item at 0-based index 2, i.e. c, then she must generate three queries, one for each server, following the steps below.

1. Sample $q_1 \leftarrow\$ \mathsf{GF}(2)^5$ and $q_2 \leftarrow\$ \mathsf{GF}(2)^5$ uniformly at random. Suppose this results in $q_1 = (0,0,1,1,0)$ and $q_2 = (0,1,0,0,0)$.
2. Compute $q_3' \leftarrow q_1 \oplus q_2$ to obtain $q_3' = (0,1,1,1,0)$.
3. For target index i, encode it as the standard-basis vector e_{i+1}. Here, index 2 is encoded as $e_3 = (0,0,1,0,0)$
4. Compute $q_3 \leftarrow q_3' \oplus e_3$. This results in $q_3 = (0,1,0,1,0)$.

Next, $\mathcal{C}$ sends each query q_j to server $\mathcal{S}_j$, who then generates reply r_j by computing a dot product $r_j \leftarrow q_j \cdot V$ in $\mathsf{GF}(2)^\ell$. As a result, we have $r_1 = c \oplus d$, $r_2 = b$, and $r_3 = b \oplus d$. $\mathcal{S}_j$ returns reply r_j to $\mathcal{C}$. Finally, $\mathcal{C}$ 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 the 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 lop-sided (i.e., if the length of each record is super-logarithmic in the number of records). Otherwise, the query size is asymptotically equivalent to the database size.

Correctness and privacy are defined similarly to sPIR. For correctness, the answer received by the client must match the one she queried. For privacy, the definition includes the notion of *t-collusion threshold* in addition: 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 from a uniformly random string of the same length.

In our following construction, we modify CGKS so that 1) the servers do not require consistent databases, and 2) the participating parties use a PRG with pre-determined seeds to generate all but one query to reduce the communication cost.

4 Assisted PIR

This section aims to provide an abstraction for Assisted PIR (APIR), a new framework for PIR generalizing mPIR, including the definition of privacy and correctness. In the upcoming section, we will show how to construct our APIR scheme called *Synchronized APIR*.

APIR comprises three groups of participating parties with distinct roles, *client* $\mathcal{C}$, *main server* $\mathcal{S}_0$, and *assisting servers* $\mathcal{S}_1 - \mathcal{S}_n$. Client $\mathcal{C}$ wishes to retrieve the value associated with keyword k^* from keyword-value database DB_0, i.e. $\mathsf{DB}_0[k^*]$, operated by main server $\mathcal{S}_0$. This process is assisted by n assisting servers $\mathcal{S}_1, \dots, \mathcal{S}_n$ who independently operate keyword-value databases $\mathsf{DB}_1, \dots, \mathsf{DB}_n$, respectively. The purpose of assistance includes but is not limited to reducing operational costs. $\mathsf{DB}_1, \dots, \mathsf{DB}_n$ do not need to be the exact duplicates of DB_0, although they may have some common keyword-value pairs. For example, it is possible that keyword-value pair $(k, v) \in \mathsf{DB}_0$ but $(k, v) \notin \mathsf{DB}_1$, or $(k, v) \in \mathsf{DB}_1$ and $(k, v') \in \mathsf{DB}_2$ but $v \neq v'$. Note that this database definition is in contrast to that of sPIR and mPIR in Section 3 where databases are assumed to be indexed and consistent. APIR is formally defined below.

Notations

- $\mathcal{C}$, client
- $\mathcal{S}_0$, main server, operating database DB_0
- $\mathcal{S}_1, \dots, \mathcal{S}_n$, n assisting servers, operating database $\mathsf{DB}_1, \dots, \mathsf{DB}_n$, respectively
- DB, a keyword-value database. Let k be a keyword and v a value, then $(k, v) \in \mathsf{DB}$ iff $\mathsf{DB}[k] = v$
- $\mathsf{Keys}(\mathsf{DB})$, the set of all keywords in database DB

Notations

- $\mathsf{ID} := \{0, \dots, n\}$, the set of $n+1$ server ID's
- $\mathsf{AID} := \{1, \dots, n\}$, the set of n assisting server ID's
- $(a_i)_I := (a_{i_1}, \dots, a_{i_n})$ where $i_j \in I$, a sequence ordered by the index set I. We sometimes use $(a_i)_{i \in I}$ instead for clarity

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

- $\textsc{ServGen}(\mathsf{id}, \mathsf{DB}_{\mathsf{id}}) \to \mathsf{par}_{\mathsf{id}}$, where $\mathsf{id} \in \mathsf{ID}$ and $\mathsf{par}_{\mathsf{id}}$ is the database parameter for $\mathsf{DB}_{\mathsf{id}}$.
- $\textsc{CliGen}\left(1^\lambda, t, (\mathsf{par}_{\mathsf{id}})_{\mathsf{ID}}\right) \to (\mathsf{pk}, \mathsf{sk})$, where λ is the security parameter, t the collusion threshold where $1 \leq t \leq n$, pk is the public key, and sk the secret key.
- $\textsc{Query}(\mathsf{pk}, \mathsf{sk}, k) \to (\mathsf{q}_{\mathsf{id}})_{\mathsf{ID}}$ generates query q_{id} for server id from query keyword $k \in \mathsf{Keys}(\mathsf{DB}_0)$ targeting database value $\mathsf{DB}_0[k]$.
- $\textsc{Reply}(\mathsf{id}, \mathsf{pk}, \mathsf{DB}_{\mathsf{id}}, \mathsf{q}_{\mathsf{id}}) \to \mathsf{r}_{\mathsf{id}}$ generates reply r_{id} for query q_{id} from database $\mathsf{DB}_{\mathsf{id}}$.
- $\textsc{Decode}(\mathsf{sk}, (\mathsf{r}_{\mathsf{id}})_{\mathsf{ID}}) \to \mathsf{a}$ decodes replies $(\mathsf{r}_{\mathsf{id}})_{\mathsf{ID}}$ for answer a.

The flow of the scheme is similar to that of sPIR and mPIR described in Section 3: 1) The servers generate database parameters using $\textsc{ServGen}$ and send them to the client. 2) The client generates a public-secret key pair from the database and security parameters using $\textsc{CliGen}$. 3) The client chooses a query keyword from DB_0 and generates queries using $\textsc{Query}$; each server gets their own query. 4) The servers respond to the query with their database and public key using $\textsc{Reply}$. And finally, 5) the client aggregates all the replies and decodes them using the secret key with $\textsc{Decode}$ to obtain the answer.

The correctness of APIR is defined according to the content of DB_0. If the client queries with keyword k, then $\mathsf{DB}_0[k]$ is defined to be the correct answer (even if there may be other $\mathsf{DB}_{\mathsf{id}}$ such that $\mathsf{DB}_{\mathsf{id}}[k] \neq \mathsf{DB}_0[k]$); formally, correctness is defined below.

Definition 4.2 (APIR Correctness)**.** Following Definition 4.1, scheme Π^{APIR} is APIR-correct for any set ID, databases $\mathsf{DB}_{\mathsf{id}}$ for all $\mathsf{id} \in \mathsf{ID}$, and keyword $k \in \mathsf{Keys}(\mathsf{DB}_0)$ if

- $\forall \mathsf{id} \in \mathsf{ID} : \mathsf{par}_{\mathsf{id}} \leftarrow \textsc{ServGen}(\mathsf{id}, \mathsf{DB}_{\mathsf{id}})$
- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \textsc{CliGen}\left(1^\lambda, t, (\mathsf{par}_{\mathsf{id}})_{\mathsf{ID}}\right)$

- $(\mathsf{q}_{\mathsf{id}})_{\mathsf{ID}} \leftarrow \textsc{Query}(\mathsf{pk}, \mathsf{sk}, k)$
- $\forall \mathsf{id} \in \mathsf{ID} : \mathsf{r}_{\mathsf{id}} \leftarrow \textsc{Reply}(\mathsf{id}, \mathsf{pk}, \mathsf{DB}_{\mathsf{id}}, \mathsf{q}_{\mathsf{id}})$
- $\mathsf{a} \leftarrow \textsc{Decode}(\mathsf{sk}, (\mathsf{r}_{\mathsf{id}})_{\mathsf{ID}})$

then $\mathsf{a} = \mathsf{DB}_0[k]$.

The privacy definition of APIR is similar to that of mPIR in that we want to capture the indistinguishability notion of queries when server collusion does not exceed threshold t. Intuitively, an APIR scheme is privacy-preserving if an adversary, who compromises a set of at most t servers, cannot distinguish between two queries generated from any query keywords k_0 and k_1 of the adversary's own choice. Moreover, we want to model a real-world scenario in which the client is not informed of which servers are compromised or whether they are compromised at all. Since the notion of servers is not a part of the privacy definition, we assume the hypothetical *oracle*, $\mathcal{O}$, who works to relay the compromised queries to the adversary without the client's knowledge. We formally define this using a game-based definition below.

Definition 4.3 ((λ, t)-APIR Privacy)**.** Given security parameter λ, collusion threshold t, and adversary $\mathcal{A}$, define an APIR privacy experiment $\mathsf{PrivA}_{\mathcal{A},\Pi^{\mathsf{APIR}},t}(1^\lambda)$ for an APIR scheme Π^{APIR} according to Definition 4.1 below.

1. $\mathcal{A}$ chooses correctly formatted database parameters $(\mathsf{par}_{\mathsf{id}})_{\mathsf{ID}}$ and outputs $(\mathsf{par}_{\mathsf{id}})_{\mathsf{ID}}$. $\mathcal{A}$ chooses a collusion set $C \subset \mathsf{ID}$ such that $|C| \leq t$ and sends C to oracle $\mathcal{O}$.
2. The public-secret key pair is generated by $(\mathsf{pk}, \mathsf{sk}) \leftarrow \textsc{CliGen}(1^\lambda, t, (\mathsf{par}_{\mathsf{id}})_{\mathsf{ID}})$ and pk is given to $\mathcal{A}$.
3. $\mathcal{A}$ is given oracle access to $\textsc{Query}$ in the following way: $\mathcal{A}$ chooses and outputs $k \in \mathcal{K}(\mathsf{par}_0)$ where $\mathcal{K}(\mathsf{par}_0)$ is the query keyword space determined by a correctly formatted par_0. Queries are generated by $(\mathsf{q}_{\mathsf{id}})_{\mathsf{ID}} \leftarrow \textsc{Query}(\mathsf{pk}, \mathsf{sk}, k)$ and $(\mathsf{q}_{\mathsf{id}})_{\mathsf{ID}}$ is given to $\mathcal{O}$. $\mathcal{O}$ gives $(\mathsf{q}_{\mathsf{id}})_C$ to $\mathcal{A}$.
4. $\mathcal{A}$ chooses $k_0^*, k_1^* \in \mathcal{K}(\mathsf{par}_0)$, and outputs (k_0^*, k_1^*). A uniformly random bit is sampled $b \leftarrow\$ \{0,1\}$. Queries are generated by $(\mathsf{q}^*_{\mathsf{id}})_{\mathsf{ID}} \leftarrow \textsc{Query}(\mathsf{pk}, \mathsf{sk}, k_b^*)$ and $(\mathsf{q}^*_{\mathsf{id}})_{\mathsf{ID}}$ is given to $\mathcal{O}$. $\mathcal{O}$ gives $(\mathsf{q}^*_{\mathsf{id}})_C$ to $\mathcal{A}$.
5. $\mathcal{A}$ is given more oracle access to $\textsc{Query}$ according to step 3.
6. $\mathcal{A}$ outputs $b^* \in \{0,1\}$. The experiment's output is 1 if $b^* = b$ and 0 otherwise.

Π^{APIR} is (λ, t)-APIR private for all PPT adversary $\mathcal{A}$ if there exists a negligible function negl such that

$$\Pr\left[\mathsf{PrivA}_{\mathcal{A},\Pi^{\mathsf{APIR}},t}(1^\lambda) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$$

5 Our Scheme: Synchronized APIR

This section presents our construction of APIR called *Synchronized APIR*. Synchronized APIR allows the client to synchronize the global "view" of the databases before making queries. This results in a one-time communication cost to initialize the scheme and low recurring computational and communication costs to query. Synchronized APIR uses a black-box sPIR scheme described in Section 3.1 in its hybrid PIR construction, meaning that improvements to sPIR schemes can result in improvements to Synchronized APIR.

For the rest of this section, we introduce a high-level concept of Synchronized APIR in Section 5.1, followed by the complete description in Section 5.2. Finally, we describe the implementation of 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 a PIR server independently operates each database. 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 an adversary controlling some of 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 immediately achievable. Instead, the client must resort to the costly sPIR on DB_0 and completely disregard $\mathsf{DB}_1, \mathsf{DB}_2$, and DB_3.

To circumvent this issue, Synchronized APIR uses some of the keyword-value pairs in $\mathsf{DB}_1, \mathsf{DB}_2, \mathsf{DB}_3$ that are consistent with DB_0 to "assist" in reducing the cost of sPIR on DB_0 via a hybrid PIR. We will walk through the example in Figure 1 below to demonstrate this concept.

Step ①: For this demonstration, we assume that the client can fully observe $\mathsf{DB}_0, \ldots, \mathsf{DB}_3$ (how exactly this is done will be explained in the upcoming section). The client first notices that there are keyword-value pairs in $\mathsf{DB}_1, \mathsf{DB}_2, \mathsf{DB}_3$ 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 $\mathsf{DB}_1, \mathsf{DB}_2, \mathsf{DB}_3$ are disregarded.

Step ②: To lower the cost, 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 similar to the one in CGKS in Section 3.2. Let us suppose that the client decides that $t = 2$; that is, no more than two servers can collude. Recall from Section 3.2 that a given threshold t requires at least

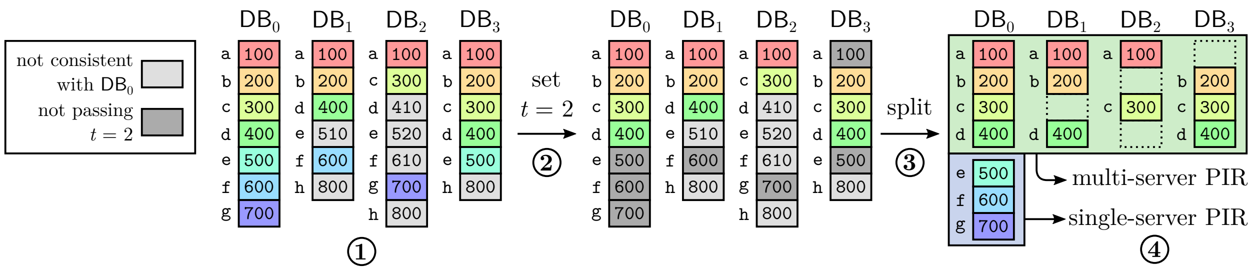

Figure 1: **Concept of Synchronized APIR.** A conceptual demonstration of how Synchronized APIR performs hybrid PIR on inconsistent databases when the collusion threshold is $t = 2$.

$t + 1$ 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 keyword-value and the mPIR duplicates are scattered across multiple databases?
- How can the communication cost of query and reply be optimized?

5.2 Protocol Description

In order to query keyword k, client $\mathcal{C}$ works with main server $\mathcal{S}_0$ and assisting servers $\mathcal{S}_1, \ldots, \mathcal{S}_n$ 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 $\mathcal{C}$ wishes to query.

In the following sections, we will describe step ① - ⑨ of Synchronized APIR as shown in Figure 2 in detail.

5.2.1 Synchronization Phase

During the Synchronization phase, $\mathcal{C}$ synchronizes the global view of the databases to pick out the consistent keyword-value pairs to be retrieved via mPIR and the rest via sPIR. This is done through *catalogs* which $\mathcal{C}$ downloads from all the servers. A catalog is a map from keywords to hashes of values in a database. The purpose of a catalog is to uniquely represent a database while keeping the communication cost low.

We refer to step ① - ③ in Figure 2 and ① - ② in 3 as a demonstration as we describe Synchronization phase below.

Notations

- $\mathsf{Map}\{(k, v) \in K \times V \mid \Phi(k, v)\}$, a keyword-value map builder notation where K is the key domain, V value domain, and Φ a predicate.
- ϕ, an empty set

Input

▷ $\mathsf{DB}_{\mathsf{id}}$	**from**	$\mathcal{S}_{\mathsf{id}}, \forall \mathsf{id} \in \mathsf{ID}$
▷ t	**from**	$\mathcal{C}$

Output

▷ $\mathsf{Cat}_{\mathsf{id}}, M_{\mathsf{id}}$	**to**	$\mathcal{S}_{\mathsf{id}}, \forall \mathsf{id} \in \mathsf{ID}$
▷ $\mathsf{Cat}_0, (\mathsf{Cat}'_{\mathsf{id}})_{\mathsf{AID}}, (M_{\mathsf{id}})_{\mathsf{ID}}, S$	**to**	$\mathcal{C}$

Synchronization Protocol

① $\mathcal{C}$ obtains the full catalog Cat_0 from $\mathcal{S}_0$ and catalog intersections $\mathsf{Cat}'_1, \ldots, \mathsf{Cat}'_n$ from $\mathcal{S}_1, \ldots, \mathcal{S}_n$ by following the steps below:

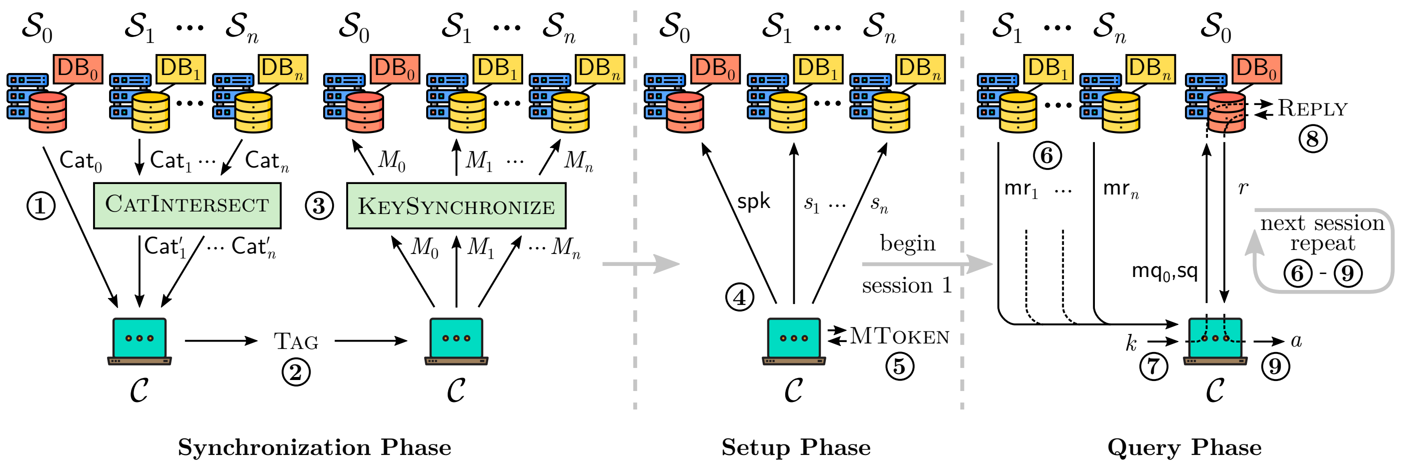

Figure 2: **Overview of Synchronized APIR.** Synchronized APIR comprises three distinct phases between client $\mathcal{C}$, main server $\mathcal{S}_0$, and assisting Servers $\mathcal{S}_1 - \mathcal{S}_n$: Synchronization, Setup, and Query. Synchronization and Setup are completed once at the start, while a new Query session can be repeated with every new query keyword k.

A. For each $\mathsf{id} \in \mathsf{ID}$, $\mathcal{S}_{\mathsf{id}}$ generates catalog

$$\mathsf{Cat}_{\mathsf{id}} \leftarrow \mathsf{Map}\left\{(k,h) \mid (k,v) \in \mathsf{DB}_{\mathsf{id}}, h = \mathsf{H}(v)\right\}$$

where $\mathsf{H}(\cdot)$ is a universal hash function with short hash values, chosen randomly by $\mathcal{C}$ from the universal family. (A shorter hash length will reduce the communication cost but increase the collision probability.) This results in catalogs that are smaller than the databases in size but capable of representing the uniqueness of database values. In Figure 3, we choose a toy hash function $\mathsf{H}(v) = v/10$ to demonstrate the point that an appropriate hash function should be able to produce short unique hash values. In general, when the distribution of database values is unknown, universal hash functions should be applied to reduce hash collisions.

B. $\mathcal{C}$ downloads Cat_0 from $\mathcal{S}_0$. For each $\mathsf{id} \in \mathsf{AID}$, $\mathcal{C}$ engages in the Catalog Intersection protocol CatIntersection with $\mathcal{S}_{\mathsf{id}}$, where $\mathcal{C}$ obtains catalog intersections $\mathsf{Cat}'_{\mathsf{id}} = \mathsf{Cat}_{\mathsf{id}} \cap \mathsf{Cat}_0$ 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 DB_0 in $\mathsf{DB}_1, \ldots, \mathsf{DB}_n$ and with Cat_0 in $\mathsf{Cat}_1, \ldots, \mathsf{Cat}_n$ are eliminated via catalog intersection. With the full catalog and catalog intersections, $\mathcal{C}$ now has complete information of the keyword-value pairs consistent with DB_0.

② $\mathcal{C}$ tags the catalogs to categorize keyword-value pairs for either mPIR or sPIR:

$$((M_{\mathsf{id}})_{\mathsf{ID}}, S) \leftarrow \textsc{Tag}\left(\mathsf{Cat}_0, \left(\mathsf{Cat}'_{\mathsf{id}}\right)_{\mathsf{AID}}, t\right)$$

where t is the collusion threshold, i.e., the highest number of colluding servers $\mathcal{C}$ can tolerate without leaking query keywords; M_{id} is the mPIR keyword set, i.e., the set of keywords determined to be queried via mPIR from $\mathcal{S}_{\mathsf{id}}$; S is the sPIR keyword set, i.e., the set of keywords determined to be queried via sPIR from $\mathcal{S}_0$. Tag is detailed in Algorithm 1 below. In Figure 3, $\mathcal{C}$ has decided that $t = 2$, so the pairs that pass the threshold for mPIR must have $t + 1 = 3$ duplicates: 1 duplicate in Cat_0 and 2 duplicates in $\mathsf{Cat}'_1, \mathsf{Cat}'_2, \mathsf{Cat}'_3$. `(a, 10)`, `(b, 20)`, `(c, 30)`, and `(d, 40)` all pass the threshold (`(a, 10)` in Cat'_3 is redundant and disregarded). Neither of `(e, 50)`, `(f, 60)`, or `(g, 70)` pass the threshold, so $\mathcal{C}$ 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.

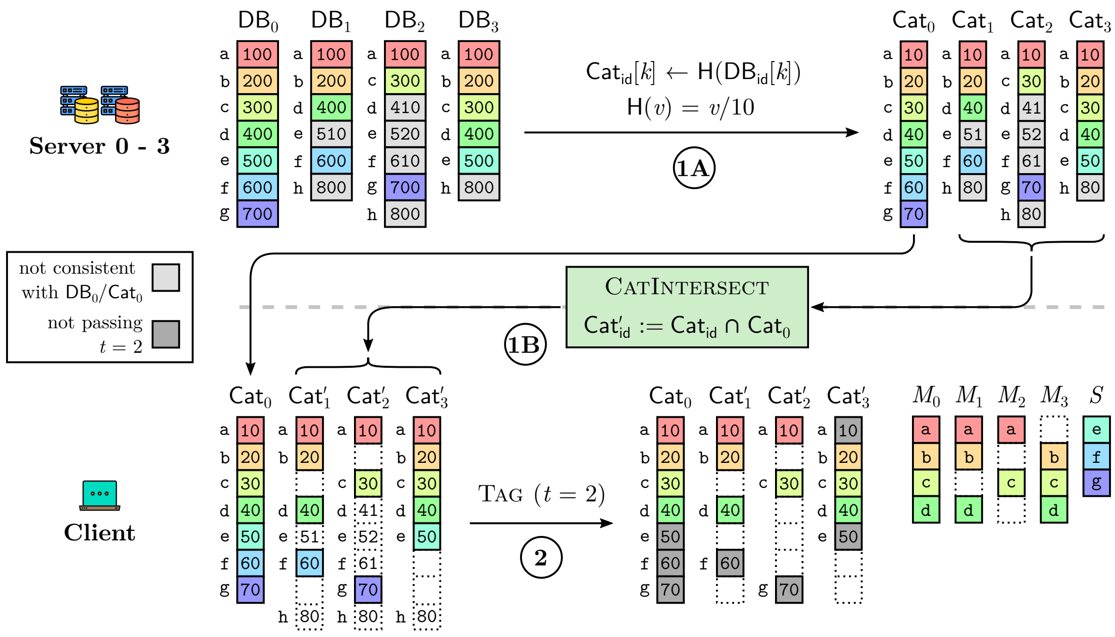

Figure 3: **Synchronization.** An example of databases in a Synchronized APIR setting with three assisting servers that undergo the Synchronization phase.

Algorithm 1 Catalog Tagging. In line 5 - 8, if there are at least t duplicates of a given keyword, then t of them are tagged for mPIR and stored in M_{id} ($t+1$ 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 randomly or with a specific optimization strategy.

```
1:  procedure TAG($\mathsf{Cat}_0, (\mathsf{Cat}'_{\mathsf{id}})_{\mathsf{AID}}, t$)
2:      $\forall \mathsf{id} \in \mathsf{ID} : M_{\mathsf{id}} \leftarrow \phi$
3:      for $k \in \mathsf{Keys}(\mathsf{Cat}_0)$ do
4:          $G \leftarrow \{\mathsf{id} \in \mathsf{AID} \mid k \in \mathsf{Keys}(\mathsf{Cat}'_{\mathsf{id}})\}$
5:          if $|G| \geq t$ then
6:              choose $C_k \subseteq G$ such that $|C_k| = t$
7:              $\forall \mathsf{id} \in C_k \cup \{0\} : M_{\mathsf{id}} \leftarrow M_{\mathsf{id}} \cup \{k\}$
8:          end if
9:      end for
10:     $S \leftarrow \mathsf{Keys}(\mathsf{Cat}_0) \setminus M_0$
11:     return $((M_{\mathsf{id}})_{\mathsf{ID}}, S)$
12: end procedure
```

③ For each $\mathsf{id} \in \mathsf{ID}$, $\mathcal{C}$ and $\mathcal{S}_{\mathsf{id}}$ engage in the Keyword Synchronization protocol KeySynchronize, where $\mathcal{S}_{\mathsf{id}}$ obtains M_{id} at the end. The goal of KeySynchronize is to transmit M_{id} at a cost lower than sending them in full. KeySynchronize is described in Section 5.2.6.

5.2.2 Setup Phase

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

Notations

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

During Setup, $\mathcal{C}$ sends each of $\mathcal{S}_1, \ldots, \mathcal{S}_n$ a random PRG seed which will be used to generate random mPIR queries for all Query sessions $\mathsf{sid} \in [l]$. We follow step ④ and ⑤ in Figure 2 and ⑤ in 4 to describe the Setup phase below.

Input

▷ $\lvert M_{\mathsf{id}} \rvert$	**from**	$\mathcal{S}_{\mathsf{id}}, \forall \mathsf{id} \in \mathsf{AID}$
▷ $l, (M_{\mathsf{id}})_{\mathsf{AID}}$	**from**	$\mathcal{C}$

Output

▷ spk	**to**	$\mathcal{S}_0$
▷ $\left(\mathsf{mq}^{\mathsf{sid}}_{\mathsf{id}}\right)_{\mathsf{sid} \in [l]}$	**to**	$\mathcal{S}_{\mathsf{id}}, \forall \mathsf{id} \in \mathsf{AID}$
▷ $\mathsf{spk}, \mathsf{ssk}, \left(\mathsf{tok}^{\mathsf{sid}}\right)_{[l]}$	**to**	$\mathcal{C}$

Setup Protocol

④ $\mathcal{C}$ generates an sPIR key pair $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \textsc{SGen}(\lambda)$ and sends spk to main server $\mathcal{S}_0$.

For each assisting servers $\mathsf{id} \in \mathsf{AID}$, $\mathcal{C}$ samples a PRG seed $s_{\mathsf{id}} \in \{0,1\}^{\lambda}$ uniformly at random and sends s_{id} to $\mathcal{S}_{\mathsf{id}}$. $\mathcal{C}$ and each $\mathcal{S}_{\mathsf{id}}$ define

$$\left(\mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}\right)_{\mathsf{sid}\in[l]} := \mathsf{PRG}(s_{\mathsf{id}}, l \cdot |M_{\mathsf{id}}|)$$

where $l \cdot |M_{\mathsf{id}}|$ is the total length of the output string and $\left|\mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}\right| = |M_{\mathsf{id}}|$.

In practice, when the PRG is implemented with a stream cipher, l does not need to be predetermined, and each $\mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}$ can be generated on the fly.

⑤ $\mathcal{C}$ defines query *tokens*

$$\mathsf{tok}^{\mathsf{sid}} := \textsc{MToken}\left((M_{\mathsf{id}})_{\mathsf{ID}}, \left(\mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}\right)_{\mathsf{AID}}\right)$$

for all $\mathsf{sid} \in [l]$. MToken is defined in Algorithm 2 below. Intuitively, a token is an XOR of the random queries $\mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}$ for all $\mathsf{id} \in \mathsf{AID}$ when the bits are aligned to the ordering of the corresponding keywords in M_0. This is visually demonstrated in Figure 4 step ⑤.

Like in the previous step, each $\mathsf{tok}^{\mathsf{sid}}$ can be generated on the fly.

Algorithm 2 mPIR Token Generation.

```
1: procedure MToken($(M_{id})_{ID}, (mq_{id})_{AID}$)
2:     $tok \leftarrow \{0\}^{|M_0|}$
3:     for $k \in M_0$ do
4:         $C_k \leftarrow \{id \in AID : k \in M_{id}\}$
5:         $\forall id \in C_k \cup \{0\} : i_{id} \leftarrow index(k, M_{id})$
6:         $tok[i_0] \leftarrow \bigoplus_{id \in C} mq_{id}[i_{id}]$
7:     end for
8:     return tok
9: end procedure
```

5.2.3 Special Case: mPIR-Only Query Phase

Before describing the full protocol of the 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., $S = \phi$ in Algorithm 1. The goal is to show how all the mPIR components fit together during the Query phase to provide correct answers without dealing with the hybrid PIR's complexity. We shall follow the steps in Figure 4 for this walk-through.

Suppose that at this point $\mathcal{C}$ and $\mathcal{S}_0, \ldots, \mathcal{S}_n$ 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.

Input

▷ $\mathsf{DB}_{\mathsf{id}}, M_{\mathsf{id}}$	**from**	$\mathcal{S}_{\mathsf{id}}, \forall \mathsf{id} \in \mathsf{ID}$
▷ $\mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}$	**from**	$\mathcal{S}_{\mathsf{id}}, \forall \mathsf{id} \in \mathsf{AID}$
▷ $M_0, \mathsf{tok}^{\mathsf{sid}}, k^{\mathsf{sid}}$	**from**	$\mathcal{C}$

Output

▷ a^{sid}	**to**	$\mathcal{C}$

mPIR-Only Query Protocol for Session sid

⑥ For each assisting servers $\mathsf{id} \in \mathsf{AID}$:

A. $\mathcal{S}_{\mathsf{id}}$ generates a reply from the query

$$\mathsf{mr}_{\mathsf{id}}^{\mathsf{sid}} \leftarrow \textsc{MReply}\left(M_{\mathsf{id}}, \mathsf{DB}_{\mathsf{id}}, \mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}\right)$$

and sends $\mathsf{mr}_{\mathsf{id}}^{\mathsf{sid}}$ to $\mathcal{C}$. MReply is described in Algorithm 3 below. Figure 4 step ⑥ demonstrates the mPIR reply generation process: the values of $\mathsf{DB}_{\mathsf{id}}$ are filtered with M_{id} by key, and the dot product in $\mathsf{GF}(2)$ between the filtered values and random query bits produces the reply. This process is identical for $\mathcal{S}_2$ and $\mathcal{S}_3$.

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

```
1: procedure MReply($M_{id}, DB_{id}, mq_{id}$)
2:     $V \leftarrow (DB_{id}[k])_{k \in M_{id}}$
3:     $mr_{id} \leftarrow mq_{id} \cdot V$
4:     return $mr_{id}$
5: end procedure
```

Note that this step can be completed offline as it does not require a query keyword k^{sid} by $\mathcal{C}$. This means that during the online time (that is, step ⑦* onwards), $\mathcal{C}$ only needs to interact with $\mathcal{S}_0$, significantly reducing the latency.

⑦* Once $\mathcal{C}$ has decided on a mPIR query keyword $k^{\mathsf{sid}} \in M_0$ to query, she generates a mPIR query for $\mathcal{S}_0$

$$\mathsf{mq}_0^{\mathsf{sid}} \leftarrow \textsc{MQuery}\left(M_0, \mathsf{tok}^{\mathsf{sid}}, k^{\mathsf{sid}}\right)$$

where MQuery is described in Algorithm 4 below. Intuitively, as visually demonstrated in Figure 4, MQuery flips one bit of the token $\mathsf{tok}^{\mathsf{sid}}$ at the index position of k^{sid} in M_0 to produce the query $\mathsf{mq}_0^{\mathsf{sid}}$.

Finally, $\mathcal{C}$ sends $\mathsf{mq}_0^{\mathsf{sid}}$ to $\mathcal{S}_0$ to query.

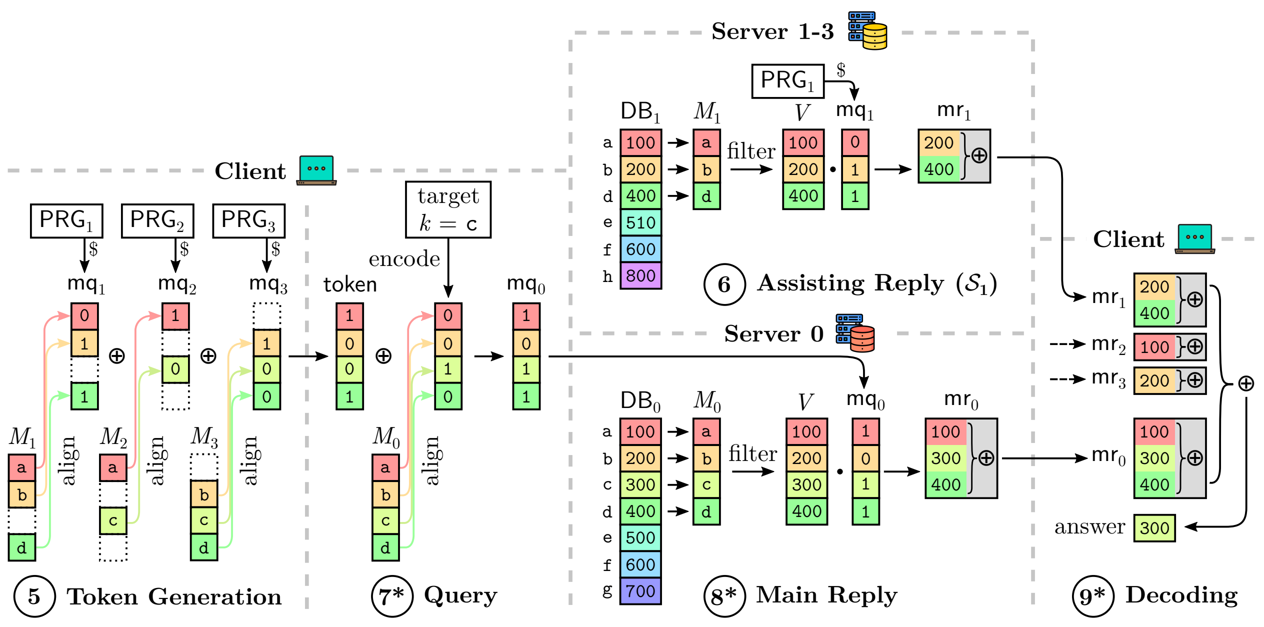

Figure 4: **mPIR-Only Query Phase Workflow.** The workflow of mPIR components during the Query phase to demonstrate how mPIR keyword-value pairs can be queried with the client's query keyword.

Algorithm 4 mPIR Query Generation.

```
1: procedure MQuery($M_0$, tok, $k$)
2:     $\mathsf{mq}_0 \leftarrow \mathsf{tok}$
3:     if $k \in M_0$ then
4:         $i \leftarrow \mathsf{index}(k, M_0)$
5:         $\mathsf{mq}_0[i] \leftarrow \mathsf{mq}_0[i] \oplus 1$
6:     end if
7:     return $\mathsf{mq}_0$
8: end procedure
```

⑧* Upon receiving $\mathsf{mq}_0^{\mathsf{sid}}$, $\mathcal{S}_0$ processes the query in the same way queries are processed in step ⑥, except that here $\mathsf{mq}_0^{\mathsf{sid}}$ is given by $\mathcal{C}$,

$$\mathsf{mr}_0^{\mathsf{sid}} \leftarrow \text{MREPLY}\left(\mathsf{mq}_0^{\mathsf{sid}}, M_0, \mathsf{DB}_0\right)$$

$\mathcal{S}_0$ returns $\mathsf{mr}_0^{\mathsf{sid}}$ to $\mathcal{C}$.

⑨* Now that $\mathcal{C}$ has received all $\mathsf{mr}_0^{\mathsf{sid}}, \ldots, \mathsf{mr}_n^{\mathsf{sid}}$, she can decode them for the answer by a simple XOR

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

where $a^{\mathsf{sid}} = \mathsf{DB}[k^{\mathsf{sid}}]$ as a result.

To see why it is the case that $a^{\mathsf{sid}} = \mathsf{DB}[k^{\mathsf{sid}}]$, recall from step ⑤ and ⑦* in Figure 4. If we "peel off" $\mathsf{mq}_1^{\mathsf{sid}}, \ldots, \mathsf{mq}_n^{\mathsf{sid}}$ from $\mathsf{mq}_0^{\mathsf{sid}}$ via XOR, the result is exactly the encoding of k^{sid} by design. Because all keyword-value pairs of $\mathsf{DB}_1, \ldots, \mathsf{DB}_n$ 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^{sid} is intact if no more than t Servers collude. This is because up to t Servers can observe up to t of $\mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}$'s which are pseudorandom as intended by the collusion threshold during 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

▷ $\mathsf{DB}_0, M_0, \mathsf{spk}$	**from**	$\mathcal{S}_0$
▷ $\mathsf{DB}_{\mathsf{id}}, M_{\mathsf{id}}, \mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}$	**from**	$\mathcal{S}_{\mathsf{id}}, \forall \mathsf{id} \in \mathsf{AID}$
▷ $M_0, S, \mathsf{spk}, \mathsf{ssk}, \mathsf{tok}^{\mathsf{sid}}, k^{\mathsf{sid}}$	**from**	$\mathcal{C}$

Output

▷ a^{sid}	**to**	$\mathcal{C}$

Query Protocol for Session sid

⑥ This is the same as step ⑥ in Section 5.2.3.

⑦ $\mathcal{C}$ decides on a query keyword $k^{\mathsf{sid}} \in M_0 \cup S$ and generates a hybrid-PIR query for $\mathcal{S}_0$

$$\left(\mathsf{mq}_0^{\mathsf{sid}}, \mathsf{sq}^{\mathsf{sid}}\right) \leftarrow \text{QUERY}\left(M_0, S, \mathsf{spk}, \mathsf{tok}^{\mathsf{sid}}, k^{\mathsf{sid}}\right)$$

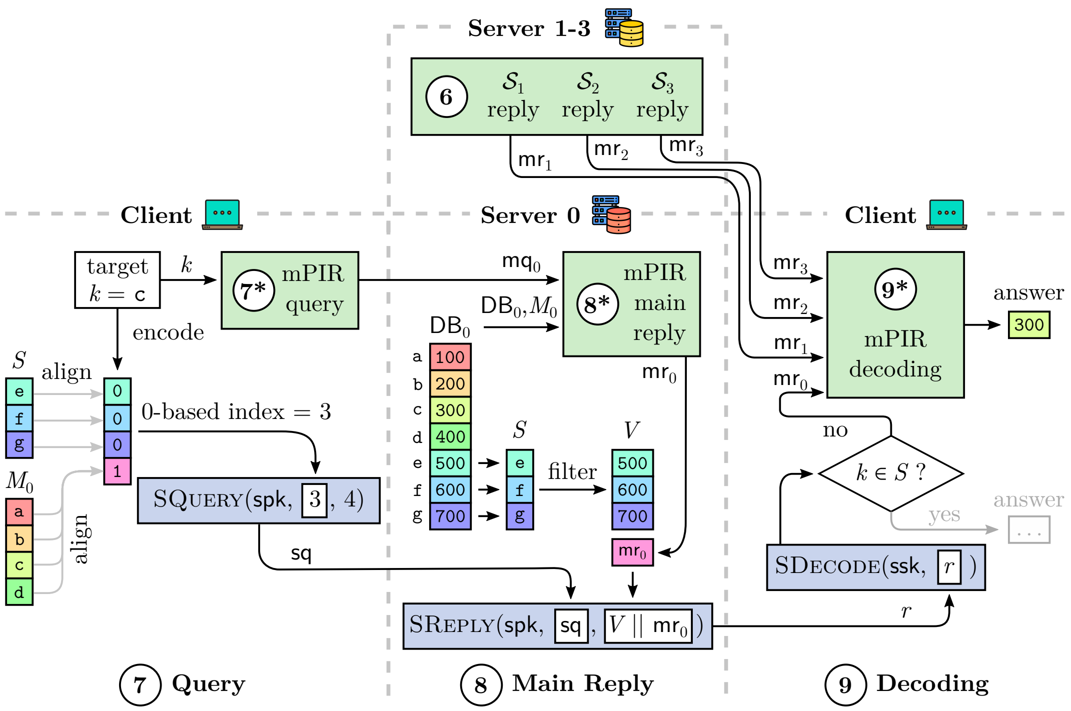

Figure 5: **Query Phase Workflow.** The workflow of hybrid components during the Query phase to demonstrate how keyword-value pairs in DB_0 can be queried with the client's query key.

where $\mathsf{sq}^{\mathsf{sid}}$ is the sPIR query and $\mathsf{mq}_0^{\mathsf{sid}}$ mPIR query. QUERY is described in Algorithm 5 below. $\mathcal{C}$ sends $(\mathsf{mq}_0^{\mathsf{sid}}, \mathsf{sq}^{\mathsf{sid}})$ to $\mathcal{S}_0$.

The hybridization of the mPIR-sPIR query is visually demonstrated in Figure 5. First, step ⑦* is followed to generate $\mathsf{mq}_0^{\mathsf{sid}}$. (Note that if $k^{\mathsf{sid}} \notin M_0$, then $\mathsf{mq}_0^{\mathsf{sid}}$ is a "blank" token as per Algorithm 4.) Next, if $k^{\mathsf{sid}} \in S$, then k^{sid} is encoded by its index position in S as an input to SQUERY; otherwise if $k^{\mathsf{sid}} \in M_0$, then it is encoded as the last index position as an input to SQUERY. In this example, because $k^{\mathsf{sid}} = \mathsf{c} \in M_0$, 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 QUERY($M_0, S, \mathsf{spk}, \mathsf{tok}, k$)
2:     $\mathsf{mq}_0 \leftarrow$ MQUERY ($M_0, \mathsf{tok}, k$)
3:     $\mathsf{sq} \leftarrow \perp$
4:     if $S \neq \phi$ then
5:         if $k \in S$ then
6:             $i \leftarrow \mathsf{index}(k, S)$
7:             $\mathsf{sq} \leftarrow$ SQUERY($\mathsf{spk}, i, |S|+1$)
8:         else
9:             $\mathsf{sq} \leftarrow$ SQUERY($\mathsf{spk}, |S|, |S|+1$)
10:        end if
11:    end if
12:    return $(\mathsf{mq}_0, \mathsf{sq})$
13: end procedure
```

⑧ $\mathcal{S}_0$ receives the query $(\mathsf{mq}_0^{\mathsf{sid}}, \mathsf{sq}^{\mathsf{sid}})$ from $\mathcal{C}$ and generates the reply

$$r^{\mathsf{sid}} \leftarrow \text{REPLY}(M_0, \mathsf{DB}_0, \mathsf{mq}_0^{\mathsf{sid}}, \mathsf{sq}^{\mathsf{sid}})$$

where r^{sid} is the resulting hybrid PIR reply. REPLY is described in Algorithm 6 below. $\mathcal{S}_0$ returns r^{sid} to $\mathcal{C}$.

Algorithm 6 Reply Generation.

```
1: procedure $\textsc{Reply}(M_0, \mathsf{DB}_0, \mathsf{mq}_0, \mathsf{sq})$
2:     $\mathsf{mr}_0 \leftarrow \textsc{MReply}(M_0, \mathsf{DB}_0, \mathsf{mq}_0)$
3:     $S \leftarrow \mathsf{Keys}\,(\mathsf{DB}_0) \setminus M_0$
4:     $r \leftarrow \perp$
5:     if $S \neq \phi$ then
6:         $V \leftarrow (\mathsf{DB}_0[k])_{k \in S}$
7:         $r \leftarrow \textsc{SReply}(\mathsf{spk}, V \| \mathsf{mr}_0, \mathsf{sq})$
8:     else
9:         $r \leftarrow \mathsf{mr}_0$
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 $\mathsf{mr}_0^{\mathsf{sid}}$. (Note that if $\mathsf{mq}_0^{\mathsf{sid}}$ is not a blank query, then $\mathsf{mr}_0^{\mathsf{sid}}$ is a genuine mPIR reply; otherwise, it is a random value. This fact, however, is unknown to $\mathcal{S}_0$.) Next, the values of DB_0 are filtered with S by key, resulting in V. $\mathsf{mr}_0^{\mathsf{sid}}$ is appended to V as the last item, i.e. $V \| \mathsf{mr}_0^{\mathsf{sid}}$. The sPIR query $\mathsf{sq}^{\mathsf{sid}}$ and $V \| \mathsf{mr}_0^{\mathsf{sid}}$ are input to the sPIR reply generation algorithm $\textsc{SReply}$ 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 $k^{\mathsf{sid}} \in S$, $\mathsf{sq}_{\mathsf{sid}}$ is generated to target values in V, ignoring $\mathsf{mr}_0^{\mathsf{sid}}$ which is the last item. However, when $k \in M_0$, $\mathsf{sq}_{\mathsf{sid}}$ is generated to target the last value, which is where $\mathsf{mr}_0^{\mathsf{sid}}$ is located. The resulting reply r^{sid} therefore encodes either a targeted value in V when $k^{\mathsf{sid}} \in S$ or $\mathsf{mr}_0^{\mathsf{sid}}$ when $k^{\mathsf{sid}} \in M_0$.

⑨ $\mathcal{C}$ receives the reply r^{sid} from $\mathcal{S}_0$ and decodes it using the same query keyword k^{sid} from step ⑦ to obtain the answer

$$a^{\mathsf{sid}} \leftarrow \textsc{Decode}\left(S, \mathsf{ssk}, k^{\mathsf{sid}}, \left(\mathsf{mr}_{\mathsf{id}}^{\mathsf{sid}}\right)_{\mathsf{id} \in \mathsf{AID}}, r^{\mathsf{sid}}\right)$$

$\textsc{Decode}$ is described in Algorithm 7 below.

Figure 5 demonstrates how all the replies from $\mathcal{S}_0, \ldots, \mathcal{S}_3$ come together for $\mathcal{C}$ to decode the answer. First, the reply r^{sid} from $\mathcal{S}_0$ is decoded using secret key ssk in the sPIR decoding algorithm $\textsc{SDecode}$. If $k^{\mathsf{sid}} \in S$, then $\mathcal{C}$ already has the final answer; otherwise, $\mathcal{C}$ obtains $\mathsf{mr}_0^{\mathsf{sid}}$ and follows step ⑨*. Here, $\mathcal{C}$ XOR all $\mathsf{mr}_0^{\mathsf{sid}}, \ldots, \mathsf{mr}_3^{\mathsf{sid}}$ to obtain the answer.

For privacy, it is important that regardless of whether $k^{\mathsf{sid}} \in S$, $\mathsf{mr}_1^{\mathsf{sid}}, \ldots, \mathsf{mr}_n^{\mathsf{sid}}$ must be downloaded by $\mathcal{C}$ from $\mathcal{S}_1, \ldots, \mathcal{S}_n$, respectively. If this step is skipped, then $\mathcal{S}_1, \ldots, \mathcal{S}_n$ can infer that $k^{\mathsf{sid}} \in S$.

Algorithm 7 Decoding.

```
1: procedure $\textsc{Decode}(S, \mathsf{ssk}, k, (\mathsf{mr}_{\mathsf{id}}^{\mathsf{sid}})_{\mathsf{id} \in \mathsf{AID}}, r)$
2:     if $k \in S$ then
3:         $a \leftarrow \textsc{SDecode}(\mathsf{ssk}, r)$
4:     else
5:         if $S \neq \phi$ then
6:             $\mathsf{mr}_0 \leftarrow \textsc{SDecode}(\mathsf{ssk}, r)$
7:         else
8:             $\mathsf{mr}_0 \leftarrow r$
9:         end if
10:        $a \leftarrow \bigoplus_{\mathsf{id} \in \mathsf{ID}} \mathsf{mr}_{\mathsf{id}}$
11:    end if
12:    return $a$
13: end procedure
```

5.2.5 Catalog Intersection

Catalog Intersection protocol is a part of the Synchronization phase to reduce the communication cost of transferring catalog intersections. The idea is as follows: given that $\mathcal{C}$ has already obtained Cat_0, then $\mathsf{Cat}'_{\mathsf{id}} = \mathsf{Cat}_{\mathsf{id}} \cap \mathsf{Cat}_0$ can be compressed with a hash function (we call these hashes *digests*) for transporting. We describe the Catalog Intersection protocol below.

Input

▷ $\mathsf{Cat}_{\mathsf{id}}$	**from**	$\mathcal{S}_{\mathsf{id}}, \forall \mathsf{id} \in \mathsf{AID}$
▷ Cat_0	**from**	$\mathcal{C}$

Output

▷ $(\mathsf{Cat}'_{\mathsf{id}})_{\mathsf{AID}}, (\mathsf{Dig}_{\mathsf{id}})_{\mathsf{AID}}$	**to**	$\mathcal{C}$

Catalog Intersection Protocol

① For each $\mathsf{id} \in \mathsf{AID}$, $\mathcal{S}_{\mathsf{id}}$ generates digests from $\mathsf{Cat}_{\mathsf{id}}$

$$\mathsf{Dig}_{\mathsf{id}} \leftarrow \{\mathsf{G}((k,h)) \mid (k,h) \in \mathsf{Cat}_{\mathsf{id}}\}$$

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

$\mathcal{S}_{\mathsf{id}}$ sends the digest set $\mathsf{Dig}_{\mathsf{id}}$ to $\mathcal{C}$.

② $\mathcal{C}$ downloads all the digest sets $(\mathsf{Dig}_{\mathsf{id}})_{\mathsf{AID}}$ from $\mathcal{S}_1, \ldots, \mathcal{S}_n$. For each $\mathsf{id} \in \mathsf{AID}$, $\mathcal{C}$ remaps digest sets into catalog intersections as follows:

$$\mathsf{Cat}'_{\mathsf{id}} \leftarrow \mathsf{Map}\{(k,h) \in \mathsf{Cat}_0 \mid \mathsf{G}((k,h)) \in \mathsf{Dig}_{\mathsf{id}}\}$$

5.2.6 Keyword Synchronization

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

Input

▷ $\mathsf{Dig}_{\mathsf{id}}, \mathsf{Cat}_{\mathsf{id}}$	**from**	$\mathcal{S}_{\mathsf{id}}, \forall \mathsf{id} \in \mathsf{ID}$
▷ $(M_{\mathsf{id}})_{\mathsf{ID}}, (\mathsf{Dig}_{\mathsf{id}})_{\mathsf{ID}}$	**from**	$\mathcal{C}$

Output

▷ M_{id}	**to**	$\mathcal{S}_{\mathsf{id}}, \forall \mathsf{id} \in \mathsf{ID}$

Keyword Synchronization Protocol

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

$$I_0 \leftarrow \textsc{KeyToIndex}_0\left(M_0, \mathsf{Cat}_0\right)$$

$$\forall \mathsf{id} \in \mathsf{AID} : I_{\mathsf{id}} \leftarrow \textsc{KeyToIndex}_{\mathsf{id}}\left(M_{\mathsf{id}}, \mathsf{Dig}_{\mathsf{id}}, \mathsf{Cat}_{\mathsf{id}}\right)$$

KeyToIndex is described in Algorithm 8 below. $\mathcal{C}$ then sends I_{id} to $\mathcal{S}_{\mathsf{id}}$. I_{id} is a small, indexed representation of M_{id} which can be transported at a low communication cost.

Algorithm 8 Key-to-Index Mapping

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

② For each $\mathsf{id} \in \mathsf{ID}$, $\mathcal{S}_{\mathsf{id}}$ receives I_{id} and remaps it back to M_{id}

$$M_0 \leftarrow \textsc{IndexToKey}_0\left(I_0, \mathsf{Cat}_0\right)$$

$$\forall \mathsf{id} \in \mathsf{AID} : M_{\mathsf{id}} \leftarrow \textsc{IndexToKey}_{\mathsf{id}}\left(I_{\mathsf{id}}, \mathsf{Dig}_{\mathsf{id}}, \mathsf{Cat}_{\mathsf{id}}\right)$$

IndexToKey is described in Algorithm 9 below.

Algorithm 9 Index-to-Key Mapping

```
1: procedure $\textsc{IndexToKey}_0(I_0, \mathsf{Cat}_0)$
2:     $K \leftarrow \mathsf{Keys}(\mathsf{Cat}_0)$
3:     $M_0 \leftarrow \{k \in K \mid i \in I_0, \mathsf{index}(k, K) = i\}$
4:     return $M_0$
5: end procedure
6: procedure $\textsc{IndexToKey}_{\mathsf{id}}(I_{\mathsf{id}}, \mathsf{Dig}_{\mathsf{id}}, \mathsf{Cat}_{\mathsf{id}})$
7:     $D \leftarrow \{d \in \mathsf{Dig}_{\mathsf{id}} \mid i \in I_{\mathsf{id}}, \mathsf{index}(d, \mathsf{Dig}_{\mathsf{id}}) = i\}$
8:     $M_{\mathsf{id}} \leftarrow \{k \mid (k,h) \in \mathsf{Cat}_{\mathsf{id}}, \mathsf{G}((k,h)) \in D\}$
9:     return $M_{\mathsf{id}}$
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 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 solves this problem more efficiently by killing two birds with one stone: instead of applying 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 $\mathsf{Dig}_{\mathsf{id}}$ 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 unclear how this cost-saving technique can be applied to existing set reconciliation schemes in the literature.

5.2.8 Keyword Compression

When query keywords in the databases are long, they can result in a high communication cost during the Synchronization phase since all the keywords in the catalogs must be sent to the client. For this reason, we aim to reduce this cost by representing each query keyword k by its hash $\tilde{\mathsf{H}}(k)$, where $\tilde{\mathsf{H}}(\cdot)$ is a universal hash function with short hash values, chosen randomly 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 a high collision rate (i.e., a high false-positive rate). Our suggested workaround for this problem is to keep the hash values short (and therefore a high collision rate) but let the servers modify the databases by prepending the keyword to the value so that a false positive can be detected at the end. That is,

$$\widetilde{\mathsf{DB}}_{\mathsf{id}} := \mathsf{Map}\left\{(k, \tilde{v}) \mid (k, v) \in \mathsf{DB}_{\mathsf{id}}, \tilde{v} := k \| v\right\}$$

When the client receives the answer $\tilde{v}$ at the end, she can check whether the obtained keyword from the answer is the same as the query keyword. Indeed, this comes at the cost of the servers processing extra loads and longer database values.

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 $\tilde{\mathsf{H}}$ 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 [5], 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 library version 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

This section provides 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 complete proofs and 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 produce 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 Π^{sPIR} and that the universal hash functions H, G, and $\tilde{\mathsf{H}}$ produce no collisions. Following Definition 4.2, the Synchronized APIR scheme Π^{SynAPIR} is correct for any server set $\mathsf{ID} = \{0, \ldots, n\}$ and $\mathsf{AID} = \{1, \ldots, n\}$, collusion threshold $1 \leq t \leq n$, databases $\mathsf{DB}_{\mathsf{id}}$ for all $\mathsf{id} \in \mathsf{ID}$, l Query sessions, and $k^{\mathsf{sid}} \in \mathsf{Keys}(\mathsf{DB}_0)$ for all $\mathsf{sid} \in [l]$. That is, if

- Synchronization protocol takes inputs $(\mathsf{DB}_{\mathsf{id}})_{\mathsf{ID}}, t$ and outputs $(\mathsf{Cat}_{\mathsf{id}})_{\mathsf{ID}}, \left(\mathsf{Cat}'_{\mathsf{id}}\right)_{\mathsf{AID}}, (M_{\mathsf{id}})_{\mathsf{ID}}, S$
- Setup protocol takes inputs $l, (M_{\mathsf{id}})_{\mathsf{AID}}$ and outputs $\mathsf{spk}, \mathsf{ssk}, \left(\mathsf{mq}^{\mathsf{sid}}_{\mathsf{id}}\right)_{\mathsf{sid} \in [l], \mathsf{id} \in \mathsf{AID}}, \left(\mathsf{tok}^{\mathsf{sid}}\right)_{[l]}$
- For all $\mathsf{sid} \in [l]$, Query protocol for session sid takes inputs $(\mathsf{DB}_{\mathsf{id}})_{\mathsf{ID}}, (M_{\mathsf{id}})_{\mathsf{ID}}, S, \left(\mathsf{mq}^{\mathsf{sid}}_{\mathsf{id}}\right)_{\mathsf{id} \in \mathsf{AID}}, \mathsf{spk}, \mathsf{ssk}, \mathsf{tok}^{\mathsf{sid}}, k^{\mathsf{sid}}$ and outputs a^{sid}

then for all $\mathsf{sid} \in [l], \mathsf{DB}_0[k^{\mathsf{sid}}] = a^{\mathsf{sid}}$.

Proof sketch. First, by assuming that the universal hash functions produce no collusions, we can claim that

- $\tilde{\mathsf{H}}(k)$ perfectly represents keyword k, so k can be efficiently replaced with $\tilde{\mathsf{H}}(k)$.
- $\mathsf{H}(v)$ perfectly represents value v, so catalog $\mathsf{Cat}_{\mathsf{id}}$ perfectly represents database $\mathsf{DB}_{\mathsf{id}}$. Therefore tagging the catalogs has the same effect of tagging the databases directly.
- $\mathsf{G}((k, h))$ perfectly represents (k, h), so the Catalog Intersection protocol and the Keyword Synchronization protocol are trivially correct.

Next, we consider the properties of the tags $(M_{\mathsf{id}})_{\mathsf{ID}}, S$. By construction of Tag (Algorithm 1), for each keyword $k \in \mathsf{Keys}(\mathsf{DB}_0)$,

1. either $k \in M_0$ or $k \in S$ but not both
2. if $k \in M_0$ and $k \in M_{\mathsf{id}}$ for any $\mathsf{id} \in \mathsf{AID}$, then $\mathsf{DB}_0[k] = \mathsf{DB}_{\mathsf{id}}[k]$
3. $\bigcup_{k \in M_0} \bigcup_{\mathsf{id} \in C_k} \{(\mathsf{id}, k)\} = \bigcup_{\mathsf{id} \in \mathsf{AID}} \bigcup_{k \in M_{\mathsf{id}}} \{(\mathsf{id}, k)\}$

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

If $k^{\mathsf{sid}} \in S$, then by construction of Query (Algorithm 5), $\mathsf{sq}^{\mathsf{sid}}$ targets item $\mathsf{index}(k^{\mathsf{sid}}, S)$ in the filtered database $(\mathsf{DB}_0[k])_{k \in S}$ in Reply (Algorithm 6), which is exactly $\mathsf{DB}_0[k^{\mathsf{sid}}]$. By sPIR correctness of Π^{sPIR}, we conclude that $a^{\mathsf{sid}} = \mathsf{DB}_0[k^{\mathsf{sid}}]$.

If $k^{\mathsf{sid}} \in M_0$, we consider how the mPIR token $\mathsf{tok}^{\mathsf{sid}}$ and queries $\mathsf{mq}^{\mathsf{sid}}_{\mathsf{id}}$ are generated. Consider each bit of

$\mathsf{tok}^{\mathsf{sid}}$ by construction in MToken (Algorithm 2): there exists $k \in M_0$ such that

$$\mathsf{tok}^{\mathsf{sid}}[\mathsf{index}(k, M_0)] = \bigoplus_{\mathsf{id} \in C_k} \mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}[\mathsf{index}(k, M_{\mathsf{id}})]$$

where C_k is a set of id's such that $\mathsf{DB}_0[k] = \mathsf{DB}_{\mathsf{id}}[k]$ by property 2. stated above. This implies

$$\begin{aligned}\mathsf{tok}^{\mathsf{sid}}&[\mathsf{index}(k, M_0)] \cdot \mathsf{DB}_0[k] \\ &= \bigoplus_{\mathsf{id} \in C_k} \mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}[\mathsf{index}(k, M_{\mathsf{id}})] \cdot \mathsf{DB}_0[k] \\ &= \bigoplus_{\mathsf{id} \in C_k} \mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}[\mathsf{index}(k, M_{\mathsf{id}})] \cdot \mathsf{DB}_{\mathsf{id}}[k]\end{aligned}$$

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

$$\begin{aligned}\mathsf{tok}^{\mathsf{sid}}&\cdot(\mathsf{DB}_0[k])_{k \in M_0} \\ &= \bigoplus_{k \in M_0} \mathsf{tok}^{\mathsf{sid}}[\mathsf{index}(k, M_0)] \cdot \mathsf{DB}_0[k] \\ &= \bigoplus_{k \in M_0} \bigoplus_{\mathsf{id} \in C_k} \mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}[\mathsf{index}(k, M_{\mathsf{id}})] \cdot \mathsf{DB}_{\mathsf{id}}[k] \\ &= \bigoplus_{\mathsf{id} \in \mathsf{AID}} \bigoplus_{k \in M_{\mathsf{id}}} \mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}[\mathsf{index}(k, M_{\mathsf{id}})] \cdot \mathsf{DB}_{\mathsf{id}}[k] \\ &= \bigoplus_{\mathsf{id} \in \mathsf{AID}} \mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}} \cdot (\mathsf{DB}_{\mathsf{id}}[k])_{k \in M_{\mathsf{id}}}\end{aligned}$$

Next, because

$$\mathsf{mq}_0^{\mathsf{sid}}[\mathsf{index}(k^{\mathsf{sid}}, M_0)] = \mathsf{tok}^{\mathsf{sid}}[\mathsf{index}(k^{\mathsf{sid}}, M_0)] \oplus 1$$

by construction of MQuery (Algorithm 4), we have

$$\begin{aligned}&\bigoplus_{\mathsf{id} \in \mathsf{ID}} \mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}} \cdot (\mathsf{DB}_{\mathsf{id}}[k])_{k \in M_{\mathsf{id}}} \\ &\quad= \bigoplus_{\mathsf{id} \in \mathsf{ID}} \mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}} \cdot (\mathsf{DB}_{\mathsf{id}}[k])_{k \in M_{\mathsf{id}}} \\ &\quad= \mathsf{tok}^{\mathsf{sid}} \cdot (\mathsf{DB}_0[k])_{k \in M_0} \oplus (1 \cdot \mathsf{DB}_0[k^{\mathsf{sid}}]) \\ &\qquad\quad \bigoplus_{\mathsf{id} \in \mathsf{AID}} \mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}} \cdot (\mathsf{DB}_{\mathsf{id}}[k])_{k \in M_{\mathsf{id}}} \\ &\quad= \mathsf{DB}_0[k^{\mathsf{sid}}]\end{aligned}$$

Thus proving the correctness of the mPIR-only Query protocol.

To show that the general case in the Query protocol is correct, we observe that when $k^{\mathsf{sid}} \in M_0$, $\mathsf{sq}^{\mathsf{sid}}$ targets item $|S|$ by the construction of Query (Algorithm 5). That is, $\mathsf{sq}^{\mathsf{sid}}$ targets targets $\mathsf{mr}_0^{\mathsf{sid}}$ by construction of Reply (Algorithm 6). By sPIR correctness of Π^{sPIR}, we conclude that in Decode (Algorithm 7), SDecode$(\mathsf{ssk}, r^{\mathsf{sid}}) = \mathsf{mr}_0^{\mathsf{sid}}$. And finally,

$$\begin{aligned}a^{\mathsf{sid}} &= \bigoplus_{\mathsf{id} \in \mathsf{ID}} \mathsf{mr}_{\mathsf{id}}^{\mathsf{sid}} \\ &= \bigoplus_{\mathsf{id} \in \mathsf{ID}} \mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}} \cdot (\mathsf{DB}_{\mathsf{id}}[k])_{k \in M_{\mathsf{id}}} \\ &= \mathsf{DB}_0[k^{\mathsf{sid}}]\end{aligned}$$

by the correctness of the mPIR-only Query protocol. □

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

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

- $E_1(U) :=$
$$\forall k, k' \in \mathsf{Keys}(U) : k \neq k' \implies \tilde{\mathsf{H}}(k) \neq \tilde{\mathsf{H}}(k')$$
- $E_2(U) :=$
$$\forall v, v' \in \mathsf{Values}(U) : v \neq v' \implies \mathsf{H}(v) \neq \mathsf{H}(v')$$
- $E_3(U) :=$
$$\begin{aligned}&\forall (k, v), (k', v') \in U : \\ &\quad \mathsf{G}((\tilde{\mathsf{H}}(k), \mathsf{H}(v))) \neq \mathsf{G}((\tilde{\mathsf{H}}(k'), \mathsf{H}(v'))) \implies \\ &\qquad\qquad (\tilde{\mathsf{H}}(k), \mathsf{H}(v)) \neq (\tilde{\mathsf{H}}(k'), \mathsf{H}(v'))\end{aligned}$$

Define $p(m, d) := e^{-\frac{m(m-1)}{2d}}$, then

$$\begin{aligned}&\Pr[E_1(U), E_2(U), E_3(U)] \\ &\quad \approx p(|U|, 2^{l_{\mathsf{G}}}) \cdot p(|\mathsf{Keys}(U)|, 2^{l_{\tilde{\mathsf{H}}}}) \cdot p(|\mathsf{Values}(U)|, 2^{l_{\mathsf{H}}})\end{aligned}$$

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 $2^{l_{\tilde{\mathsf{H}}}}$ and the number of individuals is $|\mathsf{Keys}(U)|$; likewise for $E_2(U)$, the number of birthdays is $2^{l_{\mathsf{H}}}$ and the number of individuals is $|\mathsf{Values}(U)|$. By [12], we have

$$\Pr[E_1(U)] \approx p(|\mathsf{Keys}(U)|, 2^{l_{\tilde{\mathsf{H}}}})$$

$$\Pr[E_2(U)] \approx p(|\mathsf{Values}(U)|, 2^{l_{\mathsf{H}}})$$

Given that $E_1(U)$ and $E_2(U)$ have occurred, $E_3(U)$ also describes the same birthday problem with $2^{l_{\mathsf{G}}}$ birthdays and $|U|$ individuals. Therefore,

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

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

$$\begin{aligned}
&\Pr[E_1(U), E_2(U), E_3(U)] \\
&\quad = \Pr[E_3(U) \mid E_1(U), E_2(U)] \cdot \Pr[E_1(U), E_2(U)] \\
&\quad = \Pr[E_3(U) \mid E_1(U), E_2(U)] \cdot \Pr[E_1(U)] \Pr[E_2(U)] \\
&\quad \approx p(|U|, 2^{l_{\mathsf{G}}}) \cdot p(|\mathsf{Keys}(U)|, 2^{l_{\tilde{\mathsf{H}}}}) \cdot p(|\mathsf{Values}(U)|, 2^{l_{\mathsf{H}}})
\end{aligned}$$

□

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

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

So,

$$\begin{aligned}
&\Pr[\Pi^{\mathsf{SynAPIR}} \text{ is correct}] \\
&\quad \geq \Pr[\Pi^{\mathsf{SynAPIR}} \text{ is correct} \mid E_1(U), E_2(U), E_3(U)] \\
&\qquad \cdot \Pr[E_1(U), E_2(U), E_3(U)] \\
&\quad = \Pr[E_1(U), E_2(U), E_3(U)]
\end{aligned}$$

And therefore

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

We remark 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. After all, certain hash collisions do not affect the correctness of the scheme. For example, in the catalogs, it is not an issue if $v \neq v'$ but $\mathsf{H}(v) = \mathsf{H}(v')$ as long as pairs $(k, \mathsf{H}(v))$ and $(k', \mathsf{H}(v'))$ are in the catalogs and $k \neq k'$. In future work, we will provide analysis with a tighter bound considering all 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 4.3. Then, we show that mPIR queries are pseudorandom given the variable-length PRG, even if some 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 cannot 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 ((λ, t)-Synchronized APIR Privacy)**.** Following Definition 4.3, we define the privacy experiment $\mathsf{PrivA}_{\mathcal{A},\Pi^{\mathsf{SynAPIR}},t}(1^\lambda)$ for the Synchronized APIR scheme given a λ-sPIR privacy-preserving scheme Π^{sPIR} by Definition A.2 and variable-length PRG $\mathsf{PRG}(s, \ell) \to r$ below.

1. $\mathcal{A}$ chooses the number of Query sessions $l = \text{poly}(\lambda)$, well-formed catalog Cat_0 and digests $(\mathsf{Dig}_{\mathsf{id}})_{\mathsf{AID}}$, and outputs $(l, \mathsf{Cat}_0, (\mathsf{Dig}_{\mathsf{id}})_{\mathsf{AID}})$. $\mathcal{A}$ chooses a collusion set $C \subset \mathsf{ID}$ such that $|C| \leq t$ and sends C to the oracle $\mathcal{O}$.

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 $\mathsf{Cat}_0, (\mathsf{Dig}_{\mathsf{id}})_{\mathsf{AID}}$ to produce $\left(\mathsf{Cat}'_{\mathsf{id}}\right)_{\mathsf{AID}}$. Catalogs are tagged by

   $$((M_{\mathsf{id}})_{\mathsf{ID}}, S) \leftarrow \textsc{Tag}(\mathsf{Cat}_0, \left(\mathsf{Cat}'_{\mathsf{id}}\right)_{\mathsf{AID}}, t)$$

   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 $(M_{\mathsf{id}})_{\mathsf{ID}}$ and l and outputs $\left(\mathsf{spk}, \mathsf{ssk}, \left(\mathsf{tok}^{\mathsf{sid}}\right)_{[l]}\right)$ (the mq output is ignored). The PRG seeds $(s_{\mathsf{id}})_{\mathsf{AID}}$ generated during this step is also saved.

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

3. Initialize session ID $\mathsf{sid} = 1$. $\mathcal{A}$ is given oracle access to $\textsc{Query}$ in the following way:

   (a) At $\mathsf{sid} = 1$, $(s_{\mathsf{id}})_{\mathsf{AID}}$ is given to $\mathcal{O}$, and $\mathcal{O}$ gives $(s_{\mathsf{id}})_{C\setminus\{0\}}$ to $\mathcal{A}$ (recall that s_{id} is a PRG seed).

   (b) $\mathcal{A}$ chooses and outputs $k^{\mathsf{sid}} \in \mathsf{Keys}(\mathsf{Cat}_0)$.

   (c) A query is generated by

   $$\left(\mathsf{mq}_0^{\mathsf{sid}}, \mathsf{sq}^{\mathsf{sid}}\right) \leftarrow \textsc{Query}\left(M_0, S, \mathsf{spk}, \mathsf{tok}^{\mathsf{sid}}, k^{\mathsf{sid}}\right)$$

   following step ⑦ of Query Protocol in Section 5.2.4. $\left(\mathsf{mq}_0^{\mathsf{sid}}, \mathsf{sq}^{\mathsf{sid}}\right)$ is given to $\mathcal{O}$.

   (d) If $0 \in C$, $\mathcal{O}$ gives $\left(\mathsf{mq}_0^{\mathsf{sid}}, \mathsf{sq}^{\mathsf{sid}}\right)$ to $\mathcal{A}$; otherwise, $\mathcal{O}$ gives an empty value $\perp$ to $\mathcal{A}$.

   (e) $\mathsf{sid} \leftarrow \mathsf{sid} + 1$

4. During some session $\mathsf{sid} = \mathsf{sid}^* \le l$,

   (a) $\mathcal{A}$ chooses $k_0^{\mathsf{sid}^*}, k_1^{\mathsf{sid}^*} \in \mathsf{Keys}(\mathsf{Cat}_0)$ and outputs $\left(k_0^{\mathsf{sid}^*}, k_1^{\mathsf{sid}^*}\right)$.

   (b) A uniformly random bit is sampled $b \leftarrow_\$ \{0,1\}$.

   (c) A query is generated

$$\left(\mathsf{mq}_{0,b}^{\mathsf{sid}^*}, \mathsf{sq}_b^{\mathsf{sid}^*}\right) \leftarrow \textsc{Query}\left(M_0, S, \mathsf{spk}, \mathsf{tok}^{\mathsf{sid}^*}, k_b^{\mathsf{sid}^*}\right)$$

   and $\left(\mathsf{mq}_{0,b}^{\mathsf{sid}^*}, \mathsf{sq}_b^{\mathsf{sid}^*}\right)$ is given to $\mathcal{O}$.

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

   (e) $\mathsf{sid} \leftarrow \mathsf{sid}^* + 1$

5. $\mathcal{A}$ is given more oracle access to Query until $\mathsf{sid} = l$.

6. $\mathcal{A}$ outputs $b^* \in \{0,1\}$. The experiment's output is 1 if $b^* = b$ and 0 otherwise.

Π^{SynAPIR} is (λ, t)-APIR privacy-preserving for all PPT adversary $\mathcal{A}$ if there exists a negligible function negl such that

$$\Pr\left[\mathsf{PrivA}_{\mathcal{A},\Pi^{\mathsf{SynAPIR}},t}(1^\lambda) = 1\right] \le \frac{1}{2} + \mathsf{negl}(\lambda)$$

To show that Synchronized APIR satisfies this definition, we first prove that mPIR queries are pseudorandom, even if some of the PRG seeds are predetermined. (The predetermined seeds are indicated by set A below.) mPIR query pseudorandomness implies that two queries generated with different keywords are computationally indistinguishable from one another.

Theorem 3 (mPIR Query Pseudorandomness)**.** Suppose a variable-length PRG $\mathsf{PRG}(s, \ell) \to r$ where $|s| = \lambda$ and $|r| = \ell = \mathrm{poly}(\lambda)$. For any $\mathsf{ID} := \{0, \dots, n\}$, $\mathsf{AID} := \{1, \dots, n\}$, $1 \le t \le n$, well-formed catalog Cat_0 and catalog intersections $\mathsf{Cat}'_{\mathsf{id}}$, $\mathsf{id} \in \mathsf{AID}$, l Query sessions, $\mathsf{sid} \in [l]$, $k^{\mathsf{sid}} \in \mathsf{Keys}(\mathsf{Cat}_0)$, $A \subset \mathsf{AID}$ such that $|A| < t$, and $(s_{\mathsf{id}})_A \in \{0,1\}^{\lambda \cdot |A|}$; define

- $\forall \mathsf{id} \in \mathsf{AID} \setminus A : s_{\mathsf{id}} \leftarrow_\$ \{0,1\}^\lambda$
- $((M_{\mathsf{id}})_{\mathsf{AID}}, S) = \textsc{Tag}(\mathsf{Cat}_0, (\mathsf{Cat}'_{\mathsf{id}})_{\mathsf{id}} \in \mathsf{AID})$
- $\forall \mathsf{id} \in \mathsf{AID} : \left(\mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}\right)_{\mathsf{sid} \in [l]} = \mathsf{PRG}(s_{\mathsf{id}}, l \cdot |M_{\mathsf{id}}|)$
- $\forall \mathsf{sid} \in [l] : \mathsf{tok}^{\mathsf{sid}} = \textsc{MToken}\left((M_{\mathsf{id}})_{\mathsf{ID}}, \left(\mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}\right)_{\mathsf{AID}}\right)$
- $\forall \mathsf{sid} \in [l] : \mathsf{mq}_0^{\mathsf{sid}} = \textsc{MQuery}\left(M_0, \mathsf{tok}^{\mathsf{sid}}, k^{\mathsf{sid}}\right)$
- $\mathsf{aux} = \left((M_{\mathsf{id}})_{\mathsf{ID}}, A, (s_{\mathsf{id}})_A, \left(k^{\mathsf{sid}}\right)_{[l]}\right)$

Then for all PPT distinguisher $\mathcal{D}$, there exists a negligible function negl such that

$$\left|\Pr\left[\mathcal{D}(r, \mathsf{aux}) = 1\right] - \Pr\left[\mathcal{D}\left(\left(\mathsf{mq}_0^{\mathsf{sid}}\right)_{[l]}, \mathsf{aux}\right) = 1\right]\right| \le \mathsf{negl}(\lambda)$$

where $r \leftarrow_\$ \{0,1\}^{l \cdot |M_0|}$ is sampled uniformly at random.

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

Part 1: For any $B \subset \mathsf{AID}$ such that $A \subseteq B$ and $|B| = t-1$, we will slightly modify the definition of $\mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}$ above called $\tilde{\mathsf{mq}}_{\mathsf{id}}^{\mathsf{sid}}$ where if $\mathsf{id} \in \mathsf{AID} \setminus B$, then for all $\mathsf{sid} \in [l]$, $\tilde{\mathsf{mq}}_{\mathsf{id}}^{\mathsf{sid}} \leftarrow_\$ \{0,1\}^{|M_{\mathsf{id}}|}$ is sampled uniformly at random; everything else remains unchanged i.e. $\forall \mathsf{id} \in B, \forall \mathsf{sid} \in [l] : \tilde{\mathsf{mq}}_{\mathsf{id}}^{\mathsf{sid}} = \mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}$. This results in the new $\tilde{\mathsf{mq}}_0^{\mathsf{sid}}, \mathsf{sid} \in [l]$.

We claim that these two ensembles are perfectly indistinguishable

$$\langle r, \mathsf{aux}\rangle \overset{p}{\equiv} \langle (\tilde{\mathsf{mq}}_0^{\mathsf{sid}})_{[l]}, \mathsf{aux}\rangle$$

by observing the construction of $\mathsf{mq}_0^{\mathsf{sid}}$. Let

$$\tilde{\mathsf{tok}}^{\mathsf{sid}} = \textsc{MToken}\left((M_{\mathsf{id}})_{\mathsf{ID}}, \left(\tilde{\mathsf{mq}}_{\mathsf{id}}^{\mathsf{sid}}\right)_{\mathsf{id} \in \mathsf{AID}}\right)$$

and consider each bit of $\tilde{\mathsf{tok}}^{\mathsf{sid}}$: for each $k \in M_0$,

$$\begin{aligned}\tilde{\mathsf{tok}}^{\mathsf{sid}}[\mathsf{index}(k, M_0)] &= \bigoplus_{\mathsf{id} \in C_k} \tilde{\mathsf{mq}}_{\mathsf{id}}^{\mathsf{sid}}[\mathsf{index}(k, M_{\mathsf{id}})] \\ &= \bigoplus_{\mathsf{id} \in C_k \setminus B} \tilde{\mathsf{mq}}_{\mathsf{id}}^{\mathsf{sid}}[\mathsf{index}(k, M_{\mathsf{id}})] \\ &\quad \oplus \bigoplus_{\mathsf{id} \in C_k \cap B} \tilde{\mathsf{mq}}_{\mathsf{id}}^{\mathsf{sid}}[\mathsf{index}(k, M_{\mathsf{id}})]\end{aligned}$$

Since $|C_k| = t$ by construction and $|B| = t-1$, we know that $C_k \setminus B$ is not empty. Because for all $\mathsf{id} \in C_k \setminus B$, $\tilde{\mathsf{mq}}_{\mathsf{id}}^{\mathsf{sid}}$ is uniformly random by definition, we conclude that $\bigoplus_{\mathsf{id} \in C_k \setminus B} \tilde{\mathsf{mq}}_{\mathsf{id}}^{\mathsf{sid}}[\mathsf{index}(k, M_{\mathsf{id}})]$ is unformly random, and so is $\tilde{\mathsf{tok}}^{\mathsf{sid}}[\mathsf{index}(k, M_0)]$ and $\tilde{\mathsf{tok}}^{\mathsf{sid}}$ for all $\mathsf{sid} \in [l]$. Thus by construction of MQuery, $\tilde{\mathsf{mq}}_0^{\mathsf{sid}}$ is also uniformly random for all $\mathsf{sid} \in [l]$. This proves the perfect indistinguishability.

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

$$\langle (\tilde{\mathsf{mq}}_0^{\mathsf{sid}})_{[l]}, \mathsf{aux}\rangle \overset{c}{\equiv} \langle (\mathsf{mq}_0^{\mathsf{sid}})_{[l]}, \mathsf{aux}\rangle$$

via a reduction proof.

Suppose there exists a PPT distinguisher $\mathcal{D}$ who can distinguish between the two ensembles; we will construct an adversary $\mathcal{A}$ using $\mathcal{D}$ as a subroutine to play the following game:

The challenger $\mathcal{C}$ is tasking $\mathcal{A}$ to distinguish between a uniformly random string

$$r_0 \leftarrow_\$ \{0,1\}^{\sum_{\mathsf{id}\in\mathsf{AID}\setminus B} l\cdot|M_{\mathsf{id}}|}$$

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

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

where each $s_{\mathsf{id}} \leftarrow_\$ \{0,1\}^\lambda$ is sampled uniformly at random. Upon given $r_b, b\in\{0,1\}$, $\mathcal{A}$ defines

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

and $\mathsf{mq}_{\mathsf{id}}^{\mathsf{sid},*} := \mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}$ for the rest of $\mathsf{id}\in B$ and $\mathsf{sid}\in[l]$.

Next, $\mathcal{A}$ constructs $\mathsf{mq}_0^{\mathsf{sid},*}$ out of $\mathsf{mq}_{\mathsf{id}}^{\mathsf{sid},*}, \mathsf{id}\in\mathsf{AID}$ i.e.

$$\forall\mathsf{sid}\in[l] : \mathsf{tok}^{\mathsf{sid},*} = \text{MToken}\left((M_{\mathsf{id}})_{\mathsf{ID}}, \left(\mathsf{mq}_{\mathsf{id}}^{\mathsf{sid},*}\right)_{\mathsf{AID}}\right)$$

$$\forall\mathsf{sid}\in[l] : \mathsf{mq}_0^{\mathsf{sid},*} = \text{MQuery}\left(M_0, \mathsf{tok}^{\mathsf{sid},*}, k^{\mathsf{sid}}\right)$$

and inputs $((\mathsf{mq}_0^{\mathsf{sid},*})_{[l]}, \mathsf{aux})$ to $\mathcal{D}$. If $\mathcal{D}$ determines the input is the first ensemble, then $\mathcal{A}$ outputs 0; otherwise, $\mathcal{A}$ outputs 1. By variable-length PRG assumption, we conclude that the advantage of $\mathcal{A}$ of guessing the correct string is negligible, so the two ensembles are indistinguishable.

Part 3: By "transitivity" of Part 1 and 2,

$$\langle r, \mathsf{aux}\rangle \overset{p}{\equiv} \langle(\tilde{\mathsf{mq}}_0^{\mathsf{sid}})_{[l]}, \mathsf{aux}\rangle \overset{c}{\equiv} \langle(\mathsf{mq}_0^{\mathsf{sid}})_{[l]}, \mathsf{aux}\rangle$$
$$\implies \langle r, \mathsf{aux}\rangle \overset{c}{\equiv} \langle(\mathsf{mq}_0^{\mathsf{sid}})_{[l]}, \mathsf{aux}\rangle$$

thus proving the theorem. □

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 if some PRG seeds are leaked to the attacker.

Theorem 4 (Synchronized APIR Privacy)**.** Suppose Π^{sPIR} is λ-sPIR privacy-preserving according to Definition A.2 and $\mathsf{PRG}(s,l)\to r$ is a variable-length PRG, then Synchronized APIR scheme Π^{SynAPIR} is (λ,t)-APIR privacy-preserving according to Definition 6.1.

Proof sketch. We first observe that if $0\notin C$ in $\mathsf{PrivA}_{\mathcal{A},\Pi^{\mathsf{SynAPIR}},t}$ i.e. main server $\mathcal{S}_0$ is not compromised, then $\mathcal{A}$ 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 therefore only focus on the scenario where $\mathcal{A}$ has chosen $0\in C$.

We want to show that the view of $\mathcal{A}$ in $\mathsf{PrivA}_{\mathcal{A},\Pi^{\mathsf{SynAPIR}},t}$ is computationally indistinguishable between when $b=0$ and $b=1$. Formally, define the view of $\mathcal{A}$ in the experiment as

$$\mathsf{outview}_{\mathcal{A}} := \langle(k^{\mathsf{sid}})_{\mathsf{sid}\neq\mathsf{sid}^*}, \left(k_0^{\mathsf{sid}^*}, k_1^{\mathsf{sid}^*}\right), \mathsf{auxout}\rangle$$

for the outputs of $\mathcal{A}$, where

$$\mathsf{auxout} := (l, \mathsf{Cat}_0, (\mathsf{Dig}_{\mathsf{id}})_{\mathsf{AID}}, C)$$

and

$$\mathsf{inview}_{\mathcal{A}}(b) :=$$
$$\langle(s_{\mathsf{id}})_{C\setminus\{0\}}, \left(\mathsf{mq}_0^{\mathsf{sid}}, \mathsf{sq}^{\mathsf{sid}}\right)_{\mathsf{sid}\neq\mathsf{sid}^*}, \left(\mathsf{mq}_{0,b}^{\mathsf{sid}^*}, \mathsf{sq}_b^{\mathsf{sid}^*}\right), \mathsf{auxin}\rangle$$

for the inputs of $\mathcal{A}$, where

$$\mathsf{auxin} := (\mathsf{spk}, (M_{\mathsf{id}})_{\mathsf{ID}}, S)$$

and finally

$$\mathsf{view}_{\mathcal{A}}(b) := \langle\mathsf{outview}_{\mathcal{A}}, \mathsf{inview}_{\mathcal{A}}(b)\rangle$$

We want to show that

$$\mathsf{view}_{\mathcal{A}}(0) \overset{c}{\equiv} \mathsf{view}_{\mathcal{A}}(1)$$

with respect to the security parameter λ through the follows steps:

1. Similarly to $\mathsf{inview}_{\mathcal{A}}(b)$, we define

   $$\widetilde{\mathsf{inview}}_{\mathcal{A}}(b) :=$$
   $$\langle(s_{\mathsf{id}})_{C\setminus\{0\}}, \left(\mathsf{mq}_0^{\mathsf{sid}}, \mathsf{sq}^{\mathsf{sid}}\right)_{\mathsf{sid}\neq\mathsf{sid}^*}, \left(\mathsf{mq}_{0,b}^{\mathsf{sid}^*}, \tilde{\mathsf{sq}}\right), \mathsf{auxin}\rangle$$

   where $\tilde{\mathsf{sq}} \leftarrow \text{SQuery}(\mathsf{spk}, 0)$, and

   $$\widetilde{\mathsf{view}}_{\mathcal{A}}(b) := \langle\mathsf{outview}_{\mathcal{A}}, \widetilde{\mathsf{inview}}_{\mathcal{A}}(b)\rangle$$

   By λ-sPIR privacy assumption, we claim that

   $$\mathsf{view}_{\mathcal{A}}(0) \overset{c}{\equiv} \widetilde{\mathsf{view}}_{\mathcal{A}}(0)$$

2. By Theorem 3, we claim that

   $$\widetilde{\mathsf{view}}_{\mathcal{A}}(0) \overset{c}{\equiv} \widetilde{\mathsf{view}}_{\mathcal{A}}(1)$$

   because mPIR queries are pseudorandom.

3. Similarly to step 1, by λ-sPIR privacy assumption we claim that
$$\widetilde{\mathsf{view}}_{\mathcal{A}}(1) \overset{c}{\equiv} \mathsf{view}_{\mathcal{A}}(1)$$

4. By "transitivity", we claim that
$$\mathsf{view}_{\mathcal{A}}(0) \overset{c}{\equiv} \widetilde{\mathsf{view}}_{\mathcal{A}}(0) \overset{c}{\equiv} \widetilde{\mathsf{view}}_{\mathcal{A}}(1) \overset{c}{\equiv} \mathsf{view}_{\mathcal{A}}(1)$$

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 provide a good 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 multiple DNS cache servers keep separate tables for different top-level domains (TLD) and types of records. Each server gathers the query statistics per domain from non-private DNS traffic in the past 24 hours and updates the cache table with the records of the 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 inform 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 to simulate 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 only NS records. 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 is 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 |record 1| + |record 2|+, . . . , +|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 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.

Statistics	Values
#records	382M
#domains	159M
avg. #records per domain	2.4
avg. |domain name|	17.7 bytes
avg. |records| per domain	73.1 bytes
max. |records| per domain	759 bytes
TTL	2 days

Table 1: **Statistics for NS records in ICANN's .com gTLD Zone File (M=10^6).** "#records" indicates the number of records. "#domains" indicates the number of domains. "|records| per domain" specifies the total size in bytes of all records linked to each domain, where each NS record is encoded as TTL|CLASS|TYPE|LEN|NSDOMAIN. The TTLs are of the same value of 172800 across all the records

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 names in DNS queries follows a Zipfian distribution [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 compute 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, accumulating to 22 million queries in the 24-hour cycle. The average QPS in a 24-hour window for the regional setting is 7032.3, accumulating to 608 million queries in the 24-hour cycle.

We follow these assumptions and parameters to simulate three cache servers for Synchronized APIR for the local and regional experiment: 1 main and two assisting servers with the collusion threshold of $t = 2$. The cache table size, or $|\mathsf{DB}|$, 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* [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 $|\mathsf{DB}|$ popular ranks are kept in the cache table. The exact keywords and values in the cache tables are randomly generated; this does not affect the scheme's performance 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 [15] and Jung *et al.* [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 there is no universally true and accurate distribution.

Instead, in our experiments, we aim to demonstrate 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 three 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., $|S|/|\mathsf{DB}_0| \times 100\%$. The results indices that $s = 1.0$ is an optimal parameter. (Coincidentally, this is close to 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.

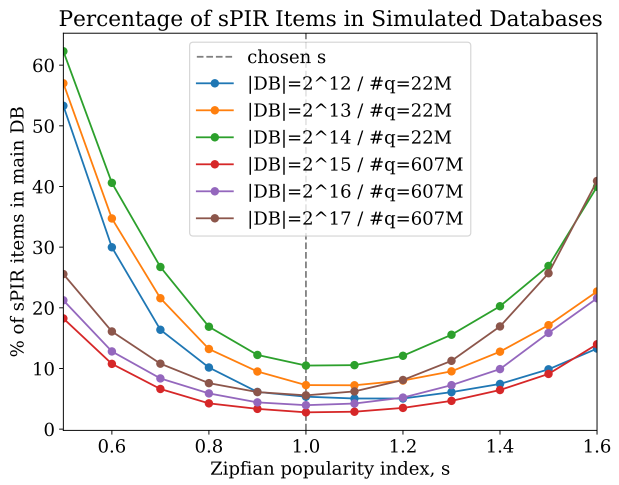

Figure 6: **The effect of Zipfian distribution on the percentage of sPIR items in simulated databases for 3 Synchronized APIR servers at collusion threshold t = 2.** #q denotes the total number of simulated queries. $s = 1.0$ is chosen as an optimal parameter.

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 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 each party spends in milli-seconds.

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

Parameters	Values
#servers	3
collusion threshold t	2
Zipfian s	1.0
|value|	1,024 bytes
security parameter	128
SealPIR Parameters	**Values**
$\log(\text{plaintext modulus}), d = 1$	14
$\log(\text{plaintext modulus}), d = 2$	16

Table 2: **Shared parameters between local and regional experiments.** "#servers" indicates the number of servers. |value| indicates the length of each value. d is the query dimensionality of SealPIR.

Parameters	Local	Regional
#sampling queries	22M	608M
$\lvert\mathsf{DB}\rvert$	$2^{13} \sim 8$K	$2^{16} \sim 66$K
|hash|(bits)	43	54
Statistics		
% of sPIR	7.3	3.9
Pr[failure]	$\lessapprox 1.3 \times 10^{-5}$	$\lessapprox 1.2 \times 10^{-5}$

Table 3: **Specific parameters and statistics for local and regional experiments.** "#sampling queries" indicates the number of queries to simulate the cache tables. "$\lvert\mathsf{DB}\rvert$" indicates the number of domain names in each cache table. "% of sPIR" indicates the percentage of sPIR items in the main cache table (DB_0). "|hash|" indicates the hash length in bits. "Pr[failure]" indicates the probability of failure of Synchronized APIR according to Theorem 2.

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 cannot handle databases larger than a certain size for $d = 1$.

In the 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 the 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 determines the noise budget in the underlying SEAL fully homomorphic encryption scheme (where 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 are calculated from the simulated databases. The probability of failure, i.e., the probability of at least one hash collision 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 baseline 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 significant gaps in query size between the small and large number of sPIR query bits.

Where Synchronized APIR outperforms the baseline is in computational cost. According to Table 5, the computational costs of the assisting servers ($\mathcal{S}_1$ and $\mathcal{S}_2$) are nearly negligible across the experiments. The computational costs during Synchronization and Setup are slightly cheaper for the main server ($\mathcal{S}_0$) in Synchronized APIR than in the baseline and somewhat more expensive for the client in Synchronized APIR than in the baseline. However, the more significant part is the recurring costs of the main server during Query. Here, in the local setting, the computational cost of the main server in $d = 1$ and $d = 2$ Synchronized APIR are 84% and 25% that of the baseline, respectively. In the regional setting, 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 a low one-time communication cost but high recurring costs.

Phases	SealPIR, $d=2$	Sync-APIR, $d=1$		Sync-APIR, $d=2$	
	$\mathcal{S}_0$	$\mathcal{S}_0$	$\mathcal{S}_1+\mathcal{S}_2$	$\mathcal{S}_0$	$\mathcal{S}_1+\mathcal{S}_2$
Local					
Synchronization					
Download	44K	95K	89K	95K	89K
Setup					
Upload (spk)	3.5M	3.5M	-	3.5M	-
Upload (others)	-	769	19K	769	19K
Query (per session)					
Upload	64K	34K	-	65K	-
Download	257K	32K	2K	257K	2K
Regional					
Synchronization					
Download	352K	799K	704K	799K	704K
Setup					
Upload (spk)	3.5M	3.5M	-	3.5M	-
Upload (others)	-	3.1K	154K	3.1K	154K
Query (per session)					
Upload	64K	40K	-	72K	-
Download	257K	32K	2K	257K	2K

Table 4: **Client's communication costs for one query in local and regional PIR experiment (in bytes, K=2^{10}, M=2^{20}).** "$\mathcal{S}_1+\mathcal{S}_2$" indicates the client's combined upload/download costs to/from Server 1 and 2. The upload costs during Setup are listed separately between the SEAL public key and other public parameters.

2. $d = 1$ Synchronized APIR offers 4.6x-4.7x one-time communication cost and ~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 ~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 a significant advantage over SealPIR in the long run after the one-time cost, but what does this mean 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, 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 compared to SealPIR at the same computational cost. However, a downside is the recurring upload cost may double with a small database.

Phases	SealPIR, $d=2$		Sync-APIR, $d=1$			Sync-APIR, $d=2$		
	$\mathcal{C}$	$\mathcal{S}_0$	$\mathcal{C}$	$\mathcal{S}_0$	$\mathcal{S}_1$ or $\mathcal{S}_2$	$\mathcal{C}$	$\mathcal{S}_0$	$\mathcal{S}_1$ or $\mathcal{S}_2$
Local								
Synchronization	< 1	2	20	3	3	20	3	3
Setup	107	150	98	89	1	93	86	1
Query (per session)	8	90	1	76	< 1	8	23	< 1
Regional								
Synchronization	7	31	184	35	42	184	35	42
Setup	106	653	221	127	16	214	119	16
Query (per session)	8	414	9	323	4	16	48	4

Table 5: **Computational costs for one query in the local and regional PIR experiment (in milliseconds).** "$\mathcal{S}_1$ or $\mathcal{S}_2$" indicates the computational cost on $\mathcal{S}_1$ or $\mathcal{S}_2$ since they cost the same CPU time.

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 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 outperforms the baseline single-server PIR in communication or computational costs. Although the costs are high compared 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 the University of California, Davis, for his 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

[1] Caching DNS capacity and performance guidelines, Nov 2021. Available at `https://www.cisco.com/c/en/us/td/docs/net_mgmt/prime/network_registrar/10-1/install/guide/Install_Guide/Install_Guide_appendix_010001.html`.

[2] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR: Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies*, 2(2016):155–174, 2016.

[3] Andris Ambainis. Upper bound on the communication complexity of private information retrieval. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming*, pages 401–407, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

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

[5] Daniel J Bernstein et al. ChaCha, a variant of Salsa20. In *Workshop record of SASC*, volume 8, pages 3–5. Lausanne, Switzerland, 2008.

[6] Hao Chen, Kyoohyung Han, Zhicong Huang, Amir Jalali, and Kim Laine. Simple encrypted arithmetic library v2.3.0. 2017.

[7] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50. IEEE, 1995.

[8] David Eppstein, Michael T. Goodrich, Frank Uyeda, and George Varghese. What's the difference? efficient set reconciliation without prior context. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 218–229, New York, NY, USA, 2011. Association for Computing Machinery.

[9] Giulia Fanti and Kannan Ramchandran. Efficient private information retrieval over unsynchronized databases. *IEEE Journal of Selected Topics in Signal Processing*, 9(7):1229–1239, 2015.

[10] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. DNS performance and the effectiveness of caching. *IEEE/ACM Transactions on networking*, 10(5):589–603, 2002.

[11] Muhammad Haris Mughees, Hao Chen, and Ling Ren. OnionPIR: Response efficient single-server pir. 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.

[12] M. Sayrafiezadeh. The birthday problem revisited. *Mathematics Magazine*, 67(3):220–223, 1994.

[13] Radu Sion and Bogdan Carbunar. On the computational practicality of private information retrieval. In *Proceedings of the Network and Distributed Systems Security Symposium*, pages 2006–06. Internet Society Geneva, Switzerland, 2007.

[14] Julien P. Stern. A new and efficient all-or-nothing disclosure of secrets protocol. In Kazuo Ohta and Dingyi Pei, editors, *Advances in Cryptology — ASIACRYPT'98*, pages 357–371, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[15] Zheng Wang. Analysis of DNS cache effects on query distribution. *The Scientific World Journal*, 2013, 2013.

A Appendix

A.1 sPIR Correctness and Privacy Definition

Definition A.1 (sPIR Correctness)**.** Following Definition 3.1, scheme Π^{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

- $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \textsc{SGen}(1^\lambda)$
- $\mathsf{sq} \leftarrow \textsc{SQuery}(\mathsf{spk}, i, m)$
- $\mathsf{sr} \leftarrow \textsc{SReply}(\mathsf{spk}, V, \mathsf{sq})$
- $\mathsf{sa} \leftarrow \textsc{SDecode}(\mathsf{ssk}, \mathsf{sr})$

then $\mathsf{sa} = v_i$.

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

1. $\mathcal{A}$ chooses and outputs m.
2. The parameters are generated $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \textsc{SGen}(1^\lambda)$ and spk is given to $\mathcal{A}$.
3. $\mathcal{A}$ is given oracle access to $\textsc{SQuery}(\mathsf{spk}, \cdot)$.
4. $\mathcal{A}$ 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 $\mathsf{sq}^* \leftarrow \textsc{SQuery}(\mathsf{spk}, i_b^*)$ is generated and sq^* is given to $\mathcal{A}$.
5. $\mathcal{A}$ is given more oracle access to $\textsc{SQuery}(\mathsf{spk}, \cdot)$.
6. $\mathcal{A}$ outputs $b^* \in \{0, 1\}$. The experiment's output is 1 if $b^* = b$ and 0 otherwise.

Π^{sPIR} is λ-sPIR privacy-preserving if for all PPT adversary $\mathcal{A}$, there exists a negligible function negl such that

$$\Pr\left[\mathsf{PrivS}_{\mathcal{A},\Pi^{\mathsf{sPIR}}}(1^\lambda) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$$