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 face the challenges of either high computational costs or restrictive security assumptions, resulting in a barrier to deployment. In this work, we introduce *Assisted Private Information Retrieval* (APIR), a new PIR framework for keyword-value databases generalizing multi-server PIR and relaxing its database consistency assumption. We propose the construction of *Synchronized APIR*, an efficient hybrid APIR scheme combining black-box single-server PIR and non-black-box multi-server PIR. To evaluate the scheme, we apply it to a proof-of-concept privacy-preserving DNS application. The experiment results demonstrate that Synchronized APIR outperforms the baseline single-server PIR protocol in communication and computational cost 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 scenario in which 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 Tor to hide her IP address, 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) [16] and DNS-over-TLS (DoT) [17]. 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 face practical challenges. The family of PIR schemes that requires one server to operate, or *single-server* PIR [2–4, 8, 10, 20, 24], makes minimal assumptions about data management. However, 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 [7, 9, 11, 12, 14, 15], are orders of magnitude more efficient than single-server PIR to deploy. However, 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 where both requirements hold simultaneously. 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 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. Correctness is defined by the copy operated by the server chosen to be the *main server*.

We present *Synchronized APIR* (SAPIR), an APIR scheme that requires the client to synchronize the "view" of the databases across the servers before making queries. SAPIR is a hybrid protocol combining a black-box single-server PIR scheme and a non-black-box multi-server PIR scheme similar to CGKS protocol [7]. Due to the black-box use of single-server PIR, improvements in single-server PIR imply improvements in SAPIR. We provide a formal analysis and an implementation of Synchronized APIR in Rust using SealPIR [4] as the underlying single-server PIR scheme.

We apply SAPIR to demonstrate a proof-of-concept privacy-preserving DNS 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 after the initial one-time cost, privacy-preserving DNS via SAPIR outperforms SealPIR, a baseline single-server PIR, in communication and computational costs.

2 Related Works

PIR over unsynchronized databases Our work is partly inspired by Fanti *et al.* [13], 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 requires 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 the APIR problem with two additional limitations: 1) the databases are strictly indexed, and 2) the client cannot query any missing values in at least one of the databases.

Oblivious DNS Prior techniques have been proposed for Oblivious DNS (ODNS) [19, 22], and Oblivious DNS over HTTPS (ODoH) [23]. Oblivious DNS techniques offer anonymity for DNS clients by decoupling their IP addresses from their queries via a proxy server between them and the DNS resolvers. Unlike PIR approaches, these techniques require minimal modifications to DNS clients and servers for deployment. However, their privacy definition is fundamentally different from ours. Despite the clients' anonymity in Oblivious DNS, the DNS servers can aggregate anonymized queries. In our threat model, we consider data concentration—anonymized or otherwise—by DNS operators as a security and privacy risk. Our approach addresses this by making it impossible to aggregate such data by design.

3 Preliminaries

3.1 Single-Server PIR

Single-server PIR (sPIR) [2–4, 8, 10, 20, 24] 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 while keeping the communication cost lower than the trivial PIR.

We formally define sPIR in Definition 3.1 below. This definition will be referred to in our construction in Section 5.

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

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

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.

The formal definitions of sPIR correctness and privacy are provided in Definition A.1 and Definition A.2, respectively, in Appendix A. Intuitively, an sPIR scheme is correct if the answer matches the target value. An sPIR scheme is privacy-preserving if a query encoding index i and a query encoding index j are indistinguishable for any i and j.

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, SealPIR offers a significant advantage over its predecessors via a query compression technique, which cuts down the size of a query by a significant factor.

3.2 Multi-Server PIR

Multi-server PIR (mPIR) [7, 9, 11, 12, 14, 15] is the family of PIR schemes that requires the client to interact with multiple database-operating servers to query. 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 a modified version of CGKS [7], one of the most well-known mPIR schemes, in a non-block-box manner. For instructional purposes, we provide a tutorial of CGKS [7] in Appendix B. In our construction, we modify CGKS so that the servers do not require consistent databases and the participating parties use a PRG to generate all but one query and reply 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. We provide a construction of APIR called Synchronized APIR in Section 5.

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 operated by main server $\mathcal{S}_0$. This process is assisted by n assisting servers $\mathcal{S}_1, \ldots, \mathcal{S}_n$ who independently operate keyword-value databases $\mathsf{DB}_1, \ldots, \mathsf{DB}_n$, respectively. The purpose of assistance includes but is not limited to reducing operational costs. $\mathsf{DB}_1, \ldots, \mathsf{DB}_n$ are not required to be the exact copies 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 $v \neq v'$. This database definition differs from sPIR and mPIR in Section 3 where databases are assumed to be indexed and consistent. APIR is formally defined below.

Notations

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

Definition 4.1 (APIR Scheme)**.** Given a set of $n+1$ database-operating servers. An APIR scheme is a tuple $\Pi^{\mathsf{APIR}} =$ (ServGen, CliGen, Query, Reply, 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 $\mathsf{DB}_{\mathsf{id}}$.
- $\textsc{CliGen}(1^\lambda, t, (\mathsf{par}_{\mathsf{id}})_{\mathsf{ID}}) \to (\mathsf{pk}, \mathsf{sk})$, 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.
- $\textsc{Query}(\mathsf{pk}, \mathsf{sk}, k) \to (\mathsf{q}_{\mathsf{id}})_{\mathsf{ID}}$, where q_{id} is the query for server id from query keyword $k \in \mathsf{Keys}(\mathsf{DB}_0)$ targeting database value $\mathsf{DB}_0[k]$.
- $\textsc{Reply}(\mathsf{id}, \mathsf{pk}, \mathsf{DB}_{\mathsf{id}}, \mathsf{q}_{\mathsf{id}}) \to \mathsf{r}_{\mathsf{id}}$, where r_{id} is the reply for query q_{id} targeting database $\mathsf{DB}_{\mathsf{id}}$.
- $\textsc{Decode}(\mathsf{sk}, (\mathsf{r}_{\mathsf{id}})_{\mathsf{ID}}) \to \mathsf{a}$, where a is the answer decoded from replies $(\mathsf{r}_{\mathsf{id}})_{\mathsf{ID}}$.

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

The correctness of APIR is defined according to the content of DB_0. That is, if the client generates queries with keyword k, then $\mathsf{DB}_0[k]$ is defined to be the correct answer (even if there is another $\mathsf{DB}_{\mathsf{id}}$ 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 of scheme Π^{APIR}, suppose any set ID, databases $\mathsf{DB}_{\mathsf{id}}$ for all $\mathsf{id} \in \mathsf{ID}$, and keyword $k \in \mathsf{Keys}(\mathsf{DB}_0)$, and

- $\forall \mathsf{id} \in \mathsf{ID} : \mathsf{par}_{\mathsf{id}} \leftarrow \textsc{ServGen}(\mathsf{id}, \mathsf{DB}_{\mathsf{id}})$
- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \textsc{CliGen}(1^\lambda, t, (\mathsf{par}_{\mathsf{id}})_{\mathsf{ID}})$
- $(\mathsf{q}_{\mathsf{id}})_{\mathsf{ID}} \leftarrow \textsc{Query}(\mathsf{pk}, \mathsf{sk}, k)$
- $\forall \mathsf{id} \in \mathsf{ID} : \mathsf{r}_{\mathsf{id}} \leftarrow \textsc{Reply}(\mathsf{id}, \mathsf{pk}, \mathsf{DB}_{\mathsf{id}}, \mathsf{q}_{\mathsf{id}})$
- $\mathsf{a} \leftarrow \textsc{Decode}(\mathsf{sk}, (\mathsf{r}_{\mathsf{id}})_{\mathsf{ID}})$

Then, Π^{APIR} is correct if $\mathsf{a} = \mathsf{DB}_0[k]$ with a non-zero probability.

The privacy definition of APIR captures the indistinguishability 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. We model a realistic scenario in which the client is not informed of which servers are compromised or whether they are compromised at all. To this end, 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 APIR privacy using a game-based definition below.

Definition 4.3 ((λ, t)-APIR Privacy)**.** Given security parameter λ, collusion threshold t, adversary $\mathcal{A}$, and oracle $\mathcal{O}$, define an APIR privacy experiment $\mathsf{PrivA}_{\mathcal{A},\Pi^{\mathsf{APIR}},t}(1^\lambda)$ for 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 $\mathcal{O}$.
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 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 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 privacy-preserving for all PPT adversary $\mathcal{A}$ if there exists a negligible function negl such that

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

5 Synchronized APIR

This section presents our construction of APIR called Synchronized APIR (SAPIR). SAPIR requires 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. SAPIR uses a black-box sPIR scheme described in Section 3.1 in its hybrid PIR construction and a non-black-block construction of modified CGKS.

For the rest of this section, we detail the protocol description, optimization techniques, and implementation of SAPIR. For instructional purposes, we provide a high-level concept of SAPIR in Appendix C. The formal analysis of SAPIR is detailed in Appendix D.

5.1 Overview

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 1: *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 SAPIR as shown in Figure 1 in detail.

5.2 Synchronization Phase

During the Synchronization phase, $\mathcal{C}$ synchronizes the global view of the databases to select the consistent keyword-value pairs to be retrieved via mPIR and the rest via sPIR. This selection 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 (see step ①A). The purpose of a catalog is to uniquely represent a database while keeping the communication cost low.

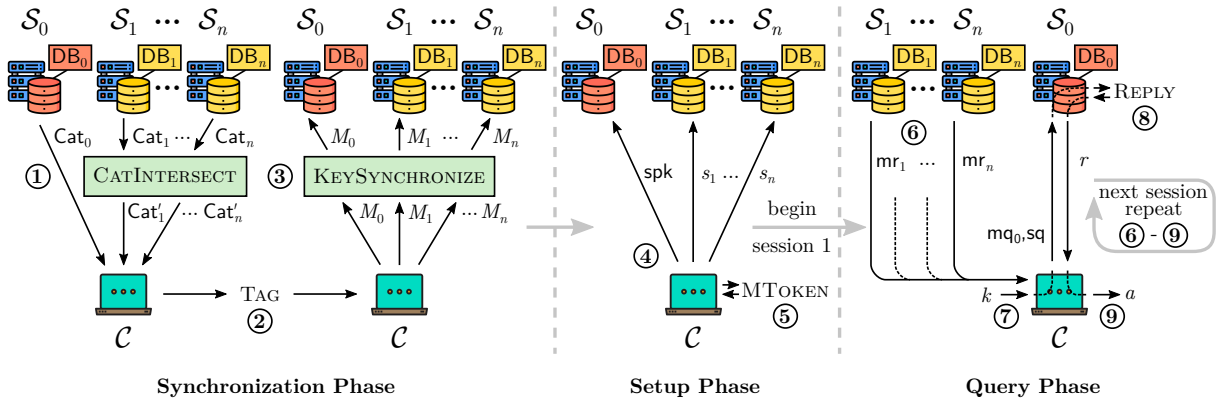

Figure 1: *Overview of Synchronized APIR.* Synchronized APIR comprises three distinct phases between client $\mathcal{C}$, main server $\mathcal{S}_0$, and assisting Servers $\mathcal{S}_1 - \mathcal{S}_n$: Synchronization, Setup, and Query. Synchronization and Setup are completed once at the start, while a new Query session can be repeated with every new query keyword k. ① The servers generate catalogs $\mathsf{Cat}_{\mathsf{id}}$ from database $\mathsf{DB}_{\mathsf{id}}$. $\mathcal{C}$ obtains catalog Cat_0 and catalog intersections $\mathsf{Cat}'_{\mathsf{id}}$ via the Catalog Intersection protocol. ② $\mathcal{C}$ tags the catalogs for sets M_{id}, indicating the keywords retrievable via mPIR in $\mathsf{DB}_{\mathsf{id}}$. ③ $\mathcal{C}$ sends M_{id} to $\mathcal{S}_{\mathsf{id}}$ via the Keyword Synchronization protocol. ④ $\mathcal{C}$ generates the sPIR public-secret key pair and PRG seeds. The sPIR public key spk is sent to $\mathcal{S}_0$ and PRG seed s_{id} to $\mathcal{S}_{\mathsf{id}}$. ⑤ $\mathcal{C}$ generate mPIR query tokens using the PRG seeds. ⑥ Each assisting server $\mathcal{S}_{\mathsf{id}}$ generates the reply $\mathsf{mr}_{\mathsf{id}}$ from the PRG seed and database $\mathsf{DB}_{\mathsf{id}}$ and sends $\mathsf{mr}_{\mathsf{id}}$ to $\mathcal{C}$. ⑦ $\mathcal{C}$ generates mPIR query mq_0 and sPIR query sq from keyword k and sends them to $\mathcal{S}_0$. ⑧ $\mathcal{S}_0$ generates reply r with query mq_0 and sq and database DB_0 and sends r to $\mathcal{C}$. ⑨ $\mathcal{C}$ decodes r to obtain the answer a. Step ⑥-⑨ can be repeated in the next session with a new keyword k.

We refer to step ① - ③ in Figure 1 and ① - ② in 2 as a demonstration to describe the Synchronization protocol 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.

Synchronization Protocol

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

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:

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

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

where $\mathsf{H}(\cdot)$ is a universal hash function with short hash values, chosen randomly from the universal family by $\mathcal{C}$.

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 5.6. 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:

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

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 to be queried via mPIR from $\mathcal{S}_{\mathsf{id}}$; S is the sPIR keyword set, i.e., the set of keywords to be queried via sPIR from $\mathcal{S}_0$. Tag is detailed in Algorithm 1 below.

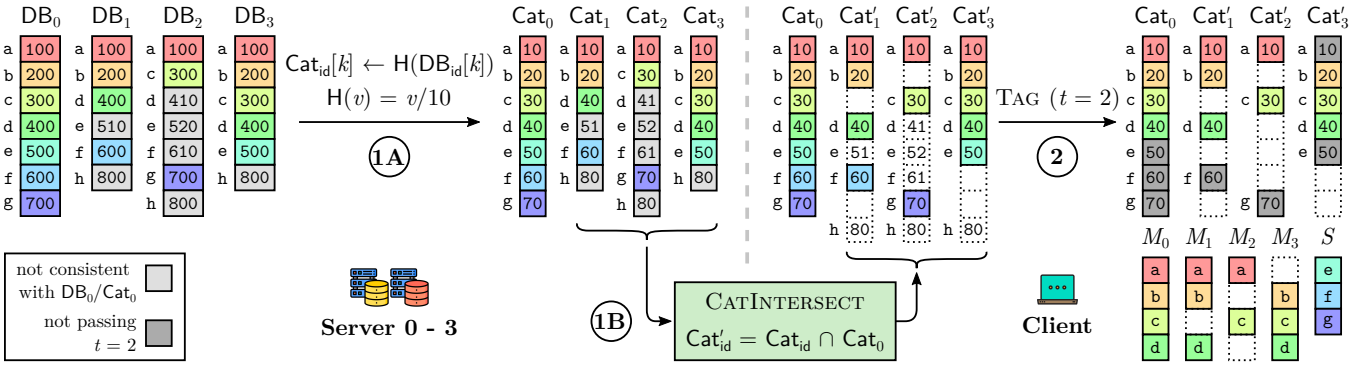

Figure 2: *Synchronization.* An example of databases in a Synchronized APIR setting with three assisting servers that undergo the Synchronization phase. ①A: $\mathcal{S}_{\mathsf{id}}$ generates catalog $\mathsf{Cat}_{\mathsf{id}}$ from database $\mathsf{DB}_{\mathsf{id}}$. A toy hash function $\mathsf{H}(v) = v/10$ is chosen to demonstrate that an appropriate hash function should be able to produce short unique hash values. Generally, when the distribution of database values is unknown, universal hash functions should be applied to reduce hash collisions. ①B: $\mathcal{C}$ obtains catalog Cat_0 from $\mathcal{S}_0$ and catalog intersections $\mathsf{Cat}'_{\mathsf{id}}$ from $\mathcal{S}_{\mathsf{id}}$ via the Catalog Intersection protocol. Keyword-value pairs inconsistent with DB_0 in $\mathsf{DB}_1, \ldots, \mathsf{DB}_3$ and with Cat_0 in $\mathsf{Cat}_1, \ldots, \mathsf{Cat}_3$ are eliminated via catalog intersection. ② $\mathcal{C}$ decides the collusion threshold $t = 2$. To tag the catalogs, the pairs that pass the threshold for mPIR must have $t + 1 = 3$ copies: one copy in Cat_0 and two copies in $\mathsf{Cat}'_1, \mathsf{Cat}'_2, \mathsf{Cat}'_3$. $(\mathsf{a}, 10), (\mathsf{b}, 20), (\mathsf{c}, 30)$, and $(\mathsf{d}, 40)$ all pass the threshold ($(\mathsf{a}, 10)$ in Cat'_3 is redundant and disregarded). Neither of $(\mathsf{e}, 50), (\mathsf{f}, 60)$, or $(\mathsf{g}, 70)$ 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.

Algorithm 1 Catalog Tagging. C_k can be chosen randomly or with a specific optimization strategy.

procedure $\textsc{Tag}(\mathsf{Cat}_0, (\mathsf{Cat}'_{\mathsf{id}})_{\mathsf{AID}}, t)$
  $\forall \mathsf{id} \in \mathsf{ID} : M_{\mathsf{id}} \leftarrow \phi$ ▷ initialize to an empty set
  for $k \in \mathsf{Keys}(\mathsf{Cat}_0)$ **do**
    $G \leftarrow \{\mathsf{id} \in \mathsf{AID} \mid k \in \mathsf{Keys}(\mathsf{Cat}'_{\mathsf{id}}))\}$
    if $|G| \geq t$ **then**
      choose $C_k \subseteq G$ such that $|C_k| = t$
      $\forall \mathsf{id} \in C_k \cup \{0\} : M_{\mathsf{id}} \leftarrow M_{\mathsf{id}} \cup \{k\}$
    end if
  end for
  $S \leftarrow \mathsf{Keys}(\mathsf{Cat}_0) \setminus M_0$
  return $((M_{\mathsf{id}})_{\mathsf{ID}}, S)$
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 5.7.

5.3 Setup Phase

Suppose $l = \mathrm{poly}(\lambda)$, where λ is the security parameter, is the total number of queries that $\mathcal{C}$ will be making during the Query phase. We call each query during the Query phase a Query *session*, each denoted with session ID $\mathsf{sid} \in [l]$. Starting at $\mathsf{sid} = 1$, sid increases by 1 for each Query session.

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 1 and ⑤ in 3 to describe the Setup protocol below.

Notations

- $[l] := \{1 \ldots l\}$, the set of all l session ID's.
- $\mathsf{index}(a, A) := |\{b \in A \mid b < a\}|$, index of a in set A.

Setup Protocol

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

④ $\mathcal{C}$ generates an sPIR key pair $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \text{SGen}(\lambda)$ 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 is not predetermined and each $\mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}$ can be generated on the fly.

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

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

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.

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 ← {0}^{|M_0|}
3:     for k ∈ M_0 do
4:         C_k ← {id ∈ AID | k ∈ M_id}
5:         ∀id ∈ C_k ∪ {0} : i_id ← index(k, M_id)
6:         tok[i_0] ← ⊕_{id∈C} mq_id[i_id]
7:     end for
8:     return tok
9: end procedure
```

5.4 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 3 for a 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 and Setup in Section 5.3. We will now pick up step ⑥ and ⑦*-⑨* (* to denote the steps for this special case) in Figure 3 from here.

mPIR-Only Query Protocol for Session sid

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**	$\mathcal{C}$

⑥ For each assisting servers $\mathsf{id} \in \mathsf{AID}$, $\mathcal{S}_{\mathsf{id}}$ generates a reply from the query

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

and sends $\mathsf{mr}_{\mathsf{id}}^{\mathsf{sid}}$ to $\mathcal{C}$. MReply is described in Algorithm 3 below.

Algorithm 3 mPIR Reply Generation

```
1: procedure MReply(M_id, DB_id, mq_id)
2:     V ← (DB_id[k])_{k∈M_id}
3:     mr_id ← mq_id · V                ▷ dot product in GF(2)
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 $\mathcal{S}_0$

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

where MQuery is described in Algorithm 4 below. Finally, $\mathcal{C}$ sends $\mathsf{mq}_0^{\mathsf{sid}}$ to $\mathcal{S}_0$ to query.

Algorithm 4 mPIR Query Generation

```
1: procedure MQuery(M_0, tok, k)
2:     mq_0 ← tok
3:     if k ∈ M_0 then
4:         mq_0[i] ← mq_0[i] ⊕ 1, where i = index(k, M_0)
5:     end if
6:     return mq_0        ▷ note that mq_0 = tok if k ∉ M_0
7: 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 $\mathcal{C}$,

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

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

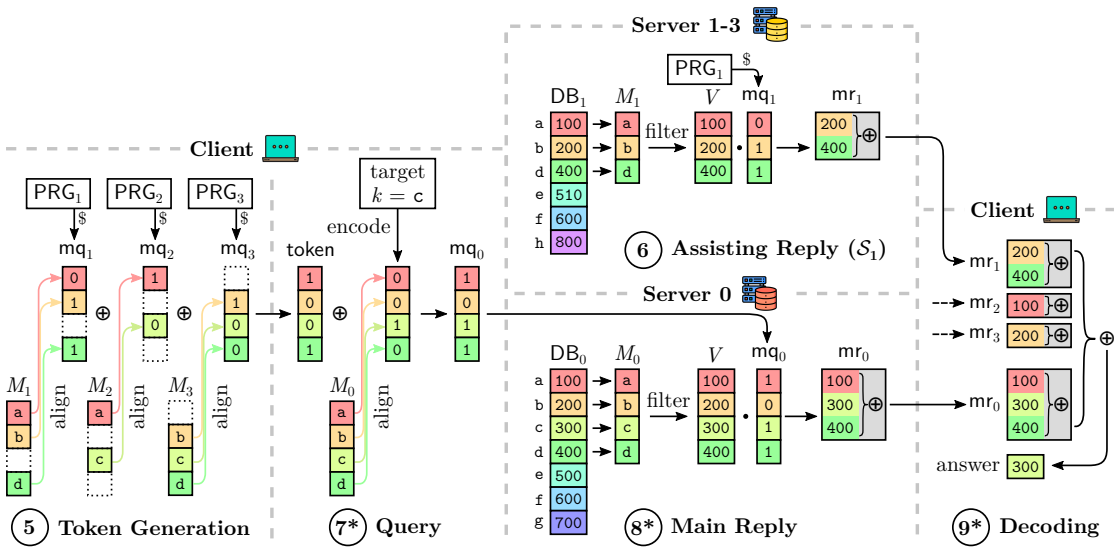

Figure 3: *mPIR-Only Query Phase Workflow.* The workflow of mPIR components during the Query phase to demonstrate how mPIR keyword-value pairs can be queried with the client's query keyword. ⑤ $\mathcal{C}$ generates mPIR query $\mathsf{mq}_{\mathsf{id}}$ with a PRG with seed s_{id} (represented by $\mathsf{PRG}_{\mathsf{id}}$ in the figure). The query bits of $\mathsf{mq}_{\mathsf{id}}$ can be aligned with respect to their associated keywords in M_{id}. $\mathcal{C}$ XORs these aligned bits to produce the query token. ⑥ Given $\mathsf{mq}_{\mathsf{id}}$ generated with $\mathsf{PRG}_{\mathsf{id}}$, $\mathcal{S}_{\mathsf{id}}$ generates the mPIR reply $\mathsf{mr}_{\mathsf{id}}$ as follows: the values of $\mathsf{DB}_{\mathsf{id}}$ are filtered with M_{id} by key, resulting in V. Then, the dot product in $\mathsf{GF}(2)$ is applied between V and $\mathsf{mq}_{\mathsf{id}}$ to produce $\mathsf{mr}_{\mathsf{id}}$. $\mathcal{S}_{\mathsf{id}}$ sends $\mathsf{mr}_{\mathsf{id}}$ to $\mathcal{C}$. ⑦* Suppose that $\mathcal{C}$ wishes to query the keyword c, she flips one bit of token at the index position of c in M_0 to produce the query mq_0. $\mathcal{C}$ sends mq_0 to $\mathcal{S}_0$. ⑧* $\mathcal{S}_0$ generates the reply mr_0 following ⑥. $\mathcal{S}_0$ sends mr_0 to $\mathcal{C}$. ⑨* $\mathcal{C}$ XORs all of $\mathsf{mr}_0, \ldots, \mathsf{mr}_3$ to obtain the answer $\mathsf{DB}_0[\mathsf{c}] = 300$.

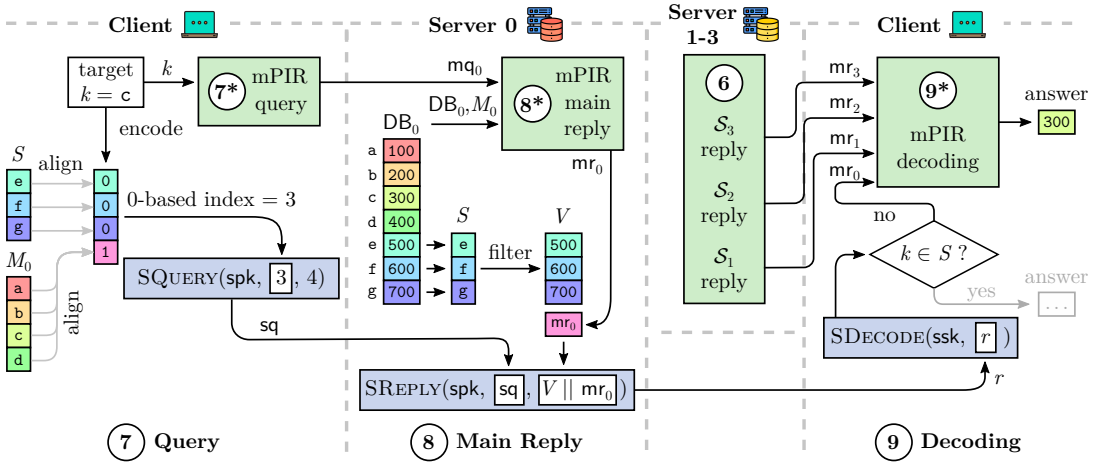

Figure 4: *Query Phase Workflow.* The workflow of hybrid components during the Query phase to demonstrate how keyword-value pairs in DB_0 can be queried with the client's query key. ⑥ This is identical to ⑥ in Figure 3. ⑦ $\mathcal{C}$ generates mPIR query mq_0 following ⑦* in Figure 3. Next, $\mathcal{C}$ generates sPIR query sq as follows: if $k \in S$, then k is encoded by its index position in S as an input to SQUERY; otherwise, if $k \in M_0$, then it is encoded as the last index position as an input to SQUERY. Here, because $k = \mathsf{c} \in M_0$, it is encoded as 3, the 0-based index of the last position. $\mathcal{C}$ sends mq_0 and sq to $\mathcal{S}_0$. ⑧ $\mathcal{S}_0$ generates the mPIR reply mr_0 according to step ⑧* in Figure 3. Next, $\mathcal{S}_0$ generates reply r as follows. First, the values of DB_0 are filtered with S by key, resulting in V. Then, mr_0 is appended to V as the last item, i.e., $V \| \mathsf{mr}_0$. Finally, the sPIR query sq and $V \| \mathsf{mr}_0$ are input to the sPIR reply generation algorithm SREPLY to produce r. $\mathcal{S}_0$ sends r to $\mathcal{C}$. ⑨ $\mathcal{C}$ decodes the replies for the answer as follows. First, r is decoded using secret key ssk with the sPIR decoding algorithm SDECODE. If $k \in S$, then $\mathcal{C}$ already has the final answer; otherwise, $\mathcal{C}$ obtains mr_0 from the decoding and follows step ⑨* in Figure 3. Here, $k = \mathsf{c} \notin S$, so $\mathcal{C}$ follows ⑨* to XOR all $\mathsf{mr}_0, \ldots, \mathsf{mr}_3$ to obtain the answer $\mathsf{DB}_0[\mathsf{c}] = 300$.

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.

Once the session is over, all parties increase sid by 1: $\mathsf{sid} \leftarrow \mathsf{sid} + 1$.

5.5 Query Phase

In this section, we will expand on the special case in Section 5.4 to describe the full Query phase. The Query phase can be repeated for every new query keyword k^{sid}. The Query session starts at $\mathsf{sid} = 1$ and increases by one for every new Query session. Below, we follow Figure 4 to describe step ⑥ - ⑨. We refer to step ⑦* - ⑨* from Section 5.4.

Query Protocol for Session sid

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**	$\mathcal{C}$

⑥ This is identical to ⑥ in Section 5.4.

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

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

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 $\mathcal{S}_0$.

Algorithm 5 Query Generation

```
1: procedure QUERY(M_0, S, spk, tok, k)
2:     mq_0 ← MQUERY(M_0, tok, k)
3:     sq ← ⊥                          ▷ initialize to null value
4:     if S ≠ φ then
5:         if k ∈ S then
6:             sq ← SQUERY(spk, index(k, S), |S| + 1)
7:         else
8:             sq ← SQUERY(spk, |S|, |S| + 1)
9:         end if
10:    end if
11:    return (mq_0, sq)               ▷ note that sq = ⊥ if S = φ
12: 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

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

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 REPLY(M_0, DB_0, mq_0, sq)
2:     mr_0 ← MREPLY(M_0, DB_0, mq_0)
3:     S ← Keys(DB_0) \ M_0
4:     if S ≠ φ then
5:         V ← (DB_0[k])_{k∈S}
6:         return r = SREPLY(spk, V‖mr_0, sq)
7:     else
8:         return r = mr_0
9:     end if
10: end procedure
```

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

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

Decode is described in Algorithm 7 below.

Algorithm 7 Decoding

```
1: procedure DECODE(S, ssk, k, (mr_id^sid)_{id∈AID}, r)
2:     if k ∈ S then
3:         return a = SDECODE(ssk, r)
4:     else
5:         if S ≠ φ then
6:             mr_0 ← SDECODE(ssk, r)
7:         else
8:             mr_0 ← r
9:         end if
10:        return a = ⊕_{id∈ID} mr_id
11:    end if
12: end procedure
```

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 $k^{\mathsf{sid}} \in S$.

Once the session is over, all parties increase sid by 1: $\mathsf{sid} \leftarrow \mathsf{sid} + 1$.

Considerations for sPIR schemes with a high preprocessing cost Many sPIR schemes require the server to preprocess the database before the reply generation step to reduce the online cost. As it pertains to SAPIR, $\mathcal{S}_0$ can compute $V \leftarrow (\mathsf{DB}_0[k])_{k \in S}$ in Algorithm 6 and preprocess V during the Synchronization phase. Later, when mr_0 is generated, $\mathcal{S}_0$ can preprocess mr_0 online and append it to the processed V.

While this is doable in many schemes like SealPIR [4], it is not the case for others. FrodoPIR [10], for example, requires the client to download a large matrix from the server during the preprocessing phase that is a product between the database matrix and a global pseudorandom matrix. It is not obvious how this matrix can be efficiently updated online for every new mr_0. To address this issue, we can modify our Query phase so that $\mathcal{S}_0$ independently processes mPIR and sPIR queries. That is, instead of appending mr_0 to V in Algorithm 6, $\mathcal{S}_0$ can send both $\mathsf{mr}_0 = \text{MR}\textsc{eply}(M_0, \mathsf{DB}_0, \mathsf{mq}_0)$ and $r = \text{SR}\textsc{eply}(\mathsf{spk}, V, \mathsf{sq})$ to the client. This increases the client's download cost by $|\mathsf{mr}_0|$, which is equivalent to the size of one database value.

5.6 Catalog Intersection

The Catalog Intersection protocol is a part of the Synchronization phase in Section 5.2 to reduce the communication cost of transferring catalog intersections. The idea is as follows: given that $\mathcal{C}$ has already obtained Cat_0 from $\mathcal{S}_0$, 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 transport by $\mathcal{S}_i$ and decompressed by $\mathcal{C}$. We describe the Catalog Intersection protocol below.

Catalog Intersection Protocol

Input

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

Output

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

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

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

where $\mathsf{G}(\cdot)$ is a universal hash function with short hash values, chosen randomly from the universal family by $\mathcal{C}$. $\mathcal{S}_{\mathsf{id}}$ 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:

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

5.7 Keyword Synchronization

The Keyword Synchronization protocol is a part of the Synchronization phase in Section 5.2 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.

Keyword Synchronization Protocol

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

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 the main server and of $\mathsf{Dig}_{\mathsf{id}}$ for the assisting servers:

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

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

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 KeyToIndex_0(M_0, Cat_0)
2:     I_0 ← {index(k, Keys(Cat_0)) | ∀k ∈ M_0}
3:     return I_0
4: end procedure
5: procedure KeyToIndex_id(M_id, Dig_id, Cat_id)
6:     D ← {G((k, h)) | ∀k ∈ M_id, h = Cat_id[k]}
7:     I_id ← {index(d, Dig_id) | ∀d ∈ D}
8:     return I_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}

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

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

IndexToKey is described in Algorithm 9 below.

Algorithm 9 Index-to-Key Mapping

procedure $\textsc{IndexToKey}_0(I_0, \mathsf{Cat}_0)$
  $M_0 \leftarrow \{k \in \mathsf{Keys}(\mathsf{Cat}_0) \mid \mathsf{index}(k, \mathsf{Keys}(\mathsf{Cat}_0)) \in I_0\}$
  return M_0
end procedure
procedure $\textsc{IndexToKey}_{\mathsf{id}}(I_{\mathsf{id}}, \mathsf{Dig}_{\mathsf{id}}, \mathsf{Cat}_{\mathsf{id}})$
  $D \leftarrow \{d \in \mathsf{Dig}_{\mathsf{id}} \mid \mathsf{index}(d, \mathsf{Dig}_{\mathsf{id}}) \in I_{\mathsf{id}}\}$
  $M_{\mathsf{id}} \leftarrow \{k \in \mathsf{Keys}(\mathsf{Cat}_{\mathsf{id}}) \mid \mathsf{G}((k, \mathsf{Cat}_{\mathsf{id}}[k])) \in D\}$
  return M_{id}
end procedure

5.8 Optimization Techniques

5.8.1 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. This cost can be reduced 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 $\tilde{\mathsf{H}}(k)$ and uses it as a keyword to query instead.

However, when the keyword space is large, this technique can introduce a high hash-collision rate, which can result in a false-positive query when $\tilde{\mathsf{H}}(k)$ replaces k. Our workaround for this problem is to make the servers modify the databases by prepending the keyword to the value so that a false positive can be detected at the end of the protocol. That is,

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

When the client receives the answer $\tilde{v}$, she can check whether the obtained keyword from the answer is the same as the query keyword.

5.8.2 Synchronization Proxy

When the cost of the Synchronization phase is high compared to the total cost of all Query sessions, Synchronization can become a bottleneck for the client. For instance, in the DNS setting, where DNS databases get updated quickly, the client may need to re-Synchronize with the servers whenever a change occurs. Here, we observe that the Synchronization phase can be done by a proxy server for the client since Synchronization does not involve private information. In fact, the proxy server can Synchronize once for multiple clients as long as the clients query the same set of servers. We detail this technique below.

1. Proxy server $\mathcal{P}$ undergoes the Synchronization phase with SAPIR servers $S_0, \ldots, S_n$ and obtains catalog tags $M_0, \ldots, M_n$,
   S.
2. $\mathcal{P}$ compresses $M_{\mathsf{id}}, \forall \mathsf{id} \in \mathsf{AID}$:
$$\tilde{M}_{\mathsf{id}} \leftarrow \{\mathsf{index}(k, M_0) \mid k \in M_{\mathsf{id}}\}$$
3. SAPIR clients download $M_0, \tilde{M}_1, \ldots, \tilde{M}_n, S$ from $\mathcal{P}$ and decompress $\tilde{M}_{\mathsf{id}}, \forall \mathsf{id} \in \mathsf{AID}$:
$$M_{\mathsf{id}} \leftarrow \left\{k \in M_0 \mid \mathsf{index}(k, M_0) \in \tilde{M}_{\mathsf{id}}\right\}$$
4. The SAPIR clients independently continue the Setup and Query phase with $S_0, \ldots, S_n$.

Compared with the standard Synchronization phase, downloading the compressed tags $M_0, \tilde{M}_1, \ldots, \tilde{M}_n, S$ is far more efficient, especially because $\tilde{M}_1, \ldots, \tilde{M}_n$ are sets of indices. Moreover, when the number of clients is large, this technique can reduce the SAPIR server-side load in Synchronization and preprocessing databases for Query.

5.9 Implementation

SAPIR is implemented in Rust, where keyword compression in Section 5.8.1 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 to update the database online. 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].

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

6 Privacy-Preserving DNS

In this section, we demonstrate the application of SAPIR to achieve privacy-preserving DNS for NS (Name Server) records among DNS cache servers. The applicability of SAPIR is not limited to NS records, but NS records provide a workable 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 use as a reference to simulate data. (For other types of DNS records with short TTLs, we suggest the Synchronization Proxy technique in Section 5.8.2.)

We assume a hypothetical setting where three DNS cache servers keep a local cache table for NS records and build a SAPIR database out of this table. The keywords are the domain names, and the values are the NS records.

If multiple records are linked to the same domain name, then all the records are concatenated to provide one long database value. The cache table is assumed to be much smaller than the number of available records for efficiency. The SAPIR client chooses one DNS cache server as the main server and the other two as assisting servers. The collusion threshold is assumed to be $t = 2$.

To evaluate this application, we design the experiments on two scales: *local* and *regional*. The local experiment represents DNS servers whose cache tables result from local traffics in a city, e.g., Chicago. The regional experiment represents DNS servers whose cache tables result from regional traffics, e.g., the entire USA. The DNS popularity distribution is assumed to follow a Zifian distribution with an optimal popularity index of $s = 1.0$. The database values are assumed to be 1,024-byte long to contain multiple NS records per domain. The database size is $2^{13} \sim 8$K pairs for the local experiment and $2^{16} \sim 66$K pairs for the regional experiment. The details of how we justify these parameters and simulate the databases are in Appendix E.

We summarize all the experiment parameters in Table 1 and 2 below.

Parameters	Values
number of servers	3
collusion threshold, t	2
Zipfian popularity index, s	1.0
value length (bytes)	1,024
security parameter	128
SealPIR Parameters	
log(plaintext modulus), $d = 1$	14
log(plaintext modulus), $d = 2$	16

Table 1: *Shared parameters between local and regional experiments.* d indicates the query dimensionality of SealPIR.

Parameters	Local	Regional
$\lvert\mathsf{DB}\rvert$	$2^{13} \sim 8$K	$2^{16} \sim 66$K
hash length (bits)	43	54
Statistics		
% of sPIR	7.3	3.9
prob. of failure	$\lessapprox 1.3 \times 10^{-5}$	$\lessapprox 1.2 \times 10^{-5}$

Table 2: *Specific parameters and statistics for local and regional experiments (K=10^3).* "$\lvert\mathsf{DB}\rvert$" indicates the number of keyword-value pairs in each cache table. "% of sPIR" indicates the percentage of sPIR items in the main cache table, i.e., $|S| \,/\, |\mathsf{DB}_0| \times 100\%$. "hash length" indicates the length of hash functions $(\mathsf{H}, \tilde{\mathsf{H}}, \mathsf{G})$ in bits. "prob. of failure" indicates the probability of failure of the scheme for each unique query.

6.1 Evaluation Settings

To evaluate the application, we use the implementation specified in Section 5.9. 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 communication and computational costs for one 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 CPU time each party spends in milli-seconds.

The application is evaluated in two experiments: regional and local. In each experiment, a comparison is made between the following schemes:

1. Baseline SealPIR with query dimensionality $d = 2$, i.e., recursive PIR
2. SAPIR with $d = 1$ SealPIR
3. SAPIR 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. That is, the server computes the hashes of all keywords and sends them to the client as the catalog. The client translates keywords into indices by finding the index position of their hashes in the catalog.

In SAPIR, the keyword compression technique is applied per the description in Section 5.8.1.

SealPIR is configured by default to provide 128-bit security (see Section 5.9). The log of plaintext modulus in Table 1 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 in our trial runs to ensure the experiments do not fail from the noise after several runs.

The sPIR percentage statistics in Table 2 are calculated from the simulated databases. The probability of failure is calculated from the hash length and the size of the

Experiment	Phase	Communication	SealPIR, $d = 2$	SAPIR, $d = 1$		SAPIR, $d = 2$	
			$\mathcal{S}_0$	$\mathcal{S}_0$	$\mathcal{S}_1 + \mathcal{S}_2$	$\mathcal{S}_0$	$\mathcal{S}_1 + \mathcal{S}_2$
Local ($\|\mathsf{DB}\| = 2^{13}$)	**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 ($\|\mathsf{DB}\| = 2^{16}$)	**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 3: *Client's communication costs for one query in the local and regional experiment (in bytes, K=2^{10}, M=2^{20}).* "$\mathcal{S}_1 + \mathcal{S}_2$" indicates the client's combined communication costs with Server 1 and 2. The upload costs during Setup are listed separately between the SEAL public key and other public parameters.

Experiment	Phase	SealPIR, $d = 2$		SAPIR, $d = 1$			SAPIR, $d = 2$		
		$\mathcal{C}$	$\mathcal{S}_0$	$\mathcal{C}$	$\mathcal{S}_0$	$\mathcal{S}_1$ or $\mathcal{S}_2$	$\mathcal{C}$	$\mathcal{S}_0$	$\mathcal{S}_1$ or $\mathcal{S}_2$
Local ($\|\mathsf{DB}\| = 2^{13}$)	**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 ($\|\mathsf{DB}\| = 2^{16}$)	**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 4: *Computational costs in CPU time for one query in the local and regional experiment (in milli-seconds).* "$\mathcal{S}_0$" indicates the only server in SealPIR and the main server in SAPIR. "$\mathcal{S}_1$ or $\mathcal{S}_2$" indicates the computational cost on $\mathcal{S}_1$ or $\mathcal{S}_2$ in SAPIR.

union of all the simulated databases following Theorem 3 in Appendix D,

6.2 Results

The results for the communication costs of the experiments are shown in Table 3 and computational costs in Table 4. We refer to the Baseline $d = 2$ SealPIR scheme as the baseline scheme for comparison.

Communication costs During Synchronization and Setup, in the local experiment, the SAPIR client transmits 6.0x ((95K + 89K + 769 + 19K)/44K) the amount of data of the baseline client; in the regional experiment, the SAPIR client transmits 4.7x ((799K + 704K + 3.1K + 154K)/352K) the amount of data of the baseline client. This calculation does not include the cost of uploading the long-term SEAL public key (3.5M), which can be mitigated through public-key infrastructure.

During Query, in the local experiment, the $d = 1$ and $d = 2$ SAPIR client transmits 0.2x ((34K + 32K + 2K)/(64K + 257K)) and 1.0x ((65K + 257K + 2K)/(64K + 257K)), respectively, the amount of data per query of the baseline client; in the regional experiment, the $d = 1$ and $d = 2$ SAPIR client transmits 0.2x ((40K + 32K + 2K)/(64K + 257K)) and 1.0x ((72K + 257K + 2K)/(64K + 257K)), respectively, the amount of data per query of the baseline client. The latency between the client and Server

1 and 2 does not affect the online cost because the client can download replies from them offline before the query keyword is determined.

Overall, it would take $d = 1$ SAPIR client only one Query session to gain a communication cost advantage over the baseline client, whereas $d = 2$ SAPIR client will always cost more than the baseline client in communication.

Computational costs During Synchronization and Setup, our general observation is that it costs the SAPIR client more than the baseline client since the SAPIR client needs to compute catalog intersection and catalog tagging. However, since this one-time cost is less than 0.5 seconds, we consider this negligible. It costs the SAPIR main server less than the baseline server since the SAPIR main server needs to preprocess a smaller database for sPIR, but the overall cost is well below one second. The costs for the Server 1 and 2 are negligible (less than 60 milliseconds) compared to other parties.

During Query, the client's costs to generate a query and decode a reply are negligible (less than 20 milliseconds) across all the experiments. Server 1 and 2's cost to generate a reply is also negligible (less than 5 milliseconds). In the local experiment, it costs $d = 1$ and $d = 2$ SAPIR main server 0.85x (76/90) and 0.26x (23/90), respectively, CPU time of the baseline server per query. In the regional experiment, it costs $d = 1$ and $d = 2$ SAPIR main server 0.78x (323/414) and 0.12x (48/414), respectively, CPU of the baseline server per query.

6.3 Discussion

In comparison to the baseline SealPIR, SAPIR offers competitive options between reducing communication costs and computational costs. $d = 1$ SAPIR reduces a significant communication cost per query (0.2x) for the client and reduces a minor computational cost per query (0.85x for local and 0.78x for regional) for the main server. $d = 2$ SAPIR reduces no communication costs per query for the client but reduces a significant computational cost per query (0.26x for local and 0.12x for regional) for the main server. The cost of operating an assisting server is small, and introducing an extra assisting server only increases minimal offline costs for the client. This makes it feasible for each assisting server to serve a large number of users in the network.

Although our experiments are based on a specific Zipfian assumption of popularity distribution, we note that the statistics that determines the costs is the percentage of sPIR items in the main database (as shown in Table 2), which directly affects the main server costs. Different distributions might increase or decrease this percentage. When it is at 100%, the scheme naturally falls back to the baseline sPIR. Thus, we conclude that SAPIR always improves the cost per query over the baseline sPIR after the initial one-time cost.

7 Conclusion

In this work, we introduce Assisted PIR, a generalization to multi-server PIR that allows for keyword-value databases and database inconsistencies. We present the construction of Synchronized APIR, a hybrid APIR protocol combining a black-box single-server PIR scheme and a non-black-box multi-server PIR scheme, taking advantage of the overlaps between inconsistent databases to reduce costs. Since Synchronized APIR relies on a black-box sPIR scheme, advances in sPIR research also improve Synchronized APIR.

We apply Synchronized APIR to demonstrate a proof-of-concept privacy-preserving DNS 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 after the higher initial one-time cost, privacy-preserving DNS via Synchronized APIR outperforms the baseline single-server PIR in communication or computational costs.

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] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication–Computation trade-offs in PIR. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1811–1828. USENIX Association, August 2021.

[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] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-server private information retrieval with sublinear amortized time. In *Advances in Cryptology–EUROCRYPT 2022: 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30–June 3, 2022, Proceedings, Part II*, pages 3–33. Springer, 2022.

[9] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*, pages 44–75. Springer, 2020.

[10] Alex Davidson, Gonçalo Pestana, and Sofía Celi. Frodopir: Simple, scalable, single-server private information retrieval. Cryptology ePrint Archive, Paper 2022/981, 2022. `https://eprint.iacr.org/2022/981`.

[11] Casey Devet and Ian Goldberg. The best of both worlds: Combining information-theoretic and computational PIR for communication efficiency. In *Privacy Enhancing Technologies: 14th International Symposium, PETS 2014, Amsterdam, The Netherlands, July 16-18, 2014. Proceedings 14*, pages 63–82. Springer, 2014.

[12] Casey Devet, Ian Goldberg, and Nadia Heninger. Optimally robust private information retrieval. In *USENIX Security Symposium*, pages 269–283, 2012.

[13] 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.

[14] Ian Goldberg. Improving the robustness of private information retrieval. In *2007 IEEE Symposium on Security and Privacy (SP'07)*, pages 131–148. IEEE, 2007.

[15] Ryan Henry. Polynomial batch codes for efficient it-pir. *Proceedings on Privacy Enhancing Technologies*, 4:202–218, 2016.

[16] P. Hoffman and P. McManus. DNS Queries over HTTPS (DoH). RFC 8484, RFC Editor, October 2018.

[17] Z. Hu, L. Zhu, J. Heidemann, A. Mankin, D. Wessels, and P. Hoffman. Specification for DNS over Transport Layer Security (TLS). RFC 7858, RFC Editor, May 2016.

[18] 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.

[19] E. Kinnear, P. McManus, T. Pauly, T. Verma, and C.A. Wood. Oblivious dns over https. RFC 9230, RFC Editor, June 2022.

[20] 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.

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

[22] Paul Schmitt, Anne Edmundson, Allison Mankin, and Nick Feamster. Oblivious dns: Practical privacy for dns queries: Published in popets 2019. In *Proceedings of the Applied Networking Research Workshop*, ANRW '19, page 17–19, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Sudheesh Singanamalla, Suphanat Chunhapanya, Jonathan Hoyland, Marek Vavruša, Tanya Verma, Peter Wu, Marwan Fayed, Kurtis Heimerl, Nick Sullivan, and Christopher Wood. Oblivious DNS over HTTPS (ODoH): A Practical Privacy Enhancement

to DNS. *Proceedings on Privacy Enhancing Technologies*, 4:575–592, 2021.

[24] 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.

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

A sPIR Correctness and Privacy Definition

Definition A.1 (sPIR Correctness)**.** Following Definition 3.1 for scheme Π^{sPIR}. Suppose an indexed database $V = (v_0, \ldots, v_{m-1})$ for all $m \geq 1$, $i \in \{0, \ldots, m-1\}$, and

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

Then, Π^{sPIR} is correct if $\mathsf{sa} = v_i$ with a non-zero probability.

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 public-secret key pair is generated by $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \text{SGen}(1^\lambda)$. The public key spk is given to $\mathcal{A}$.
3. $\mathcal{A}$ is given oracle access to $\text{SQuery}(\mathsf{spk}, \cdot)$.
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 $\mathsf{sq}^* \leftarrow \text{SQuery}(\mathsf{spk}, i_b^*)$ is generated and sq^* is given to $\mathcal{A}$.
5. $\mathcal{A}$ is given more oracle access to $\text{SQuery}(\mathsf{spk}, \cdot)$.
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

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

B CGKS Tutorial

We provide a 3-server tutorial of CGKS [7] as follows. 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, the 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.

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.

C Concept of Synchronized APIR

Let us consider an example of keyword-value databases in Figure 5 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 5 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.

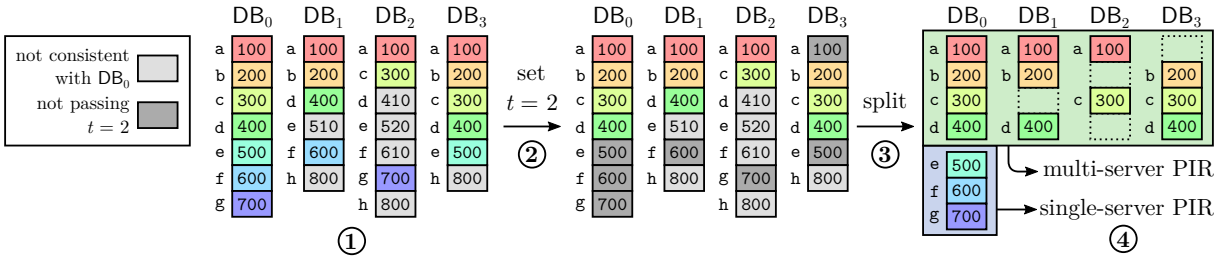

Figure 5: *Concept of Synchronized APIR.* A conceptual demonstration of how Synchronized APIR performs hybrid PIR on inconsistent databases when the collusion threshold is $t = 2$.

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 5 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 is explained in Section 5). 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. Suppose the client decides that $t = 2$; that is, no more than two servers can collude. Threshold t requires at least $t + 1$ copies 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 copies. (`(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 Section 5, we 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?

D Analysis

This section provides theorems 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.

D.1 Correctness

We will first prove the correctness of Synchronized APIR Theorem 1 by assuming that H, G, and $\tilde{\mathsf{H}}$ are perfect hash functions, i.e., they produce no hash collisions in the scheme. Then, we provide an analysis for the probability of no hash collisions if H, G, and $\tilde{\mathsf{H}}$ are, in fact, universal hash functions in Theorem 2. Since Synchronized APIR may fail only when there is a hash collision, the probability of no hash collisions implies an upper bound for the probability of failure of Synchronized APIR. We prove this in Theorem 3.

Theorem 1 (Synchronized APIR Correctness With Perfect Hashing)**.** Suppose an sPIR-correct scheme Π^{sPIR} and that the perfect hash functions H, G, and $\tilde{\mathsf{H}}$. Following Definition 4.2, Synchronized APIR scheme Π^{SAPIR} is correct for any $\mathsf{ID} = \{0, \ldots, n\}$ 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. First, by assuming the perfect hash functions, we can claim that

- $\tilde{\mathsf{H}}(k)$ perfectly represents keyword k, so k can be effectively 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 as 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

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

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

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

And therefore, by the equation above and property (3),

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

Next, because

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

by construction of MQuery (Algorithm 4), we have

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

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 $\mathsf{mr}^{\mathsf{sid}}_0$ by construction of Reply (Algorithm 6). By sPIR correctness of Π^{sPIR}, we conclude that in Decode (Algorithm 7), $\mathrm{SDecode}(\mathsf{ssk}, r^{\mathsf{sid}}) = \mathsf{mr}^{\mathsf{sid}}_0$. And finally,

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

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 $\mathsf{ID} = \{0, \ldots, n\}$ 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:

- $E_1(U) :=$

  $$\forall k, k' \in \mathsf{Keys}(U) : k \neq k' \implies \tilde{\mathsf{H}}(k) \neq \tilde{\mathsf{H}}(k')$$

  (For all database keywords, no two distinct keywords result in a hash collision.)

- $E_2(U) :=$

  $$\forall v, v' \in \mathsf{Values}(U) : v \neq v' \implies \mathsf{H}(v) \neq \mathsf{H}(v')$$

  (For all database values, no two distinct values result in a hash collision.)

- $E_3(U) :=$

  $$\forall (k, v), (k', v') \in U : (k, v) \neq (k', v') \implies \mathsf{G}((\tilde{\mathsf{H}}(k), \mathsf{H}(v))) \neq \mathsf{G}((\tilde{\mathsf{H}}(k'), \mathsf{H}(v')))$$

  (For all database keyword-value pairs, no two distinct pairs result in a digest collision.)

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

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

Proof. 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 [21], 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. That is,

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

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

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

□

Finally, we prove the upper bound of the probability of failure of Synchronized APIR with universal hash functions.

Theorem 3 (Synchronized APIR Correctness)**.** Suppose an sPIR-correct scheme Π^{sPIR} and $\mathsf{H}, \mathsf{G}, \tilde{\mathsf{H}}$ are random universal hash functions from universal families whose hash values are of length $l_{\mathsf{H}}, l_{\mathsf{G}}, l_{\tilde{\mathsf{H}}}$, respectively. Following Definition 4.2, for any $\mathsf{ID} = \{0, \ldots, n\}$ 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 $\mathsf{sid} \in [l]$, 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 $\mathsf{spk}, \mathsf{ssk}$, $\left(\mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}\right)_{\mathsf{sid} \in [l], \mathsf{id} \in \mathsf{AID}}, \left(\mathsf{tok}^{\mathsf{sid}}\right)_{[l]}$
- For all $\mathsf{sid} \in [l]$, Query protocol for session sid takes inputs $(\mathsf{DB}_{\mathsf{id}})_{\mathsf{ID}}, (M_{\mathsf{id}})_{\mathsf{ID}}, S, \left(\mathsf{mq}_{\mathsf{id}}^{\mathsf{sid}}\right)_{\mathsf{id} \in \mathsf{AID}}, \mathsf{spk}, \mathsf{ssk}, \mathsf{tok}^{\mathsf{sid}}, k^{\mathsf{sid}}$ and outputs a^{sid}

Then, for all $\mathsf{sid} \in [l], \mathsf{DB}_0[k^{\mathsf{sid}}] = a^{\mathsf{sid}}$, i.e. Synchronized APIR scheme Π^{SAPIR} is correct, with probability

$$\Pr[\Pi^{\mathsf{SAPIR}} \text{ is correct}] \gtrapprox p(|U|, 2^{l_{\mathsf{G}}}) \cdot p(|\mathsf{Keys}(U)|, 2^{l_{\tilde{\mathsf{H}}}}) \cdot p(|\mathsf{Values}(U)|, 2^{l_{\mathsf{H}}})$$

Likewise,

$$\Pr[\Pi^{\mathsf{SAPIR}} \text{ fails}] \lessapprox 1 - p(|U|, 2^{l_{\mathsf{G}}}) \cdot p(|\mathsf{Keys}(U)|, 2^{l_{\tilde{\mathsf{H}}}}) \cdot p(|\mathsf{Values}(U)|, 2^{l_{\mathsf{H}}})$$

where $U := \bigcup_{\mathsf{id} \in \mathsf{ID}} \mathsf{DB}_{\mathsf{id}}$ and $p(m, d) := e^{-\frac{m(m-1)}{2d}}$.

Proof. By Theorem 1 and Theorem 2, we know that if events $E_1(U), E_2(U), E_3(U)$ take place, then Theorem 3 is true with probability 1. That is,

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

Therefore,

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

And so,

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

□

We remark that the probability of failure 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 $k \neq k'$.

D.2 Privacy

To prove the privacy of Synchronized APIR, we first provide the privacy definition of Synchronized APIR in Definition D.1, 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 in Theorem 4. Leveraging this fact and the privacy of sPIR scheme, we show that Synchronized APIR is privacy-preserving in Theorem 5.

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 D.1 ((λ, t)-Synchronized APIR Privacy)**.** Following Definition 4.3, we define the privacy experiment $\mathsf{PrivA}_{\mathcal{A},\Pi^{\mathsf{SAPIR}},t}(1^\lambda)$ 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 $l = \mathrm{poly}(\lambda)$, correctly formatted 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 steps below are followed to generate public and secret parameters:

   (a) **Synchronization:** Step ② of Catalog Intersection Protocol in Section 5.6 is followed with inputs Cat_0, $(\mathsf{Dig}_{\mathsf{id}})_{\mathsf{AID}}$ to produce $\left(\mathsf{Cat}'_{\mathsf{id}}\right)_{\mathsf{AID}}$. Catalogs are tagged by

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

   as per step ② of Synchronization Protocol in Section 5.2.

   (b) **Setup:** Setup Protocol in Section 5.3 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

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

   following step ⑦ of Query Protocol in Section 5.5. $\left(\mathsf{mq}_0^{\mathsf{sid}}, \mathsf{sq}^{\mathsf{sid}}\right)$ 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 $\mathsf{sid} = \mathsf{sid}^* \leq l$,

   (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

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

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

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

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 4 (mPIR Query Pseudorandomness)**.** Suppose a variable-length PRG $\mathsf{PRG}(s, \ell) \to r$ where $|s| = \lambda$ and $|r| = \ell = \text{poly}(\lambda)$. For any $\mathsf{ID} := \{0, \ldots, n\}$, $\mathsf{AID} := \{1, \ldots, n\}$, $1 \le t \le n$, correctly formatted 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

- $\forall \mathsf{id} \in \mathsf{AID} \setminus A : s_{\mathsf{id}} \leftarrow_{\$} \{0,1\}^{\lambda}$
- $((M_{\mathsf{id}})_{\mathsf{AID}}, S) = \textsc{Tag}(\mathsf{Cat}_0, (\mathsf{Cat}'_{\mathsf{id}})_{\mathsf{AID}}, t)$
- $\forall \mathsf{id} \in \mathsf{AID} : (\mathsf{mq}^{\mathsf{sid}}_{\mathsf{id}})_{\mathsf{sid} \in [l]} = \mathsf{PRG}(s_{\mathsf{id}}, l \cdot |M_{\mathsf{id}}|)$
- $\forall \mathsf{sid} \in [l] : \mathsf{tok}^{\mathsf{sid}} = \textsc{MToken}\left((M_{\mathsf{id}})_{\mathsf{ID}}, (\mathsf{mq}^{\mathsf{sid}}_{\mathsf{id}})_{\mathsf{AID}}\right)$
- $\forall \mathsf{sid} \in [l] : \mathsf{mq}^{\mathsf{sid}}_0 = \textsc{MQuery}\left(M_0, \mathsf{tok}^{\mathsf{sid}}, k^{\mathsf{sid}}\right)$
- $\mathsf{aux} = \left((M_{\mathsf{id}})_{\mathsf{ID}}, A, (s_{\mathsf{id}})_A, (k^{\mathsf{sid}})_{[l]}\right)$

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

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

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 $\mathsf{mq}^{\mathsf{sid}}_{\mathsf{id}}$ above called $\tilde{\mathsf{mq}}^{\mathsf{sid}}_{\mathsf{id}}$ where if $\mathsf{id} \in \mathsf{AID} \setminus B$, then for all $\mathsf{sid} \in [l], \tilde{\mathsf{mq}}^{\mathsf{sid}}_{\mathsf{id}} \leftarrow_{\$} \{0,1\}^{|M_{\mathsf{id}}|}$ is sampled uniformly at random; everything else remains unchanged i.e. $\forall \mathsf{id} \in B, \forall \mathsf{sid} \in [l] : \tilde{\mathsf{mq}}^{\mathsf{sid}}_{\mathsf{id}} = \mathsf{mq}^{\mathsf{sid}}_{\mathsf{id}}$. This results in the new $\tilde{\mathsf{mq}}^{\mathsf{sid}}_0, \mathsf{sid} \in [l]$.

We claim that these two ensembles are perfectly indistinguishable

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

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

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

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

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

Since $|C_k| = t$ by construction and $|B| = t-1$, we know that $C_k \setminus B$ is not empty. Because for all $\mathsf{id} \in C_k \setminus B, \tilde{\mathsf{mq}}^{\mathsf{sid}}_{\mathsf{id}}$ is uniformly random by definition, we conclude that $\bigoplus_{\mathsf{id} \in C_k \setminus B} \tilde{\mathsf{mq}}^{\mathsf{sid}}_{\mathsf{id}}[\mathsf{index}(k, M_{\mathsf{id}})]$ 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 $\textsc{MQuery}$, $\tilde{\mathsf{mq}}^{\mathsf{sid}}_0$ 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

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

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

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

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

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

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

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

and inputs $((\mathsf{mq}^{\mathsf{sid},*}_0)_{[l]}, \mathsf{aux})$ 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,

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

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 5 (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^{SAPIR} is (λ, t)-APIR privacy-preserving according to Definition D.1.

Proof sketch. We first observe that if $0 \notin C$ in $\mathsf{PrivA}_{\mathcal{A},\Pi^{\mathsf{SAPIR}},t}$ 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 $\mathsf{PrivA}_{\mathcal{A},\Pi^{\mathsf{SAPIR}},t}$ is computationally indistinguishable between when $b = 0$ and $b = 1$. Formally, define the view of $\mathcal{A}$ in the experiment as

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

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

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

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 $\tilde{\mathsf{sq}} \leftarrow \mathrm{SQUERY}(\mathsf{spk}, 0)$, 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

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

2. By Theorem 4, we claim that

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

   because mPIR queries are pseudorandom.

3. Similarly to step 1, by λ-sPIR privacy assumption we claim that

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

4. By "transitivity", we claim that

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

   Thus proven the theorem.

□

E Data Simulation Method

Reference NS dataset To simulate cache tables for SAPIR servers, 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 `.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 5.

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 SAPIR database values must 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
number of records	382M
number of 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 5: *Statistics for NS records in ICANN's .com gTLD Zone File (M=10^6).* "|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.

Data simulation method Although we have access to NS records, 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 SAPIR. To simulate the servers' cache tables for this purpose, we thus 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 [18,25], with the total number of ranks being the total number of domain names with NS records (159 million domains according to Table 5).

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 SAPIR for the local and regional experiment: one main server 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 [25] and Jung *et al.* [18] 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 SAPIR'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 SAPIR 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 thus choose $s = 1.0$ to evaluate SAPIR.

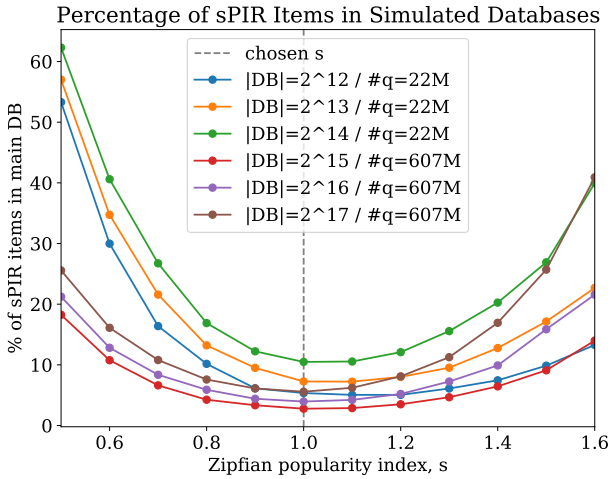

Figure 6: *The effect of Zipfian distribution on the percentage of sPIR items in simulated databases for 3 SAPIR servers at collusion threshold $t = 2$.* "#q" denotes the total number of simulated queries. $s = 1.0$ is chosen as an optimal parameter.