Encode and Permute that Database! Single-Server Private Information Retrieval with Constant Online Time, Communication, and Client-Side Storage

Shuaishuai Li*[1,2], Weiran Liu*[3], Liqiang Peng[3], Cong Zhang[1,2], Xinwei Gao[3], Aiping Liang[3], Lei Zhang[3], Dongdai Lin[1,2] and Yuan Hong[4]

[1] *State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China*
[2] *School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China*
{lishuaishuai,zhangcong,ddlin}@iie.ac.cn
[3] *Alibaba Group*
{weiran.lwr,plq270998,qiyuan.gxw,aiping.lap}@alibaba-inc.com, zongchao.zl@taobao.com
[4] *University of Connecticut*
yuan.hong@uconn.edu

Abstract

Private Information Retrieval (PIR) facilitates the retrieval of database entries by a client from a remote server without revealing which specific entry is being queried. The preprocessing model has emerged as a significant technique for constructing efficient PIR systems, allowing parties to execute a one-time, query-independent offline phase, and then a fast online retrieval phase. In particular, Corrigan-Gibbs and Kogan (EUROCRYPT 2020) presented a new framework for constructing PIR with sublinear online time. Nevertheless, their protocol is deemed impractical in the single-server setting due to the heavy use of Fully Homomorphic Encryption (FHE). More recently, two state-of-the-art (SOTA) single-server PIR protocols (Zhou et al., S&P 2024 and Mughees-Ren, ePrint 2023) have eliminated FHE, at the price of linear offline communication. However, the client-side storage is still relatively large ($\tilde{O}(\sqrt{n})$), which poses challenges to practical deployment, especially when the client has limited computation and storage capabilities. To address such limitation, we propose a novel PIR protocol Pai, which only requires constant online time, communication, and client-side storage. The price we pay is only a 1 - 5× increase in offline communication, which would be acceptable since it is a one-time cost. Building upon our Pai, we also present a Symmetric KPIR (KSPIR) PaiKSPIR and a Chargeable KSPIR (CKSPIR) PaiCKSPIR. These two variants of PIR offer enhanced functionalities while maintaining computational complexities similar to the original Pai.

In addition to providing rigorous theoretical proofs of correctness and privacy for Pai, we have undertaken comprehensive protocol implementations and conducted extensive experiments to validate their high efficiency. Our empirical findings demonstrate that our protocols achieve notably higher online efficiency than SOTA protocols, e.g., Pai exhibits 8.8 - 91.8× better online communication cost and 2.5 - 8.8× better online time. Given the superior online time and storage, our protocol is well-suited for practical deployment.

* The first two authors contribute equally.

1 Introduction

Private Information Retrieval (PIR) allows a client to obtain an entry from a public database hosted on a server without revealing which entry is retrieved. PIR has been one of the main research directions in cryptography since its inception by Chor et al. [16,17].[1] It has served as a key building block in many privacy-preserving applications, such as private contact tracing [51], safe browsing [32], and private blocklist lookups [32]. The naive PIR solution is to have the server send the whole database to the client for each query. However, this solution involves a high communication cost, making it less practical for databases (with large numbers of entries). A long line of work [3,4,7,10,12,14,21,23,27,28,30,31,33,37,39–41,43,45] has been devoted to designing PIR protocols with sublinear communication cost.

Although PIR has been studied for nearly three decades, its efficiency is still not satisfactory. One of the main reasons is that all existing PIR protocols require linear computation. This raises the question of designing PIR protocols with sublinear computation. Unfortunately, there is a fundamental barrier that the amount of server computation will inevitably be linear in the size of the database, formally proved by Beimel et al. [5]. Intuitively, the server must touch every single entry during some query; otherwise, the server learns that the queried entry is not one of the untouched entries.

Several directions have been explored to circumvent the fundamental barrier of linear computation. One approach is to amortize the cost over multiple queries, which assumes that the client submits a batch of queries at once. Several studies have presented efficient PIR protocols in this scenario [4,31,43]. However, the downside of these protocols is that they do not support adaptive queries, limiting its scope of application. Another approach for dealing with linear computation is PIR with Differential Privacy (DP) [2,50]. In DP-based PIR, the server may learn some information about

[1] The seminal work [16,17] considered PIR in the multi-server setting, where the client can interact with multiple non-concluding servers. In this work, we focus on the single-server setting.

the client's query location, which may cause security risks in practical applications. In this work, we are more concerned with PIR protocols that satisfy *standard security definition* while supporting adaptive queries.

Single-Server PIR with Preprocessing. Beimel et al. [5] proposed to use preprocessing to deal with the lower bound of linear computation. Concretely, the protocol is divided into two phases: an *offline phase* and an *online phase*. The offline phase is a one-time and query-independent process, and its one-time nature enables the cost to be amortized over multiple (adaptive) queries. Furthermore, the query-independent property allows the execution of the offline phase to occur before the client decides its queries. The main goal of PIR with preprocessing is to facilitate a fast online phase, with a specific focus on achieving sublinear online computation. To date, there are two classes of preprocessing models: *global preprocessing* and *client preprocessing*.

(1) PIR with Global Preprocessing. In the global preprocessing model, the server performs a global offline phase and computes an encoding of the database upfront for all clients. Doubly Efficient PIR (DEPIR) is defined as a PIR protocol in the global preprocessing model with sublinear online time. Two prior works of [11] and [9] took the first steps towards constructing DEPIR. However, the security of their protocols is based on a new non-standard computational hardness assumption from permuted Reed-Muller codes, which makes the security vulnerable to compromise [6,8] and their proposal purely theoretical in nature. The first DEPIR protocol based on standard computational assumption was given by Lin, Mook, and Wichs [36]. In their protocol, for a database of size n and any constant $\varepsilon > 0$, with $O(n^{1+\varepsilon})$ offline time and server storage, each online query can be done with $\mathsf{poly}(\log^{1/\varepsilon} n)$ communications and server computation. However, since their construction involves homomorphic evaluations of multivariate polynomials, the concrete performance is rather far from actual deployment. In fact, the recent work of [44] introduced several optimizations for the DEPIR protocol of [36] that improve both the asymptotic and concrete running times, as well as storage requirements, by orders of magnitude. In spite of this, the experimental results of [44] show that even for a very small database ($n \approx 2^{15.5}$), the server-side storage can be as large as 1TB on disk and 35GB on RAM, while the offline time takes 46.3 hours. Therefore, the DEPIR protocol of [36] is deemed impractical.

(2) PIR with Client Preprocessing. A more promising and concretely efficient preprocessing model is the client-preprocessing model (also called the subscription model or stateful model). In this model, the client downloads and stores "hints" from the server during the offline phase. Then, the client can efficiently execute many online queries by consuming these hints. Although the offline phase can be fairly expensive or may even require downloading the entire database, the amortized cost per query is extremely low, making the client-preprocessing model promising. The client-preprocessing model was initially introduced by Patel, Persiano, and Yeo [47], with a concrete scheme that still needs linear server computation per online query. Corrigan-Gibbs and Kogan [20] proposed the first client-preprocessing PIR scheme with amortized sublinear server computation. Follow-up works continue to make further improvements [19, 32, 34, 35, 49, 52]. Very recently, Zhou et al. [53] proposed an extremely simple efficient client-preprocessing PIR scheme Piano (short for Private Information Access NOw). The simplicity lies in that the construction is completely self-contained and does not invoke any existing PIR scheme as a building block. The efficiency lies in that the scheme only needs lightweight cryptographic primitives such as Pseudo-Random Functions (PRFs) (that can be accelerated with AES-NI instructions), with $\tilde{O}(\sqrt{n})$[2] client storage and $\tilde{O}(\sqrt{n})$ online communication (both the client request and the server response) and server computation per query. Mughees, I and Ren [42] further optimized the hint system and online query construction and proposed a simple and practical amortized sublinear PIR (for simplicity of notations, we name their scheme as Spam, short for Simple and Practical AMortized), achieving a server response with constant overhead. Notice that,

- *Lower Bound of Client-Preprocessing PIR*. Corrigan-Gibbs, Henzinger and Koga [19] showed a lower bound for client-preprocessing PIR. In particular, if the server stores the database in its original form and keeps no additional state, then the online client-side storage S and amortized online time T must satisfy that $S \cdot T = \tilde{\Omega}(n)$. In fact, Piano and Spam almost match this lower bound (up to poly-logarithmic factors). The theoretical works [35, 52] showed that the asymptotic amortized online communication cost can be further reduced to poly-logarithmic in n. But to date, Piano and Spam are concretely more efficient.

- *Circumventing the Lower Bound via Database Encoding*. The lower bound given by [19] implies that we cannot have $o(\sqrt{n})$ online communication and online client-side storage at the same time. However, in some applications, the client may have only a small amount of computation and storage resources in the online phase. Note that the lower bound of [19] is derived under the assumption that the database is stored in unencoded form. To circumvent this lower bound, we can use database encoding. In fact, DEPIR achieves $\tilde{O}(1)$ online communication and online client-side storage by letting the server generate and store an encoding of the database. However, as we have said, existing DEPIR protocols are far from being practical.

As a result, we ask the following question.

Can we construct a concretely efficient PIR protocol with $o(\sqrt{n})$ or even $\tilde{O}(1)$ online time, communication, and client-side storage in the preprocessing model?

[2]Throughout this work, we use $\tilde{O}(\cdot)$ to hide polylogarithmic terms, i.e., for any function $f(n)$, we have $\tilde{O}(f(n)) = O(f(n) \cdot \mathsf{poly}(\log n))$.

Table 1: **Asymptotic Comparison of Single-Server PIR Schemes.** n is the database size, $\varepsilon > 0$ is some suitable constant. "Offline Comm." means communication cost in preprocessing. "Online Comm." means communication per online query. "*" means the result is under simplified assumptions and additional cost will be incurred to address duplicate queries (assumptions are posited without loss of generality). See Section 2.4 for details.

Scheme	Offline Comm.	Offline Time	Online Comm.	Online Time	Online Client-Side Storage
Theoretical Single-Server PIR Schemes with Preprocessing					
Corrigan-Gibbs-Kogan [20]	$\tilde{O}(\sqrt{n})$	$\tilde{O}(n)$	$\tilde{O}(\sqrt{n})$	$\tilde{O}(\sqrt{n})$	$\tilde{O}(\sqrt{n})$
Corrigan-Gibbs-Henzinger-Kogan [19]	$\tilde{O}(\sqrt{n})$	$\tilde{O}(n)$	$\tilde{O}(\sqrt{n})$	$\tilde{O}(\sqrt{n})$	$\tilde{O}(\sqrt{n})$
Lazzaretti-Papamanthou [35]	$\tilde{O}(\sqrt{n})$	$\tilde{O}(n)$	$\tilde{O}(1)$	$\tilde{O}(\sqrt{n})$	$\tilde{O}(\sqrt{n})$
Zhou et al. [52]	$\tilde{O}(\sqrt{n})$	$\tilde{O}(n)$	$\tilde{O}(1)$	$\tilde{O}(\sqrt{n})$	$\tilde{O}(\sqrt{n})$
Lin-Mook-Wichs (DEPIR [36])	0	$O(n^{1+\varepsilon})$	$\mathsf{poly}(\log^{1/\varepsilon} n)$	$\mathsf{poly}(\log^{1/\varepsilon} n)$	$\tilde{O}(1)$
Practical Single-Server PIR Schemes with Preprocessing					
Zhou et al. (Piano [53])	$O(n)$	$O(n)$	$O(\sqrt{n})$	$\tilde{O}(\sqrt{n})$	$\tilde{O}(\sqrt{n})$
Mughee-I-Ren (Spam [42])	$O(n)$	$O(n)$	$O(\sqrt{n})$	$\tilde{O}(\sqrt{n})$	$\tilde{O}(\sqrt{n})$
Ours (Pai)	$\tilde{O}(n)$	$\tilde{O}(n)$	$\tilde{O}(1)$	$\tilde{O}(1)$	$\tilde{O}(1)$*

1.1 Contributions

We present a new single-server PIR protocol called Pai (as the conventional symbol name π for permutation) with many asymptotically near-optimal (optimal up to polylogarithmic factors) efficiency features.

Table 1 summarizes the asymptotic complexity of PIR protocols in the preprocessing model. Compared to the SOTA PIR schemes with preprocessing (Piano and Spam), Pai significantly reduces the online costs (i.e., communication, computation, and client-side storage) from sublinear to $\tilde{O}(1)$.

PIR with Significantly Reduced Online Costs. Pai circumvents the lower bound by encoding the database *with the help of the client*, i.e., encoding by encryption and permutation (that is why we name it Pai), at the price of more offline communication cost. Similar to Piano [53] and Spam [42], the offline communication cost of Pai is still $\tilde{O}(n)$. Considering the offline phase in Pai is one-time, such offline communication can be amortized when the client has multiple queries. As in Piano and Spam, when amortized over $\tilde{O}(\sqrt{n})$ queries, the amortized offline communication for each query is merely $\tilde{O}(\sqrt{n})$. Another rationale for accepting $\tilde{O}(n)$ linear offline communication is that, in many scenarios, the paramount requirement for the client is *fast online query response*. Given the capability to execute the offline phase prior to query submission, a slightly more expensive offline phase would be acceptable for a more efficient online phase.

In Piano and Spam, the offline communication is linear (the client downloads the whole database from the server). This implies that the client must have $O(n)$ transient storage before submitting its query. To deal with this, they use a streaming algorithm to achieve $O(\sqrt{n})$ bandwidth in the offline phase, which reduces the transient client-side storage to $O(\sqrt{n})$. Similar to their work, Pai also uses a streaming algorithm to achieve $\tilde{O}(\sqrt{n})$ bandwidth in the offline phase.

Efficiently Supporting PIR Variants. Pai can be extended to support other PIR variants with high efficiency, some of which were briefly discussed in Piano [53]. Specifically,

(1) Keyword Symmetric PIR (KSPIR). Many variants of PIR have also been studied since its initial proposal. For example, Chor et al. [15] considered Keyword PIR (KPIR), where the database is defined as a set of n key-value pairs and the client uses a search key to retrieve the corresponding value. Gertner et al. [29] introduced Symmetric PIR (SPIR), which additionally requires that the client can only know its desired entry. In this work, we consider both of these two extensions, i.e., Keyword SPIR (KSPIR), also known as Labeled Private Set Intersection (LPSI) [13, 18].

There exist general conversion methods from PIR to KSPIR. For example, Ali et al. [3] introduced a method to convert PIR to KPIR, and Freedman et al. [26] presented a method to convert KPIR to KSPIR. Combining these two ideas, we can derive a transformation from PIR to KSPIR. However, to our best knowledge, for the most efficient PIR protocols in the preprocessing model (e.g., Piano and Spam), there is no implementation to verify the efficiency of the KSPIR protocol resulting from the conversions.

We apply the transformations of [3, 26] to Pai, Piano, and Spam, respectively. Then, we obtain three new KSPIR protocols, denoted as PaiKSPIR, PianoKSPIR, and SpamKSPIR. To verify the efficiency of this transformation, we have implemented all of these three KSPIR protocols, and compared them with the state-of-the-art KSPIR protocol without preprocessing by Cong et al. [18]. Our results concretely show that KSPIR can also enjoy fast online query response in the preprocessing model.

(2) Chargeable KSPIR (CKSPIR). We also explore an alternative variant of KSPIR, which arises from a specific application context where the database contains valuable items (e.g., copyrighted music or movies), and the server seeks *payment*

from the client for retrieval. Note that in KSPIR, the client may fail to retrieve a value (if its search key is not in the key set of the database). In this case, it is unreasonable to charge the client for the failed retrieval attempt.

To address this issue, we let the server know whether the client successfully obtains a value and charge if and only if the retrieval is successful. We introduce CKSPIR as a novel framework to model and address this particular variant of KSPIR. We note that any KSPIR (e.g., [13, 18]) can be converted to CKSPIR by letting the client send a bit indicating whether it successfully retrieves a value with its key. Based on the idea of Pai, we construct a more efficient CKSPIR protocol PaiCKSPIR by inherently allowing the server to know whether the retrieved key matches some key in the database.

Open-Source Implementations and Evaluations. We implemented Pai and other PIR variants in Java. The source code is available at https://github.com/alibaba-edu/mpc4j. It is our intention to open-source the code after publication. We conduct experiments with variants of database sizes $n = 2^{20}, 2^{22}, 2^{24}$ and entry sizes $64, 128, 256$. Pai and its variants enjoy good concrete efficiency, e.g., 8.8 - 91.8$\times$ better online communication cost and 2.5 - 8.8$\times$ better online time than SOTA PIR protocols (Paino and Spam). Our KSPIR protocol with preprocessing also enjoys extremely fast online queries, at least three orders of magnitude improvement compared with the SOTA KSPIR without preprocessing [18].

2 Preliminaries

In this section, we introduce some preliminaries related to this work. Some notations are first defined as below.

Math Notations. Let λ be the computational security parameter and κ be the statistical security parameter. For any integer n, we use $[n]$ to represent the set $\{1, \cdots, n\}$. For any two distributions $\mathsf{D}_0, \mathsf{D}_1$, if no probabilistic polynomial time (PPT) algorithm can distinguish them, then we say that D_0 and D_1 are computationally indistinguishable, denoted $\mathsf{D}_0 \approx_c \mathsf{D}_1$.

Protocol Notations. All of our protocols proceed between a server and a client. In PIR, we use $\mathsf{DB} = (v_1, \ldots, v_n)$ to denote the database. In KSPIR and CKSPIR, the database DB is a set of n key-value pairs, i.e., $\mathsf{DB} = \{(k_i, v_i)\}_{i \in [n]}$. Let l_k and l_v denote the length of k_i and v_i, respectively. Namely, we assume each k_i belongs to $\mathcal{K} = \{0,1\}^{l_k}$ and each v_i belongs to $\mathcal{V} = \{0,1\}^{l_v}$. Finally, we assume that the keys in the database are distinct, which implies that $l_k \geq \log n$.

2.1 Adversarial Model

In this work, we consider a semi-honest (or passive) adversary. That is, the parties will not deviate from the protocol and will not collude with each other. The adversary attacks the protocol by corrupting one party and using its internal state to infer information about the inputs of the other party.

2.2 Definitions of PIR and Its Variants

In this section, we present the formal definitions for PIR, KSPIR, and CKSPIR. For the sake of simplicity, we present the definitions in the single-query setting, where the client only has a single query. When the client has multiple queries, the corresponding definitions can be obtained naturally.

Definition 1 (PIR). *In PIR, the server takes a database* $\mathsf{DB} = (v_1, \ldots, v_n)$ *as input, and the client takes an index* $i \in [n]$ *as input. PIR has the following properties.*

- ***Correctness.*** *At the end, the client outputs* v_i.
- ***Client-Privacy.*** *The server knows nothing about i. Let* $\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, i)$ *be the view of the server in the protocol. There exists a PPT algorithm* Sim *taking* DB *as input such that*
$$\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, i) \approx_c \mathsf{Sim}(\mathsf{DB}).$$

Definition 2 (KSPIR). *In KSPIR, the server takes a database* $\mathsf{DB} = \{(k_i, v_i)\}_{i \in [n]}$ *as input, and the client takes a search key k as input. KSPIR has the following properties.*

- ***Correctness.*** *At the end, the client outputs* $z = v_i$ *if there exists some* $i \in [n]$ *such that* $k_i = k$*; otherwise, the client outputs* $z = \perp$.
- ***Server-Privacy.*** *The client knows nothing about* DB *except information contained in the key k it requests and the entry z it obtains. Namely, let* $\mathsf{View}_{\mathsf{cli}}(\mathsf{DB}, k)$ *be the view of the client in the protocol. There exists a PPT algorithm* Sim *taking* k, z *as inputs such that*
$$\mathsf{View}_{\mathsf{cli}}(\mathsf{DB}, k) \approx_c \mathsf{Sim}(k, z).$$
- ***Client-Privacy.*** *The server knows nothing about k. Let* $\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, k)$ *be the view of the server in the protocol. There exists a PPT algorithm* Sim *taking* DB *as input such that*
$$\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, k) \approx_c \mathsf{Sim}(\mathsf{DB}).$$

Compared to KSPIR, CKSPIR additionally outputs a bit to the server, which makes the server know whether the client successfully obtained a value. This enables important real-world applications, where the server has a database containing valuable items and seeks payment from the client for retrieval.

Definition 3 (CKSPIR). *In CKSPIR, the server takes a database* $\mathsf{DB} = \{(k_i, v_i)\}_{i \in [n]}$ *as input, and the client takes a search key k as input. CKSPIR has the following properties.*

- ***Correctness.*** *If there exists some* $i \in [n]$ *such that* $k_i = k$, *then the server outputs* $b = 1$, *and the client outputs* $z = v_i$. *Otherwise, the server outputs* $b = 0$, *and the client outputs* $z = \perp$.

- ***Server-Privacy.*** *The client knows nothing about* DB *except the information contained in* k, z. *Let* $\mathsf{View}_{\mathsf{cli}}(\mathsf{DB}, k)$ *be the view of the client in the protocol. Then there exists a PPT algorithm* Sim *taking* k, z *as inputs such that*

$$\mathsf{View}_{\mathsf{cli}}(\mathsf{DB}, k) \approx_c \mathsf{Sim}(k, z).$$

- ***Client-Privacy.*** *The server knows nothing about k except the information contained in b. Let* $\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, k)$ *be the view of the server in the protocol. There exists a PPT algorithm* Sim *taking* DB, b *as inputs such that*

$$\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, k) \approx_c \mathsf{Sim}(\mathsf{DB}, b).$$

2.3 Oblivious Pseudo-Random Function

Oblivious Pseudo-Random Function (OPRF) is a two-party protocol between a server and a client, where the server has a master key mk for a PRF F, and the client holds an input x. The protocol requires that the server obtains nothing, and the client obtains the value $\mathsf{F}(mk, x)$ and nothing else. In this work, we will use the OPRF protocol of [22] that defines $\mathsf{F}(mk, x) = \mathsf{H}(x)^{mk}$, where H is a hash function modeled as a random oracle. This protocol is secure under the Decisional Diffie-Hellman (DDH) assumption against semi-honest adversaries. We recall this protocol in the following.

Public Parameters. Let $\mathbb{G}$ be a DDH-hard cyclic group with prime order p, and $\mathsf{H} : \{0,1\}^* \to \mathbb{G}$ a hash function modeled as a random oracle.

Input. The server has a master key $mk \in \mathbb{Z}_p^*$, and the client has an input $x \in \{0,1\}^*$.

1. The client first samples $\beta \in \mathbb{Z}_p^*$ and sends $\mathsf{H}(x)^\beta$ to the server, who responds with $(\mathsf{H}(x)^\beta)^{mk}$.
2. The client outputs $\mathsf{H}(x)^{mk} = ((\mathsf{H}(x)^\beta)^{mk})^{1/\beta}$.

2.4 Simplifying PIR Assumptions

We introduce several simplifying assumptions pertaining to client-preprocessing PIR. Some of these assumptions are needed in Piano, Spam, and our Pai. As shown in [53], we can make all these assumptions *without loss of generality*.

- **Handling Duplicate Queries.** Piano, Spam, and our Pai assume the client does not make any duplicate queries. Piano and Spam leverage this assumption to prevent unnecessary "hint" consumption, while Pai is to prevent the client from generating identical requests. This assumption is made without loss of generality by asking the client to save the retrieved indices and their answers. Whenever the client makes a duplicate query, it loops up the answer locally and make a non-duplicated dummy query instead. This method only introduces $\tilde{O}(\sqrt{n})$ client storage cost *in practice*, as long as the total number of queries is $\tilde{O}(\sqrt{n})$.
- **Supporting Unbounded Queries from Bounded $\tilde{O}(\sqrt{n})$ Queries.** Piano, Spam and our Pai assume the client makes $\tilde{O}(\sqrt{n})$ queries. Piano and Spam leverage this assumption to prepare $\tilde{O}(\sqrt{n})$ "hints" in preprocessing, and consume them during the online phase. Pai only requires this assumption *in practice* to make the client storage $\tilde{O}(\sqrt{n})$ for address the duplicate queries. We can support unbounded queries simply by rerunning the preprocessing every $\tilde{O}(\sqrt{n})$ queries, and amortizing the periodic preprocessing cost.
- **Supporting Target Queries from Random Queries.** Piano assumes the client queries indices that are sampled at random without replacement. Otherwise, parts of "hints" will be consumed quickly. This assumption is without loss of generality, by letting the server randomly permute the database upfront via a Pseudo-Random Permutation (PRP) with a random key (independent of the client queries) and then sharing the key to the client. Spam and Pai do not require this assumption.

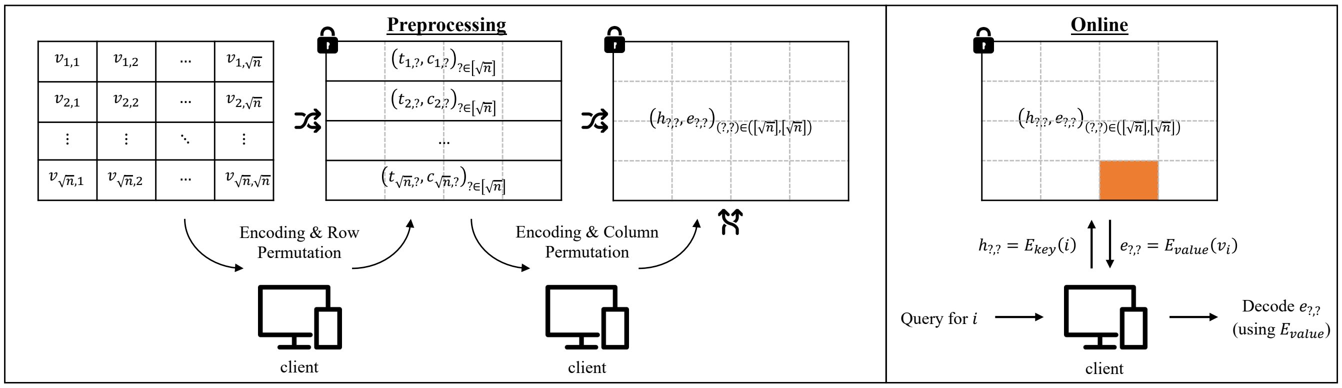
3 Pai Framework

The two main PIR frameworks with sublinear online time are the PIR construction of [19, 20, 42, 52, 53] and the DEPIR protocol of [36]. The key idea of the first framework is that the client interacts with the server to generates $\tilde{O}(\sqrt{n})$ query-independent hints in the offline phase, and then consumes those hints to achieve concretely efficient online phase. DEPIR avoids hint storage costs by utilizing (global) non-interactive preprocessing, where the server encodes the database in the offline phase without interacting with the client. However, current SOTA DEPIR protocol [36] has been experimentally verified to be impractical due to considerable storage and preprocessing time [44].

The main idea for devising Pai is to harness the capabilities of both interaction and database encoding. Concretely, we involve the client in assisting the server with database encoding during the offline phase. We now give a technical overview of Pai, and then briefly describe how to derive our CKSPIR and KSPIR protocols, respectively.

Overview of Pai. Figure 1 illustrates the framework of Pai (both offline and online phases). Similar to other client-preprocessing PIRs, the design of Pai focuses on the offline phase (preprocessing), including three key steps as below.

- *Step 1: Encoding and Permuting the Database.* Two random encoding algorithms are used to encode the indices and entries, respectively. The database is simultaneously randomly permuted, making the encoded database pseudorandom from the server's view while allowing the client to retrieve and decode in the online phase.
- *Step 2: Encoding the Database with the Client.* To avoid heavy encoding procedures as in DEPIR, we introduce



Preprocessing
Online
Encoding & Row Permutation
Encoding & Column Permutation
client
client
client
Query for i
Decode $e_{?,?}$ (using E_{value})
$h_{?,?} = E_{key}(i)$
$e_{?,?} = E_{value}(v_i)$

Figure 1: The Pai framework. Three key steps are embedded in the encoding & row permutation and encoding & column permutation that are jointly performed by the server and client.

offline interaction so that the database encoding and permutation are jointly executed by the server and client.

- *Step 3: Reducing the Bandwidth via a Streaming Algorithm*. To achieve a low bandwidth in the offline interaction, the database is represented in two-dimensions so that the rows and columns of the database are encoded and permuted respectively. As will be subsequently demonstrated, this approach reduces the offline bandwidth to $\tilde{O}(\sqrt{n})$, same as in Piano and Spam.

Finally, during the online phase, the client sends the encoded index to the server. The server finds and returns the matching encoded value, which can be efficiently decoded and by the client to derive the true result.

From PIR to KSPIR. We introduce cuckoo hashing table [3] and OPRF [26] to concretely convert our Pai (as well as Piano and Spam) to corresponding KSPIR. More details are deferred to Section 5.

From PIR to CKSPIR. We can upgrade our PIR protocol to obtain a more efficient CKSPIR. Due to limit of space, we defer our CKSPIR to Appendix A, which also contains the corresponding security proof, complexity analysis, and implementation.

4 Pai Construction and Analysis

In this section, we are poised to delineate our PIR protocol Pai. Furthermore, we provide formal security proof and complexity analysis for Pai.

4.1 Pai Construction

Pai is designed in the client-preprocessing model and consists of an offline phase and an online phase. In the offline phase, the client helps the server encode the database to enable a very fast online phase. Here we demonstrate how to design the aforementioned three key steps in the offline phase. The detailed construction is shown in Figure 2.

Step 1: Encoding and Permuting the Database. The foundational concept behind our database encoding approach involves both permutation and encryption of the database. To be precise, given a database denoted as $v_{j\,j\in[n]}$, the encoding procedure is articulated as follows.

1. Choose a random permutation π over $[n]$.
2. Choose two encoding algorithms $\mathsf{E}_{\mathsf{index}}$ and $\mathsf{E}_{\mathsf{entry}}$, which are used to encode the indices and entries, respectively.
3. Then, the encoded database is $\{(h_j, e_j)\}_{j\in[n]}$, where $h_j = \mathsf{E}_{\mathsf{index}}(\pi(j)), e_j = \mathsf{E}_{\mathsf{entry}}(v_{\pi(j)})$.

The main goal of the above encoding is to enable "secure" retrieval against the original database by using "insecure" retrieval against the securely encoded (key-value) database. Specifically, we aim to design the following online phase.

1. When the client decides its search index i, it computes $h = \mathsf{E}_{\mathsf{index}}(i)$ as the query message.
2. The server finds $j \in [n]$ such that $h = h_j$ and returns $e = e_j$ to the client as the response.
3. The client decodes e to v and outputs v.

To ensure the correctness of the protocol, it is imperative that the client produces an output $v = v_i$. Two primary conditions must be satisfied: (1) the response message is $e_{\pi^{-1}(i)}$ (which is the encoding of v_i), and (2) extracting v_i from $e_{\pi^{-1}(i)}$ is easy. To meet this, the encoding functions are subjected to the following requirements.

- $\mathsf{E}_{\mathsf{index}}$ is deterministic and collision-resistant.
- $\mathsf{E}_{\mathsf{entry}}$ is invertible. That is, the client can easily inverse v from $e = \mathsf{E}_{\mathsf{entry}}(v)$.

Figure 2: The construction of Pai.

Protocol Π_{Pai}: PIR with $\tilde{O}(1)$ Client-Side Storage and Online Time

Input: The server $\mathcal{S}$ has a database $\mathsf{DB} = (v_1, \ldots, v_n)$, and the client $\mathcal{C}$ has an index $i \in [n]$.
Output: $\mathcal{C}$ outputs v_i.

Offline Phase: $\mathcal{S}$ represents the database as a matrix $(v_{j_0,j_1})_{j_0,j_1 \in [\sqrt{n}]}$, where $v_{j_0,j_1} = v_{(j_0-1)\sqrt{n}+j_1}$. $\mathcal{S}$ and $\mathcal{C}$ agree on an IND-CPA-secure SKE scheme $(\mathsf{SKE.Gen}, \mathsf{SKE.Enc}, \mathsf{SKE.Dec})$ and a PRP $\mathsf{P} : \{0,1\}^\lambda \times \{0,1\}^l \to \{0,1\}^l$ with $l \geq \lceil \log n \rceil$. Then, $\mathcal{C}$ chooses two random permutations π_0, π_1 over $[\sqrt{n}]$, an SKE key ek, and two PRP keys pk_0, pk_1. We note that π_0 is used for permuting the rows, and π_1 is used for permuting the columns. $\mathcal{S}$ and $\mathcal{C}$ interact to encode the database in a streaming way.

1. Row-Permuting. For each $j_1 \in [\sqrt{n}]$, $\mathcal{S}$ and $\mathcal{C}$ perform the following steps.
   (a) $\mathcal{S}$ sends $\{v_{j_0,j_1}\}_{j_0 \in [\sqrt{n}]}$ to $\mathcal{C}$.
   (b) For all $j_0 \in [\sqrt{n}]$, $\mathcal{C}$ sets $j = (\pi_0(j_0) - 1)\sqrt{n} + j_1$, and then $\mathcal{C}$ computes $t_{j_0,j_1} = \mathsf{P}(pk_0, j)$ and $c_{j_0,j_1} = \mathsf{SKE.Enc}(ek, v_j)$. Then, $\mathcal{C}$ sends $\{(t_{j_0,j_1}, c_{j_0,j_1})\}_{j_0 \in [\sqrt{n}]}$ to $\mathcal{S}$.
   (c) Finally, $\mathcal{C}$ deletes $\{v_{j_0,j_1}, (t_{j_0,j_1}, c_{j_0,j_1})\}_{j_0 \in [\sqrt{n}]}$ from its local storage.
2. Column-Permuting. For each $j_0 \in [\sqrt{n}]$, $\mathcal{S}$ and $\mathcal{C}$ perform the following steps.
   (a) $\mathcal{S}$ sends $\{(t_{j_0,j_1}, c_{j_0,j_1})\}_{j_1 \in [\sqrt{n}]}$ to $\mathcal{C}$.
   (b) For all $j_1 \in [\sqrt{n}]$, $\mathcal{C}$ computes $h_{j_0,j_1} = \mathsf{P}(pk_1, t_{j_0,\pi_1(j_1)})$, and $e_{j_0,j_1} = \mathsf{SKE.Enc}(ek, \mathsf{SKE.Dec}(ek, c_{j_0,\pi_1(j_1)}))$. Then, $\mathcal{C}$ sends $\{(h_{j_0,j_1}, e_{j_0,j_1})\}_{j_1 \in [\sqrt{n}]}$ to $\mathcal{S}$.
   (c) Finally, $\mathcal{C}$ deletes $\{(t_{j_0,j_1}, c_{j_0,j_1}), (h_{j_0,j_1}, e_{j_0,j_1})\}_{j_0 \in [\sqrt{n}]}$ from its local storage.

Online Phase: To query an index $i \in [n]$, $\mathcal{S}$ and $\mathcal{C}$ proceed as follows.

1. Query. $\mathcal{C}$ computes $h = \mathsf{P}(pk_1, \mathsf{P}(pk_0, i))$ and sends h to $\mathcal{S}$.
2. Response. $\mathcal{S}$ finds h_{j_0,j_1} such that $h_{j_0,j_1} = h$. Then, it sends $e = e_{j_0,j_1}$ to $\mathcal{C}$.
3. Extract. $\mathcal{C}$ decrypts and outputs $z = \mathsf{SKE.Dec}(ek, e)$.

For the client-privacy guarantee, recall that the query message is $h = h_j = \mathsf{E}_{\mathsf{index}}(i)$ with $j = \pi^{-1}(i)$, and the response message is $e_j = e_{\pi^{-1}(i)}$. We need to guarantee that the server obtains nothing about i by seeing j, h_j and e_j with the knowledge of all v_i. We have the following requirements.

- π^{-1} should be a random permutation that is unknown to the server.
- $\mathsf{E}_{\mathsf{index}}$ is pseudorandom to the server, which guarantees that h_j leaks nothing about i.
- $\mathsf{E}_{\mathsf{entry}}$ is pseudorandom to the server, which guarantees that e_j leaks nothing about v_i.

For the deterministic, collision-resistant, and pseudorandom encoding function $\mathsf{E}_{\mathsf{index}}$, we choose a Pseudo-Random Permutation (PRP) (e.g., AES) with the client holding the PRP key. For the invertible and pseudorandom encoding function $\mathsf{E}_{\mathsf{entry}}$, we instantiate it with an IND-CPA secure SKE scheme with the client holding the SKE key.

Step 2: Encoding the Database with the Client. We now discuss how to encode the database with the client holding the corresponding keys. While it is feasible for the client to initially transmit the encrypted keys and subsequently permit the server to employ FHE for computing the encoded database, this would lead to a rather high computational overhead.

To make our offline phase practical, we follow the basic idea of Piano and Spam and let the client download the entire database and encode it. Specifically, the client chooses a PRP key pk and an encryption key ek of an IND-CPA secure SKE scheme. Moreover, the client chooses a random permutation π over $[n]$. Then, the client encrypts the database as $\{(h_j, e_j)\}_{j \in [n]}$, where $h_j = \mathsf{P}(\pi(j)), e_j = \mathsf{SKE.Enc}(ek, v_{\pi(j)})$. Finally, the client sends $\{(h_j, e_j)\}_{j \in [n]}$ to the server. Compared with Piano and Spam, the clients further needs to upload the encoded database, which incur some extra offline communication.

Step 3: Reducing the Bandwidth via a Streaming Algorithm. The basic idea involves bandwidth $\tilde{O}(n)$ during the offline phase. This makes it necessary for the client to have $\tilde{O}(n)$ transient storage (in the offline phase). To solve a similar problem, Piano [53] and Spam [42] leverage a streaming algorithm to reduce the bandwidth to $O(\sqrt{n})$, which significantly reduces the amount of storage required by the client in the offline phase. Their idea is that the server sends $O(\sqrt{n})$ entries each time, and the client processes and then deletes those entries from its storage. However, this idea is not applicable to our protocol since we need to permute the entire database. If the client downloads $O(\sqrt{n})$ entries each time and uploads the processed entries, then the server knows that the encoded entries it receives are the permutation of the entries that the client downloaded previously, which leaks information about the permutation. To reduce the bandwidth of Pai, we instead use the folding technique of [33], which represents the database as a 2-dimensional hypercube. Then, the parties permute the rows and columns of the database, respectively. As we will see, this allows us to reduce the offline bandwidth to be $\tilde{O}(\sqrt{n})$. By representing the database as a hypercube with a higher dimension d, we can further reduce the bandwidth to $\tilde{O}(n^{1/d})$. However, the offline communication of our protocol will also increase by a factor of $O(d)$. To make a fair comparison with Piano and Spam, we set $d = 2$ in both the description and implementation of our protocol. In practice, one can obtain the required tradeoff between offline bandwidth and offline communication by using different d.

The above three steps lead to an extremely simple online phase in Pai. Assume that the client has an index i, the client first computes and sends $h = \mathsf{P}(i)$ to the server. The server finds $j \in [n]$ such that $h = h_j$ and returns $e = e_j$ to the client, where $j = \pi^{-1}(i)$. By decrypting e_j using the SKE key ek, the client obtains the value $v_{\pi(j)} = v_i$.

4.2 Security Analysis for Pai

We show the security of our protocol by proving the following theorem.

Theorem 1. Π_{Pai} *is a secure PIR protocol.*

Proof. We show the correctness and client-privacy of Π_{Pai}.

Correctness. To show correctness, we need to prove that the client's output is v_i. Assume that $i_0, i_1 \in [\sqrt{n}]$ satisfy that $i = (i_0 - 1)\sqrt{n} + i_1$, then we have

$$\begin{aligned} h =& \mathsf{P}(pk_1, \mathsf{P}(pk_0, i)) = \mathsf{P}(pk_1, \mathsf{P}(pk_0, (i_0 - 1)\sqrt{n} + i_1)) \\ =& \mathsf{P}(pk_1, t_{\pi_0^{-1}(i_0), i_1}) = h_{\pi_0^{-1}(i_0), \pi_1^{-1}(i_1)}. \end{aligned}$$

This means that $e = e_{\pi_0^{-1}(i_0), \pi_1^{-1}(i_1)}$. Following the procedures in the offline phase, we know that for any $j_0, j_1 \in [\sqrt{n}]$, e_{j_0,j_1} is an encryption of $v_{\pi_0(j_0),\pi_1(j_1)}$, which implies that e is an encryption of $v_{i_0,i_1} = v_i$. Note that the output of the client is the decryption of e. Therefore, the output of the client is v_i.

Client-Privacy. Let $\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, i)$ be the view of the server. We need to construct a PPT simulator Sim taking DB as input such that

$$\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, i) \approx_c \mathsf{Sim}(\mathsf{DB}).$$

Note that $\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, i)$ consists of

$$(\mathsf{DB}, \{(t_{j_0,j_1}, c_{j_0,j_1})\}_{j_0,j_1 \in [\sqrt{n}]}, \{(h_{j_0,j_1}, e_{j_0,j_1})\}_{j_0,j_1 \in [\sqrt{n}]}, h),$$

where $h = \mathsf{P}(pk_1, \mathsf{P}(pk_0, i))$. To simulate this view, Sim proceeds as follows.

1. Sample two sets $\{t^*_{j_0,j_1}\}_{j_0,j_1 \in [\sqrt{n}]}$, $\{h^*_{j_0,j_1}\}_{j_0,j_1 \in [\sqrt{n}]}$ of elements from $\{0,1\}^l$, where each set contains n distinct and random elements.
2. Sample $2n$ ciphertexts $\{c^*_{j_0,j_1}, e^*_{j_0,j_1}\}_{j_0,j_1 \in [\sqrt{n}]}$ of zero.
3. Pick two random elements $j_0^*, j_1^* \in [\sqrt{n}]$ and sets
$$h^* = h^*_{j_0^*, j_1^*}.$$
4. Output the simulated view
$$\begin{aligned} (\mathsf{DB}, \{(t^*_{j_0,j_1}, c^*_{j_0,j_1})\}_{j_0,j_1 \in [\sqrt{n}]}, \\ \{(h^*_{j_0,j_1}, e^*_{j_0,j_1})\}_{j_0,j_1 \in [\sqrt{n}]}, h^*). \end{aligned}$$

Now, we show that the real and simulated views are indistinguishable. Firstly, since P is a PRP over $\{0,1\}^l$, both $\{t_{j_0,j_1}\}_{j_0,j_1 \in [\sqrt{n}]}$ and $\{h_{j_0,j_1}\}_{j_0,j_1 \in [\sqrt{n}]}$ contain n distinct and random elements in $\{0,1\}^l$. Secondly, for any $j_0, j_1 \in [\sqrt{n}]$, by the IND-CPA security of the underlying SKE scheme, we know that the distribution of both c_{j_0,j_1} and e_{j_0,j_1} is indistinguishable from the distribution of a fresh encryption of zero. Finally, by the correctness of our protocol, we know that $h = h_{\pi_0^{-1}(i_0), \pi_1^{-1}(i_1)}$. Note that π_0, π_1 are random permutations over $[\sqrt{n}]$ that are unknown to the server, hence h is a random element in $\{h_{j_0,j_1}\}_{j_0,j_1 \in [\sqrt{n}]}$. From what has been discussed above, we know that the real and simulated views are indistinguishable. □

We remark that the security of Pai needs the assumption that the client does not make any duplicate queries. The underlying reason is that the client's query is deterministic w.r.t. the client's indices, which allows the server to determine whether two queries correspond to the same index. This would violate the security definition. Fortunately, as shown in Section 2.4, this assumption can be relaxed without loss of generality by letting the client save the retrieved indices and their answers locally, with additionally $\tilde{O}(\sqrt{n})$ client storage cost.

Complexity Analysis. We analyze the asymptotic complexity of Π_{Pai} from four aspects: communication, computation, bandwidth, and storage. This analysis encompasses both the offline and online phases.

We first consider the offline phase. During this phase, the parties are required to transmit n plaintexts, $3n$ ciphertexts, and $3n$ strings, each of a length l, where $l \geq \lceil \log n \rceil$. Hence, the total communication cost is $\tilde{O}(n)$. In addition, the client needs to compute P $2n$ times, $2n$ encryptions, and n decryptions, which means the total computation is also $\tilde{O}(n)$. Moreover, since parties run the offline phase in a streaming way, and each message sent by the parties only contains $\sqrt{n}$ plaintexts or ciphertexts (and $\sqrt{n}$ l-bit long bitstrings), the bandwidth of the protocol is $\tilde{O}(\sqrt{n})$. Finally, it is easy to see that the server-side and client-side storage are $\tilde{O}(n)$ and $\tilde{O}(\sqrt{n})$, respectively.

Next we consider the online phase. For each query in the online phase, the parties transmit an l-bit string h and a ciphertext e. Hence, the communication cost per query is $\tilde{O}(1)$. In addition, the client needs to compute the PRP P one time in addition to executing a single decryption. Since the computational complexity of the permutations is a polynomial of the input length $O(\log n)$, and the computational cost of the decryption is independent of n, the total computational cost is $\tilde{O}(1)$. The bandwidth of the online phase is $\tilde{O}(1)$, for sending h and e. The server-side storage is $\tilde{O}(n)$, and the client-side storage is $\tilde{O}(1)$ for pk_0, pk_1 and ek.

5 PIR Extensions

In this section, we introduce how Pai can be extended to support other variants of PIR with high efficiency.

5.1 Overview

Pai, Piano and Spam consider the *public* database as *an array*, and the client is allowed to retrieve the i-th entry from the database. In most real-world applications, the database is organized as key-value pairs, and the client wants to retrieve the value associated with a certain key. For example, in private Wikipedia[3], the client wants to retrieve Wikipedia articles with article titles. This PIR variant has been considered as Keyword PIR (KPIR) by Chor, Gilboa, and Naor [15].

KPIR without preprocessing has been studied adequately. Several schemes have been proposed with concrete implementations and evaluations [1, 38, 48]. In contrast, there are few studies on KPIR in the preprocessing model. Zhou et al. [53] demonstrated that Piano can be modified to be KPIR, but no implementations are provided.

In some applications, the server may further restrict the client to retrieve nothing beyond its chosen key. For example, in data marketing, data sellers may share customized databases with valuable items (e.g., copyrighted music or movies) in exchange for money, and buyers pay for their desired data [25]. Allowing the client to retrieve more data than it requests leads to the loss of revenue for data sellers. Such a kind of KPIR is called Symmetric KPIR (KSPIR), introduced by Gertner et al. [29]. To the best of our knowledge, there are no concrete constructions or implementations for KSPIR in the preprocessing model.

In this section, we show that it is possible to obtain preprocessing KPIR and KSPIR based on Piano, Spam, and our Pai. However, the construction is non-trivial, and additional privacy problems should be considered. We also provide concrete implementations for KSPIR and compare their efficiency with SOTA KSPIR without preprocessing. The experimental results show that preprocessing KSPIR also enjoys extremely low online communication and computation costs.

In addition, we can leverage our idea of Pai to construct a more efficient CKSPIR, allowing the server to know whether it contains the key that the client retrieves. This is normally considered as an "extra leakage" in KSPIR. However, such "extra leakage" is necessary in practice. For example, in data marketing, the data sellers can decide how many entries the client is truly retrieved based on that "extra leakage" and ask for data pricing.

5.2 KPIR from PIR

Zhou et al. [53] proposed to use cuckoo hashing table [46] to construct KPIR from Piano. Their construction can be traced back to the idea introduced by Ali et al. [3].

A cuckoo hashing table is specified by v hash functions $\mathsf{H}_1, \ldots, \mathsf{H}_v$ with range $[m]$, where $m = (1+\varepsilon) \cdot n$ for some $\varepsilon > 0$. The goal of the cuckoo hashing table is to locate n elements into m bins so that each bin contains at most 1 element. The cuckoo hashing table avoids possible collisions by inserting elements using a recursive eviction procedure with the help of these v hash functions: whenever an element is located in an occupied bin, the occupying element is evicted and recursively relocated with a different hash function. By choosing suitable ε, we can always find suitable v hash functions with high probability. Like [3], we formalize the procedure of finding suitable v hash functions as $\mathsf{Cuckoo.KeyGen}(\mathsf{EB})$, where EB is a database containing n elements.

The KPIR construction works as follows. Let $\mathsf{EB} = \{(h_i, e_i)\}_{i \in [n]}$ be the key-value pair database held by the server. The server chooses suitable v hash functions by running $(\mathsf{H}_1, \ldots, \mathsf{H}_v) \leftarrow \mathsf{Cuckoo.KeyGen}(\mathsf{EB})$. The server then inserts each (h_i, e_i) in the bin $\mathsf{B}_{\mathsf{H}_\tau(h_i)}$ for some $\tau \in [v]$. We remark that whether the insertion succeeds depends on the selected hash functions. This means that the client may be able to learn information about the database based on the selected hash functions that result in a successful insertion. However, this does not violate privacy since the database is not private in KPIR. For empty bins, the server pads with dummy values that are denoted by $\perp$. The server finally sets the PIR

[3]See https://spiralwiki.com/ for the demo.

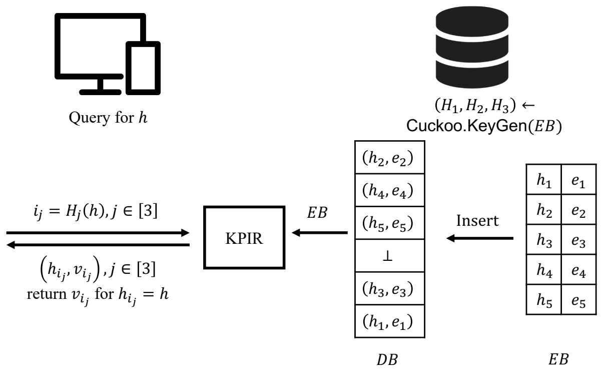
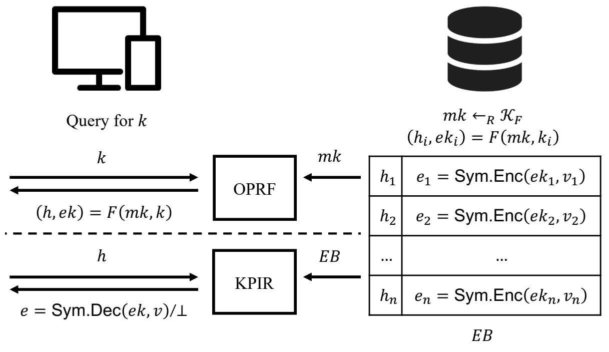


Figure 3: KPIR from PIR.



Query for k
$mk \leftarrow_R \mathcal{K}_F$
$(h_i, ek_i) = F(mk, k_i)$
k
$(h, ek) = F(mk, k)$
OPRF
mk
h
$e = $ Sym.Dec$(ek, v)/\perp$
KPIR
EB
h_1 $e_1 = $ Sym.Enc(ek_1, v_1)
h_2 $e_2 = $ Sym.Enc(ek_2, v_2)
... ...
h_n $e_n = $ Sym.Enc(ek_n, v_n)
EB

Figure 4: KSPIR from KPIR.

database as $\mathsf{DB} = (x_1, \dots, x_m)$, where each x_j is the element in B_j containing some (h_i, e_i), and sends the descriptions of $(\mathsf{H}_1, \dots, \mathsf{H}_v)$ to the client.

Whenever the client wants to query the key h, it first computes $i_j = \mathsf{H}_j(h), j \in [v]$. Then, the client runs PIR with the server, where the server takes DB as the database, and the client takes indices $i_1, \dots, i_v$ as inputs. At the end of the protocol, the client obtains $x_{i_1}, \dots, x_{i_v}$. The client finds x_{i_j} such that the first entry of x_{i_j} is h and outputs the second entry of x_{i_j}. If no such x_{i_j} exists, then the client outputs $\perp$. See Figure 3 for a tiny example with $n = 5$ and $v = 3$.

It seems trivial to directly leverage the construction into Piano, Spam, and our Pai to obtain corresponding KSPIR protocols. However, recall that some preprocessing PIR introduce simplifying assumptions shown in Section 2.4, i.e., *no duplicate queries* and *random queries*. Although these assumptions can be made without loss of generality (as shown in [53]. We stress that the methods for achieving these assumptions are necessary for KPIR constructions.

Duplicated Queries Suppression via Dummy Queries. Piano, Spam and Pai assume that the client does not make any duplicate queries. For every key query in KSPIR, the client needs to request corresponding v PIR queries determined by the output of v hash functions. This implies that in the adaptive key query setting, the client may make duplicate PIR queries for different keys k and k' when some hash functions output the same index, which happens with non-negligible probability.

Although we can allow the client to retrieve previously queried results locally, we remark that the *making dummy queries is necessary in the KSPIR construction*. Otherwise, the server would notice that the client retrievals a key h' with less than v requests, which implies that some $i_{j'} = \mathsf{H}_{j'}(h')$ is equal to previously retrieved key h with $i_j = i_{j'}$ for some $j \in [v]$, resulting in some leakages.

Random Queries without Permutation. Piano assumes that each client queries indices that are sampled at random without replacement. This assumption is naturally satisfied in the KPIR construction by modeling v hash functions as random oracles. Since the queried indices are determined by the key h and v hash functions, as long as the client does not make duplicate key queries (which have been handled by duplicated suppression), they look random from the server side.

5.3 KSPIR from KPIR

Freeman et al. [26] presented a method to construct (K)SPIR from (K)PIR based on OPRF. The key idea is to let the server encrypt each entry in the database by a session key ek_i derived from PRF F with the master key mk and each retrieval key k_i. The server and the client then run (K)PIR, and the client obtains the encrypted entry corresponding to its retrieval key k. To enable the client to decrypt, they invoke OPRF that allows the client to only obtain the corresponding session key ek for the retrieval key k without the server obtaining any information about k.

For the sake of completeness, we review the transformation in detail. See Figure 4 for a tiny example. The security proof of this transformation can be found in [26, Section 4.2].

Let $\{k_i, v_i\}_{i \in [n]}$ be the database in KSPIR. The server first encodes the database as follows.

1. Let $(\mathsf{SKE.Gen}, \mathsf{SKE.Enc}, \mathsf{SKE.Dec})$ be an semantically secure SKE scheme.

2. Let F be a PRF for an OPRF protocol. The server samples a master key mk.

3. For each $i \in [n]$, the server parses $\mathsf{F}(mk, k_i)$ as (h_i, ek_i), where h_i serves as the key of the encoded database, and ek_i is a SKE key.

4. For each $i \in [n]$, the server computes $e_i = \mathsf{SKE.Enc}(ek_i, v_i)$.

5. Let $\mathsf{EB} = \{(h_i, e_i)\}$ be the encoded database.

Now, we can obtain a KSPIR protocol from a KPIR protocol and an OPRF protocol as follows.

1. To query the key k, the parties run an OPRF, where the server takes mk as input, and the client takes k as input. At the end of OPRF, the client obtains $ek = \mathsf{F}(mk, k)$. The client parses ek as (h, ek).

2. The parties run a KPIR protocol, where the server takes EB as input and the client takes h as input. Let e be the output of the client.

3. If $e = \perp$, then the client outputs $\perp$; otherwise, the client runs $v = \mathsf{SKE.Dec}(ek, e)$ to decrypt e and outputs v.

5.4 More Efficient CKSPIR

Due to the page limitation, the detailed construction and its experimental results are shown in Appendix A.

6 Implementations and Evaluations

In this section, we illustrate the details for the implementation and experimental results, benchmarking with the SOTA schemes: Piano, Spam, and APSI (for KSPIR).

6.1 Implementation Details

We implement all our schemes and compare them with several baselines. To eliminate performance gaps caused by programming languages, we carefully study existing open-source codes, and re-implement baselines mainly using Java. The source code is available at https://github.com/alibaba-edu/mpc4j. Here, we summarize each baseline with implementation details.

Piano by Zhou et al. [53]. The authors provide a full implementation in Go. Both are open-sourced at https://github.com/pianopir/Piano-PIR. Our re-implementation exactly follows all parameter settings shown in their implementation. In their full implementation, the client generates online queries with *random* indices. To support our KSPIR construction, our re-implementation additionally allows the client to query *specific* indices.

Spam by Mughees et al. [42]. The authors implement their scheme in C++. They set the computational security parameter λ to 80 rather than 128 used in other baselines. To date, we have not found the open-source repository. We fully implement Spam with the parameter setting shown in their work with $\lambda = 128$. We also try our best to implement some optimizations shown in their work, including two subset encoding (Section 3.2), pair backup "hints" generation (Section 3.4), and improved median finding (Section 4.1).

APSI by Chen et al. [18]. We choose Labeled Private Set Intersection (LPSI) as the baseline for KSPIR. LPSI is a specific type of PSI that allows the client to learn the labels of the elements in the intersection. Chen et al. [13] pointed out that LPSI is equivalent to KSPIR in the batching setting, in the sense that the client can ask multiple keys (elements) for entries (labels) in one query. The state-of-the-art labeled PSI was proposed by Chen et al. [18], with the open-sourced library APSI available at https://github.com/microsoft/APSI. Their implementation invokes Microsoft SEAL library[4] for Fully Homomorphic Encryption (FHE) and FourQ[5] for Elliptic Curve Cryptography (ECC).

We implement Piano, Spam, and Pai under a unified API. Then, we implement the transformation from preprocessing PIR to KSPIR by invoking these schemes in a black-box manner with a cuckoo hash. Following the estimates in [24], we choose the cuckoo hash parameters as containing 3 hashes and the number of bins $b = 1.5n$ such that inserting n elements in the cuckoo hash fails with probability at most $2^{-\kappa}$. In this way, we obtain the corresponding KSPIR, namely, PianoKSPIR, SpamKSPIR, and PaiKSPIR. We finally implement our PaiCKSPIR with FourQ ECC as the DDH-hard group.

In our implementation, we set the statistical security parameter κ to 40 and the computational security parameter λ to 128 for all schemes. We utilize AES-NI hardware instruction that is inherently supported by modern Java Virtual Machines for fast PRF and PRP evaluations. The CPA-secure SKE used in Pai and PaiCKSPIR is instantiated as AES with OFB encryption mode. The semantic-secure SKE used in PianoKSPIR, SpamKSPIR, and PaiKSPIR is instantiated as AES with CTR encryption mode.

6.2 Experimental Setup

We evaluate all schemes on a single Intel Core i9-9900K with 3.6GHz and 128GB RAM. Our platform runs Ubuntu 20.04.6 LTS, with Microsoft SEAL 4.1.1, FourQ v3.1, and Java 17.0.1. All query costs are computed as the average over 1000 queries, except the somewhat inefficient APSI, which we average the cost over 100 queries. All experiments are run on a single machine with the 10Gbps bandwidth and 0.05ms RTT latency simulated by the Linux `tc` command. All computations for the server and the client are performed on 8 threads.

We analyze the performance of all schemes for databases with $n = 2^{20}, 2^{22}, 2^{24}$ entries. The entry sizes are $64, 128, 256$ bytes. Since our experiments consider single-query setting rather than batch-query setting, for APSI, we choose the single-query parameter 1M-1-32[6] for $n = 2^{20}$ entries, and 16M-1-32[7] for $n = 2^{22}$ and $n = 2^{24}$ entries.

6.3 Evaluation Results

We analyze the experimental results for PIR and KSPIR, respectively. The metrics include the costs of the queries in

[4]https://github.com/microsoft/SEAL

[5]https://github.com/microsoft/FourQlib

[6]https://github.com/microsoft/APSI/blob/main/parameters/1M-1-32.json

[7]https://github.com/microsoft/APSI/blob/main/parameters/16M-1-32.json

Table 2: Performance of Pai, Piano and Spam on $n = 2^{20}, 2^{22}, 2^{24}$ database sizes and $64, 128, 256$ bytes entry sizes. "Comm." stands for communication cost. Offline costs are the whole preprocessing. Online costs are amortized over 1000 queries.

		$n = 2^{20}$			$n = 2^{22}$			$n = 2^{24}$		
		64 bytes	128 bytes	256 bytes	64 bytes	128 bytes	256 bytes	64 bytes	128 bytes	256 bytes
Offline Comm. (MB)	Piano	64.0	128.0	256.0	256.0	512.0	1024.0	1024.0	2048.0	4096.0
	Spam	64.0	128.0	256.0	256.0	512.0	1024.0	1024.0	2048.0	4096.0
	Pai	352.0	608.0	1120.0	1408.0	2432.0	4480.0	5632.0	9728.0	17920.0
Offline Time (s)	Piano	4.730	4.610	5.737	18.352	16.434	19.738	78.094	69.327	86.643
	Spam	7.462	7.999	8.313	23.573	26.214	33.434	92.303	102.933	132.366
	Pai	4.504	5.238	6.798	14.496	16.988	22.777	57.525	66.840	90.584
Online Comm. (KB)	Piano	65.998	129.998	257.998	131.998	259.998	515.998	263.998	519.998	1031.998
	Spam	2.250	2.375	2.625	4.375	4.500	4.750	8.625	8.750	9.000
	Pai	0.094	0.156	0.297	0.094	0.156	0.297	0.094	0.156	0.297
Online Time (ms)	Piano	1.184	1.509	1.906	1.575	2.244	2.833	2.816	4.375	5.492
	Spam	0.905	1.088	1.144	0.977	1.094	1.306	1.542	1.935	2.241
	Pai	0.357	0.364	0.281	0.245	0.253	0.281	0.210	0.221	0.281
Online Client-Side Storage (MB)	Piano	17.177	22.143	32.398	30.747	41.373	62.848	60.125	82.627	127.847
	Spam	16.841	22.191	32.218	28.595	39.070	59.347	52.462	73.182	113.964
	Pai	5.131	5.097	4.907	4.960	4.959	4.734	4.959	4.958	4.734

the online phase, as well as the one-time pre-processing cost in the offline phase. We further obtain the online client-side storage cost by using the JOL (Java Object Layout) library[8] to measure the deep sizes (i.e., the size of an object including the size of all referred objects, in addition to the size of the object itself) of Objects packaging the client.

Evaluations for PIR. Table 2 shows the detailed experiment results for Pai, Piano, and Spam. Similar to results provided in [42], our re-implementation of Spam also achieves better communication and computation than Piano. Under our fair comparisons, the online communication cost is 29.3 - 114.7× better, and the online computation cost is 1.3 - 2.5× better. Compared with Spam, Pai further reduces the online communication cost by 8.8 - 91.8×, and the online time by 2.5 - 8.8×. In addition, since Pai does not require the client to store any hints, the online client-side storage is constant, compared to Piano and Spam that sublinearly increase in n. Based on our parameter setting, the concrete storage costs of Pai are only $3.7\% - 30.5\%$ than Piano and Spam.

The offline time of Pai is also similar to that of Piano and lower than that of Spam. The only possible disadvantage of Pai is it introduces additional offline communication costs, which are about 4 - 6× compared to that of Piano and Spam. Since the server and the client can enjoy the extremely fast online phase by performing the preprocessing upfront *only once*, a relatively high but reasonable offline communication is acceptable.

Evaluations for KSPIR. Table 3 shows the detailed experiment results for APSI, PianoKSPIR, SpamKSPIR, and PaiKSPIR. The online complexity of APSI is much higher than that of the other three protocols, at least three orders of magnitude more expensive. This shows that introducing preprocessing in KSPIR extremely decreases the online costs. The advantage of APSI is that the communication in the offline phase is very low. However, due to the high offline computation complexity of APSI, its overall offline time is similar to that of the other three protocols. We can argue that the application scenario of APSI is different from that of the other three protocols. In particular, when low offline communication complexity is required, APSI will be a better choice than others.

Now, let us compare the complexity of the other three protocols. The experimental results show that SpamKSPIR achieves better communication and computation than PianoKSPIR. Under our fair comparisons, the online communication cost is 36.1 - 122.4× better, and the online computation cost is 1.4 - 2.4× better. PaiKSPIR further achieves more efficient online phase than SpamKSPIR. Concretely, PaiKSPIR reduces the online communication cost by 9.9 - 81.3×, and the online time by 2.3 - 8.9×. In addition, PaiKSPIR inherits the storage advantages of Pai. Under the parameters we tested, the concrete storage cost of PaiKSPIR is only $2.9\% - 23.4\%$ than that of PianoKSPIR and SpamKSPIR.

Finally, we consider the offline complexity of the protocols. Similar to the comparisons between Pai, Piano, and Spam, the offline time of PaiKSPIR is similar to that of PianoKSPIR and lower than that of SpamKSPIR. Also, PaiKSPIR inherits the disadvantage of high offline communication from Pai, which is about 4 - 6× than that of PianoKSPIR and SpamKSPIR. Given that we have an extremely fast online phase, a relatively high but reasonable offline communication is acceptable.

[8]http://hg.openjdk.java.net/code-tools/jol

Table 3: Performance of APSI, PaiKSPIR, PianoKSPIR and SpamKPSIR on $n = 2^{20}, 2^{22}, 2^{24}$ database sizes and 64, 128, 256 bytes entry sizes. "Comm." stands for communication cost. Offline costs are the whole preprocessing. Online costs are amortized over 1000 queries except APSI that are amortized over 100 queries.

		$n = 2^{20}$			$n = 2^{22}$			$n = 2^{24}$		
		64 bytes	128 bytes	256 bytes	64 bytes	128 bytes	256 bytes	64 bytes	128 bytes	256 bytes
Offline Comm. (MB)	APSI	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
	PianoKSPIR	120.069	216.124	408.235	480.085	864.153	1632.289	1920.341	3456.613	6529.158
	SpamKSPIR	120.069	216.124	408.235	480.085	864.153	1632.289	1920.341	3456.613	6529.158
	PaiKSPIR	624.359	1008.581	1777.023	2496.443	4032.715	7105.260	9985.771	16130.862	28421.042
Offline Time (s)	PianoKSPIR	17.388	15.517	18.294	70.225	63.999	69.712	288.962	255.728	296.759
	SpamKSPIR	20.544	20.918	24.371	82.174	82.754	104.972	329.514	331.764	417.963
	APSI	23.697	35.538	61.234	82.641	128.631	230.835	338.196	542.071	938.441
	PaiKSPIR	16.218	17.331	20.157	60.437	67.729	78.297	249.635	275.815	308.757
Online Comm. (KB)	APSI	5769.693	9401.396	17270.112	13517.358	22959.823	43418.503	47413.396	82277.844	157817.458
	PianoKSPIR	301.311	536.436	1006.686	602.564	1072.814	2013.314	1205.313	2146.000	4027.375
	SpamKSPIR	8.351	8.726	9.476	16.158	16.533	17.283	31.773	32.148	32.898
	PaiKSPIR	0.391	0.578	0.953	0.391	0.578	0.953	0.391	0.578	0.953
Online Time (ms)	APSI	3067.12	4072.03	6249.78	5464.79	8034.68	13886.16	15149.30	25075.80	47133.38
	PianoKSPIR	4.398	5.208	6.754	7.121	9.032	11.323	14.363	17.201	21.827
	SpamKSPIR	3.178	3.235	3.378	4.065	4.497	4.793	7.702	8.021	8.940
	PaiKSPIR	1.391	1.436	1.444	1.045	1.031	1.071	1.055	0.989	1.008
Online Client-Side Storage (MB)	APSI	5.260	5.256	5.436	5.274	5.271	5.272	5.271	5.266	5.437
	PianoKSPIR	21.888	28.172	40.740	40.272	53.444	79.789	81.095	109.272	165.626
	SpamKSPIR	21.114	27.396	39.963	37.123	49.682	74.801	69.485	94.598	144.824
	PaiKSPIR	4.931	5.161	4.950	4.760	4.990	4.778	4.758	4.988	4.776

Evaluations for CKSPIR. For our CKSPIR protocol, we defer the corresponding experimental results to Appendix A.

7 Conclusion

In this work, we propose a novel PIR framework Pai under the client-preprocessing model. This new paradigm of PIR is compelling both from theoretical and practical perspectives. Theoretically, Pai boasts near-optimal online time and client-side storage, with the optimality retained up to a poly-logarithmic factor. As a comparison, the online time and online client-side storage of the state-of-the-art PIR protocols Piano and Spam under the client-preprocessing model are both $\tilde{O}(\sqrt{n})$. It is pivotal to emphasize that the low client-side storage, rendering our protocol considerably more feasible for practical implementation and deployment in a wide variety of computing environments. Beyond the theoretical contributions of our protocols, we have also demonstrated practical efficiency of Pai through substantial experiments. Our findings indicate that across all evaluated databases, Pai consistently outperforms in terms of online communication, execution time, and client-side storage. A limitation of our protocol is the requisite for the database to undergo an offline phase for each client. This stipulation results in the server's preprocessing time scaling linearly with the client count. However, under the assumption of non-collusion between any client and the server, by letting the clients share the outputs in the offline phase, the server only needs to run the offline phase once. As a consequence, one problem is how to enable the server to perform the offline phase only once, while allowing some clients to collude with the server to attack other clients.

We also extended our PIR framework to construct two variants of PIR, i.e., KSPIR and CKSPIR. Our KSPIR protocol PaiKSPIR is obtained by applying the ideas from [26] and [3] to our PIR protocol. In order to verify the efficiency of the conversion, we also apply the conversion to Piano and Spam, obtaining two KSPIR protocols PianoKSPIR and SpamKSPIR, respectively. We implement all the KSPIR protocols, and our experimental results show that PaiKSPIR has higher efficiency than PianoKSPIR and SpamKSPIR. We also introduce CKSPIR, which is a variant of KSPIR. CKSPIR is mainly used when the server intends to charge the client for a successful query. By using different cryptography techniques, we convert Pai to a CKSPIR protocol PaiCKSPIR. Since any KSPIR protocol can be converted to a CKSPIR protocol by adding one bit of communication, we also compare the efficiency of PaiCKSPIR and our KSPIR protocol, and the experimental results show that our PaiCKSPIR protocol achieves lower online time and similar client-side storage.

References

[1] Ishtiyaque Ahmad, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Pantheon: Private retrieval from public key-value store. *Proc. VLDB Endow.*, pages 643–656, 2022.

[2] Kinan Dak Albab, Rawane Issa, Mayank Varia, and Kalman Graffi. Batched differentially private information retrieval. In *USENIX Security 2022*, pages 3327–3344.

[3] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication-computation trade-offs in PIR. In *USENIX Security 2021*, pages 1811–1828.

[4] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *SP 2018*, pages 962–979.

[5] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *CRYPTO 2000*, pages 55–73.

[6] Keller Blackwell and Mary Wootters. A note on the permuted puzzles toy conjecture. *CoRR*, 2021.

[7] Dan Boneh, Kevin Lewi, and David J. Wu. Constraining pseudorandom functions privately. In *PKC 2017*, pages 494–524.

[8] Elette Boyle, Justin Holmgren, Fermi Ma, and Mor Weiss. On the security of doubly efficient pir. Cryptology ePrint Archive, Paper 2021/1113, 2021.

[9] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can we access a database both locally and privately? In *TCC 2017*, pages 662–693.

[10] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *EUROCRYPT 1999*, pages 402–414.

[11] Ran Canetti, Justin Holmgren, and Silas Richelson. Towards doubly efficient private information retrieval. In *TCC 2017*, pages 694–726.

[12] Yan-Cheng Chang. Single database private information retrieval with logarithmic communication. In *ACISP 2004*, pages 50–61.

[13] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In *CCS 2018*, pages 1223–1237.

[14] Benny Chor and Niv Gilboa. Computationally private information retrieval (extended abstract). In *STOC 1997*, pages 304–313.

[15] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. *IACR Cryptol. ePrint Arch.*, page 3, 1998.

[16] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *FOCS 1995*, 1995.

[17] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 1998.

[18] Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. Labeled PSI from homomorphic encryption with reduced computation and communication. In *CCS 2021*, pages 1135–1150.

[19] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-server private information retrieval with sublinear amortized time. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022*, pages 3–33.

[20] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In *EUROCRYPT 2020*, pages 44–75.

[21] Giovanni Di Crescenzo, Yuval Ishai, and Rafail Ostrovsky. Universal service-providers for database private information retrieval (extended abstract). In *PODC 1998*, pages 91–100.

[22] Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Fast and private computation of cardinality of set intersection and union. In *CANS 2012*, pages 218–231.

[23] Alex Davidson, Gonçalo Pestana, and Sofía Celi. Frodopir: Simple, scalable, single-server private information retrieval. *PoPETs 2023*, pages 365–383.

[24] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: scaling private contact discovery. *PoPETs 2018*, pages 159–178.

[25] Raul Castro Fernandez. Protecting data markets from strategic buyers. In *SIGMOD 2022*, pages 1755–1769.

[26] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *TCC 2005*, pages 303–324.

[27] Craig Gentry and Shai Halevi. Compressible FHE with applications to PIR. In *TCC 2019*, pages 438–464.

[28] Craig Gentry and Zulfikar Ramzan. Single-database private information retrieval with constant communication rate. In *ICALP 2005*, pages 803–815.

[29] Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. Protecting data privacy in private information retrieval schemes. In *STOC 1998*.

[30] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. In *USENIX Security 2023*.

[31] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In *STOC 2004*, pages 262–271.

[32] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In *USENIX Security 2021*, pages 875–892.

[33] Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *FOCS 1997*, pages 364–373.

[34] Arthur Lazzaretti and Charalampos Papamanthou. Treepir: Sublinear-time and polylog-bandwidth private information retrieval from DDH. In *CRYPTO 2023*, pages 284–314.

[35] Arthur Lazzaretti and Charalampos Papamanthou. Single server PIR with sublinear amortized time and polylogarithmic bandwidth. *IACR Cryptol. ePrint Arch.*, page 830, 2022.

[36] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. Doubly efficient private information retrieval and fully homomorphic RAM computation from ring LWE. In *STOC 2023*, pages 595–608.

[37] Helger Lipmaa. First CPIR protocol with data-dependent computation. In *ICISC 2009*, pages 193–210.

[38] Rasoul Akhavan Mahdavi and Florian Kerschbaum. Constant-weight PIR: single-round keyword PIR via constant-weight equality operators. In *USENIX Security 2022*, pages 1723–1740.

[39] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR : Private information retrieval for everyone. *PoPETs 2016*, pages 155–174.

[40] Samir Jordan Menon and David J. Wu. SPIRAL: fast, high-rate single-server PIR via FHE composition. In *SP 2022*, pages 930–947.

[41] Muhammad Haris Mughees, Hao Chen, and Ling Ren. Onionpir: Response efficient single-server PIR. In *CCS 2021*, pages 2292–2306.

[42] Muhammad Haris Mughees, Sun I, and Ling Ren. Simple and practical amortized sublinear private information retrieval. 2023.

[43] Muhammad Haris Mughees and Ling Ren. Vectorized batch private information retrieval. In *SP 2023*, pages 437–452.

[44] Hiroki Okada, Rachel Player, Simon Pohmann, and Christian Weinert. Towards practical doubly-efficient private information retrieval. Cryptology ePrint Archive, Paper 2023/1510, 2023.

[45] Rafail Ostrovsky and William E. Skeith III. A survey of single-database private information retrieval: Techniques and applications. In *PKC 2007*, pages 393–411.

[46] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, pages 122–144, 2004.

[47] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Private stateful information retrieval. In *CCS 2018*, pages 1002–1019.

[48] Sarvar Patel, Joon Young Seo, and Kevin Yeo. Don't be dense: Efficient keyword PIR for sparse databases. In *USENIX Security 2023*.

[49] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce M. Maggs. Puncturable pseudorandom sets and private information retrieval with near-optimal online bandwidth and time. In *CRYPTO 2021*, pages 641–669.

[50] Raphael R. Toledo, George Danezis, and Ian Goldberg. Lower-cost $\in$-private information retrieval. *PoPETs 2016*, pages 184–201.

[51] Ni Trieu, Kareem Shehata, Prateek Saxena, Reza Shokri, and Dawn Song. Epione: Lightweight contact tracing with strong privacy. *IEEE Data Eng. Bull.*, pages 95–107, 2020.

[52] Mingxun Zhou, Wei-Kai Lin, Yiannis Tselekounis, and Elaine Shi. Optimal single-server private information retrieval. In *EUROCRYPT 2023*, pages 395–425.

[53] Mingxun Zhou, Andrew Park, Elaine Shi, and Wenting Zheng. Piano: Extremely simple, single-server PIR with sublinear server computation. *To appear in SP 2024*.

A More Efficient CKSPIR

Since PaiKSPIR is the KSPIR protocol with the best online efficiency, the CKSPIR protocol converted from PaiKSPIR is the CKSPIR protocol with the best online efficiency at present. In this section, we present a new CKSPIR protocol that achieves better online efficiency than PaiKSPIR.

The Construction. Our CKSPIR protocol PaiCKSPIR is constructed based on our PIR protocol Pai. However, since CKSPIR implements stronger functionality (keyword search) and requires higher privacy (symmetric privacy), PaiCKSPIR has a much more complicated offline phase than PaiPIR. Concretely, the offline phase PaiCKSPIR is mainly different from that of PaiPIR in the following aspects (let $\{(k_i, v_i)\}_{i\in[n]}$ be the database).

- **Encoding Functions.** The two encoding algorithms $\mathsf{E}_{\mathsf{key}}$ and $\mathsf{E}_{\mathsf{value}}$ (which play the roles of $\mathsf{E}_{\mathsf{index}}$ and $\mathsf{E}_{\mathsf{entry}}$ in PaiPIR) are used to encode the keys and values, respectively. Instead of using a PRP, we instantiate $\mathsf{E}_{\mathsf{key}}$ with the function $f_\beta(k) = \mathsf{H}(k)^\beta$, where H is a collision-resistant hash function (CRHF) that maps the keys to elements of a DDH-hard group of order p for some large prime p, and $\beta \in \mathbb{Z}_p$ is random key held by the client. In the random oracle model, the function $f_\beta(k)$ is a PRF under the Decisional Diffie–Hellman (DDH) assumption [22]. As in PaiPIR, $\mathsf{E}_{\mathsf{value}}$ is still instantiated with an IND-CPA secure SKE scheme.

- **Offline Phase.** In PaiPIR, the server directly sends the database to the client. In PaiCKSPIR, we must protect the privacy of the database. Therefore, we let the server send an encryption of the database. Concretely, we let the server sends $(t_i, c_i) = (\mathsf{H}(k_i)^\alpha, v_i \oplus r_i)$ instead of just (k_i, v_i), where $\alpha \in \mathbb{Z}_p$ is random, and r_i is a random value. This prevents the client from obtaining information about (k_i, v_i). Now, we can let the client encrypt the new database $\{(t_i, c_i)\}_{i\in[n]}$ using $\mathsf{E}_{\mathsf{key}}$ and $\mathsf{E}_{\mathsf{value}}$. Concretely, the client chooses a random $\beta \in \mathbb{Z}_p$ and an encryption key ek of an IND-CPA secure SKE scheme. Moreover, the client chooses a random permutation π over $[n]$. Then, the client encrypts the database as $\{(h_i, e_i)\}_{i\in[n]}$, where $h_i = t^\beta_{\pi(i)}, e_i = \mathsf{SKE.Enc}(ek, c_{\pi(i)})$. Finally, the client sends $\{(h_i, e_i)\}_{i\in[n]}$ to the server. We remark that using the same idea as in PaiPIR, we can also reduce the offline bandwidth from $\tilde{O}(n)$ to $\tilde{O}(\sqrt{n})$.

- **Online Phase.** Since the server also participates in encoding the database, PaiCKSPIR will have a different online phase from PaiPIR. Assume that the client has a search key k, the client first computes $h_c = \mathsf{H}(k)^\beta$ and then sends h_c to the server. Then the server computes $h = h_c^\alpha$ and checks whether there exists some h_j such that $h_j = h$. By the properties of H, we can easily verify the correctness. If the server finds that $h = h_j$, then it sends e_j to the client. By decrypting e_j using the SKE key ek, the client obtains the value $c_{\pi(j)} = v_{\pi(j)} \oplus r_{\pi(j)}$. To let the client obtain the value $v_{\pi(j)}$, we still need to let the client obtain $r_{\pi(j)}$, which seems impossible due to that the server does not know the permutation π. To deal with this, we use oblivious pseudorandom function (OPRF) in our protocol. Concretely, instead of choosing r_i at random, we let $r_i = \mathsf{F}(mk, k_i)$, where F is a PRF and mk is a PRF key sampled by the server. Then, the client can obtain the value $r_{\pi(j)} = \mathsf{F}(mk, k_{\pi(j)}) = \mathsf{F}(mk, k)$ by running an OPRF protocol with the server.

Finally, we remark that our protocol may leak information about i to the client if $k = k_i$ for some $i \in [n]$, while we only want to leak v_i to the client. In fact, this has no effect on the considered application, where the server wants the client to pay for its successful retrieval. However, this problem can be resolved by simply letting the server permute its database at the beginning of the offline phase, which does not affect the efficiency of the online phase of our protocol.

Now, we are ready to present the description of PaiCKSPIR, which is shown in Figure 5.

Security Proof. We prove the following theorem to state the security of $\Pi_{\mathsf{PaiCKSPIR}}$.

Theorem 2. $\Pi_{\mathsf{PaiCKSPIR}}$ *is a secure CKSPIR protocol.*

Proof. We need to show the correctness, server-privacy, and client-privacy of $\Pi_{\mathsf{PaiCKSPIR}}$.

Correctness. If $k = k'_{i_0,i_1}$ for some $i_0, i_1 \in [\sqrt{n}]$, then $h = h_{\pi_0^{-1}(i_0),\pi_1^{-1}(i_1)}$, and the server will output $b = 1$. Moreover, by the collision-resistance of H, for any $(j_0, j_1) \neq (\pi_0^{-1}(i_0), \pi_1^{-1}(i_1))$, with overwhelming probability it holds that $h \neq h_{j_0,j_1}$. Therefore, with overwhelming probability, the server will return $e_{\pi_0^{-1}(i_0),\pi_1^{-1}(i_1)}$ to the client. By the offline phase, we know that $e_{\pi_0^{-1}(i_0),\pi_1^{-1}(i_1)}$ is an encryption of $m_{i_0,i_1} = v'_{i_0,i_1} \oplus r_{i_0,i_1}$. Therefore, the client will output

$$v = \mathsf{SKE.Dec}(ek, e) \oplus r = v'_{i_0,i_1} \oplus r_{i_0,i_1} \oplus \mathsf{F}(mk, k_{i_0,i_1}) = v'_{i_0,i_1}.$$

If $k \neq k'_{j_0,j_1}$ for any $j_0, j_1 \in [\sqrt{n}]$, then by the collision-resistance of H, the probability that there exists some $i_0, i_1 \in [\sqrt{n}]$ such that $\mathsf{H}(k'_{i_0,i_1})^{\alpha\beta} = \mathsf{H}(k)^{\alpha\beta}$ is negligible (α, β are random). Therefore, with overwhelming probability, the server will output $b = 0$ and the client will output $\perp$.

Server-Privacy. Let $\mathsf{View}_{\mathsf{cli}}(\mathsf{DB}, k)$ be the view of the client. We need to construct a PPT simulator Sim taking (k, z) as input such that

$$\mathsf{View}_{\mathsf{cli}}(\mathsf{DB}, k) \approx_c \mathsf{Sim}(k, z).$$

Note that $\mathsf{View}_{\mathsf{cli}}(\mathsf{DB}, k)$ consists of

$$(k, \{(s_{j_0,j_1}, m_{j_0,j_1})\}_{j_0,j_1\in[\sqrt{n}]},$$
$$\{(t_{j_0,j_1}, c_{j_0,j_1})\}_{j_0,j_1\in[\sqrt{n}]}, \mathsf{resp}, \mathsf{vc}_{\mathsf{oprf}}, z)$$

where resp is the message received from the server in the online phase, and $\mathsf{vc}_{\mathsf{oprf}}$ is the view of the client in the OPRF protocol. To simulate this view, Sim performs as follows.

1. Sample two random permutations π_0^*, π_1^* over the set $[\sqrt{n}]$, two random values $\beta_0^*, \beta_1^* \in \mathbb{Z}_p$ (set $\beta^* = \beta_0^*\beta_1^*$), and one SKE key ek^*.

Figure 5: The construction of PaiCKSPIR.

Protocol $\Pi_{\mathsf{PaiCKSPIR}}$: CKSPIR with $\tilde{O}(1)$ Client-Side Storage and Online Time

Input: The server $\mathcal{S}$ has a database $\mathsf{DB} = \{(k_j, v_j)\}_{j\in[n]} \in (\mathcal{K}\times\mathcal{V})^n$, and the client $\mathcal{C}$ has a search key $k \in \mathcal{K}$.
Output: If there exists some $i \in [n]$ such that $k_i = k$, then $\mathcal{S}$ outputs $b = 1$, and $\mathcal{C}$ outputs $z = v_i$. Otherwise, $\mathcal{S}$ outputs $b = 0$, and $\mathcal{C}$ outputs $z = \perp$.

Offline Phase: $\mathcal{S}$ first samples a random permutation π over $[n]$ and permutes its database as $\mathsf{DB}' = \{(k'_j, v'_j)\}_{j\in[n]}$, where $(k'_j, v'_j) = (k_{\pi(j)}, v_{\pi(j)})$. Then, $\mathcal{S}$ represents the database as $\mathsf{DB}' = \{(k'_{j_0,j_1}, v'_{j_0,j_1})\}_{j_0,j_1\in[\sqrt{n}]}$. $\mathcal{S}$ and $\mathcal{C}$ agree on the following three cryptographic primitives.

- A hash function $\mathsf{H} : \mathcal{K} \to \mathbb{G}$, where $\mathbb{G}$ is a DDH-hard group of order p for some large prime p.
- An IND-CPA secure SKE scheme $(\mathsf{SKE.Gen}, \mathsf{SKE.Enc}, \mathsf{SKE.Dec})$.
- A PRF $\mathsf{F} : \mathcal{K}_{\mathsf{F}} \times \mathcal{K} \to \mathcal{V}$ with an OPRF protocol, where $\mathcal{K}_{\mathsf{F}}$ is the key space of F.

The parties choose the required parameters.

- $\mathcal{S}$ chooses a master key $mk \in \mathcal{K}_{\mathsf{F}}$ and a random value $\alpha \in \mathbb{Z}_p$.
- $\mathcal{C}$ chooses two random permutations π_0, π_1 over the set $[\sqrt{n}]$, two random values $\beta_0, \beta_1 \in \mathbb{Z}_p$ (set $\beta = \beta_0\beta_1$), and one SKE key ek.

Now $\mathcal{S}$ and $\mathcal{C}$ interact to encode the database in a streaming way.

1. Row-Permuting. For each $j_1 \in [\sqrt{n}]$, $\mathcal{S}$ and $\mathcal{C}$ perform the following steps.
   (a) For all $j_0 \in [\sqrt{n}]$, $\mathcal{S}$ computes $s_{j_0,j_1} = \mathsf{H}(k'_{j_0,j_1})^{\alpha}$.
   (b) For all $j_0 \in [\sqrt{n}]$, $\mathcal{S}$ first computes $r_{j_0,j_1} = \mathsf{F}(mk, k'_{j_0,j_1})$ and $m_{j_0,j_1} = v'_{j_0,j_1} \oplus r_{j_0,j_1}$. Then, $\mathcal{S}$ sends $\{(s_{j_0,j_1}, m_{j_0,j_1})\}_{j_0\in[\sqrt{n}]}$ to $\mathcal{C}$.
   (c) For all $j_0 \in [\sqrt{n}]$, $\mathcal{C}$ first computes $t_{j_0,j_1} = (s_{\pi_0(j_0),j_1})^{\beta_0}$ and $c_{j_0,j_1} = \mathsf{SKE.Enc}(ek, m_{\pi_0(j_0),j_1})$. Then, $\mathcal{C}$ sends $\{(t_{j_0,j_1}, c_{j_0,j_1})\}_{j_0\in[\sqrt{n}]}$ to $\mathcal{S}$.
   (d) $\mathcal{C}$ deletes $\{(s_{j_0,j_1}, m_{j_0,j_1}), (t_{j_0,j_1}, c_{j_0,j_1})\}_{j_0\in[\sqrt{n}]}$ from its local storage.
2. Column-Permuting. For each $j_0 \in [\sqrt{n}]$, $\mathcal{S}$ and $\mathcal{C}$ perform the following steps.
   (a) $\mathcal{S}$ sends $\{(t_{j_0,j_1}, c_{j_0,j_1})\}_{j_1\in[\sqrt{n}]}$ to $\mathcal{C}$.
   (b) For all $j_1 \in [\sqrt{n}]$, $\mathcal{C}$ computes $h_{j_0,j_1} = t^{\beta_1}_{j_0,\pi_1(j_1)}$ and decrypts $x_{j_0,j_1} = \mathsf{SKE.Dec}(ek, c_{j_0,j_1})$.
   (c) For all $j_1 \in [\sqrt{n}]$, $\mathcal{C}$ computes $e_{j_0,j_1} = \mathsf{SKE.Enc}(ek, x_{j_0,\pi_1(j_1)})$. Then, $\mathcal{C}$ sends $\{(h_{j_0,j_1}, e_{j_0,j_1})\}_{j_1\in[\sqrt{n}]}$ to $\mathcal{S}$.
   (d) $\mathcal{C}$ deletes $\{(t_{j_0,j_1}, c_{j_0,j_1}), (h_{j_0,j_1}, e_{j_0,j_1})\}_{j_1\in[\sqrt{n}]}$ from its local storage.

Online Phase: To query the key k, the parties proceed as follows (and they run the OPRF protocol to let $\mathcal{C}$ obtain $r = \mathsf{F}(mk, k)$.

1. Query. $\mathcal{C}$ first computes $q = \mathsf{H}(k)^{\beta}$ and sends q to $\mathcal{S}$.
2. Response. $\mathcal{S}$ computes $h = q^{\alpha}$ and checks whether there exists some h_{j_0,j_1} such that $h_{j_0,j_1} = h$. If the answer is yes, $\mathcal{S}$ outputs $b = 1$ and sends $e = e_{j_0,j_1}$ to $\mathcal{C}$. Otherwise, $\mathcal{S}$ outputs $b = 0$ and sends $\perp$ to $\mathcal{C}$.
3. Extract. If receiving $\perp$ from $\mathcal{S}$, then $\mathcal{C}$ outputs $z = \perp$. Otherwise, $\mathcal{C}$ computes $v = \mathsf{SKE.Dec}(ek, e) \oplus r$ and outputs $z = v$.

2. Choose n random group elements $\{s^*_{j_0,j_1}\}_{j_0,j_1\in[\sqrt{n}]}$ from $\mathbb{G}$ and n random values $\{m^*_{j_0,j_1}\}_{j_0,j_1\in[\sqrt{n}]}$ from $\mathcal{V}$.

3. Compute $t^*_{j_0,j_1} = (s^*_{\pi^*_0(j_0),j_1})^{\beta^*_0}$ and $c^*_{j_0,j_1} = \mathsf{SKE.Enc}(ek^*, m_{\pi^*_0(j_0),j_1})$ for all $j_0, j_1 \in [\sqrt{n}]$.

4. If $z \neq \perp$, then choose random $i^*_0, i^*_1 \in [\sqrt{n}]$ and compute $r^* = m^*_{i^*_0,i^*_1} \oplus z$. Otherwise, choose a random value r^* from $\mathcal{V}$.

5. If $z \neq \perp$, then compute $\mathsf{resp}^* = \mathsf{SKE.Enc}(ek^*, z \oplus r^*)$. Otherwise, compute $\mathsf{resp}^* = \perp$.

6. Invoke the OPRF simulator on (k, r^*) and let $\mathsf{vc}^*_{\mathsf{oprf}}$ be the output.

7. Output the simulated view

$$(k, \{(s^*_{j_0,j_1}, m^*_{j_0,j_1})\}_{j_0,j_1\in[\sqrt{n}]},\\ \{(t^*_{j_0,j_1}, c^*_{j_0,j_1})\}_{j_0,j_1\in[\sqrt{n}]}, \mathsf{resp}^*, \mathsf{vc}^*_{\mathsf{oprf}}, z).$$

It remains to show that the simulated view is indistinguishable from the real view. Firstly, note that $f_\alpha(k) = \mathsf{H}(k)^\alpha$ is a PRF in the RO model, hence each s_{j_0,j_1} is indistinguishable from a random value. Moreover, $t_{j_0,j_1} = (s_{\pi_0(j_0),j_1})^{\beta_0}$, where π_0 is a random permutation over $[\sqrt{n}]$ that is sampled by the client. Therefore, we know that $s_{\pi_0(j_0),j_1}$ and $s^*_{\pi^*_0(j_0),j_1}$ are indistinguishable, which implies that t_{j_0,j_1} and $t^*_{j_0,j_1}$ are indistinguishable. On the other hand, by the property of F, we know that every $r_{j_0,j_1} = \mathsf{F}(mk, k'_{j_0,j_1})$ is a pseudorandom value, hence every $m_{j_0,j_1} = v_{j_0,j_1} \oplus \mathsf{F}(mk, k'_{j_0,j_1})$ is indistinguishable from a random value. Moreover, note that $c_{j_0,j_1} = \mathsf{SKE.Enc}(ek, m_{\pi_0(j_0),j_1})$, where ek is a SKE key sampled by the client. Since $m_{\pi_0(j_0),j_1}$ and $m^*_{\pi^*_0(j_0),j_1}$ are indistinguishable, c_{j_0,j_1} and $c^*_{j_0,j_1}$ are indistinguishable. Now, we consider the message $(\mathsf{resp}, \mathsf{vc}_{\mathsf{oprf}})$. We first consider the case of $z = \perp$. In this case, we have $\mathsf{resp} = \perp$, and moreover, $k \neq k'_{j_0,j_1}$ for all $j_0, j_1 \in [\sqrt{n}]$, which implies that the OPRF output $r = \mathsf{F}(mk, k)$ has not been used in the offline phase, and we can just use a random value r^* to simulate r. Let $\mathsf{Sim}_{\mathsf{oprf}}$ be the OPRF simulator, then we know that $\mathsf{vc}_{\mathsf{oprf}} = \mathsf{Sim}_{\mathsf{oprf}}(k, r)$ and $\mathsf{vc}^*_{\mathsf{oprf}} = \mathsf{Sim}_{\mathsf{oprf}}(k, r^*)$ are indistinguishable. For the case of $z \neq \perp$, by the correctness of our protocol, we know that $k = k'_{i_0,i_1}$ for some $i_0, i_1 \in [\sqrt{n}]$. Note that the database has been permuted by the server, hence i_0, i_1 are in fact two random values. Furthermore, we have $h = h_{\pi_0^{-1}(i_0),\pi_1^{-1}(i_1)}$, which implies that the response message is $e = e_{\pi_0^{-1}(i_0),\pi_1^{-1}(i_1)}$, which is an encryption of m_{i_0,i_1}. Since i_0, i_1 are two random values, we can just simulate e by samples two random values $i^*_0, i^*_1 \in [\sqrt{n}]$ and encrypts $m^*_{i^*_0,i^*_1}$ to e^*, which guarantees that e^* and e are indistinguishable. Moreover, since $m^*_{i^*_0,i^*_1}$ and m_{i_0,i_1} are indistinguishable, $r^* = m^*_{i^*_0,i^*_1} \oplus z$ and $r = m_{i_0,i_1} \oplus z$ are indistinguishable. This implies that $\mathsf{vc}_{\mathsf{oprf}} = \mathsf{Sim}_{\mathsf{oprf}}(k, r)$ and $\mathsf{vc}^*_{\mathsf{oprf}} = \mathsf{Sim}_{\mathsf{oprf}}(k, r^*)$ are indistinguishable.

$\mathsf{Client\text{-}Privacy}$. Let $\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, k)$ be the view of the server. We need to construct a PPT simulator Sim taking DB as input such that

$$\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, k) \approx_c \mathsf{Sim}(\mathsf{DB}).$$

Note that $\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, k)$ consists of

$$(\mathsf{DB}, \{(t_{j_0,j_1}, c_{j_0,j_1})\}_{j_0,j_1\in[\sqrt{n}]},\\ \{(h_{j_0,j_1}, e_{j_0,j_1})\}_{j_0,j_1\in[\sqrt{n}]}, \mathsf{query}, \mathsf{vc}_{\mathsf{oprf}}, b),$$

where $\mathsf{query} = q$ is the query message received from the client in the online phase, and $\mathsf{vc}_{\mathsf{oprf}}$ is the view of the server in the OPRF protocol. Since the client offers no inputs in the offline phase, Sim can simulate the messages $(\{(t_{j_0,j_1}, c_{j_0,j_1})\}_{j_0,j_1\in[\sqrt{n}]}, \{(h_{j_0,j_1}, e_{j_0,j_1})\}_{j_0,j_1\in[\sqrt{n}]})$ perfectly. Moreover, since the server has no outputs in the OPRF protocol, Sim can simulate $\mathsf{vc}_{\mathsf{oprf}}$ by invoking the OPRF simulator on mk and let $\mathsf{vc}^*_{\mathsf{oprf}}$ be the output. The security of the OPRF protocol guarantees that $\mathsf{vc}^*_{\mathsf{oprf}}$ and $\mathsf{vc}_{\mathsf{oprf}}$ are indistinguishable. To simulate the message query, Sim performs as follows.

1. If $b = 0$, sample a random value $q \in \mathcal{V}$.

2. Otherwise, choose two random values $i^*_0, i^*_1 \in [\sqrt{n}]$ and compute $q^* = h^{1/\alpha}_{i^*_0,i^*_1}$.

We show that q and q^* are indistinguishable. First, consider the case of $b = 1$, which implies that $k = k'_{i_0,i_1}$ for some $i_0, i_1 \in [\sqrt{n}]$. In this case, the query message in the real view is $q = \mathsf{H}(k)^\beta = \mathsf{H}(k'_{i_0,i_1})^\beta = h^{1/\alpha}_{\pi_0^{-1}(i_0),\pi_1^{-1}(i_1)}$, and the simulated query message is $q^* = h^{1/\alpha}_{i^*_0,i^*_1}$ for random $i^*_0, i^*_1 \in [\sqrt{n}]$. Note that π_0 and π_1 are two random permutations over $[\sqrt{n}]$ that are unknown to the server. Therefore, $\pi_0^{-1}(i_0), \pi_1^{-1}(i_1)$ are two random values of $[\sqrt{n}]$ and q and q^* are indistinguishable.

Now, consider the case of $b = 0$, which implies that $k \neq k_{j_0,j_1}$ for any $j_0, j_1 \in [\sqrt{n}]$. In this case, the real query message q is $\mathsf{H}(k)^\beta$, where β is sampled by the client. And the simulated query message q^* is a random element of $\mathcal{V}$. Since β is unknown to the server, $\mathsf{H}(k)^\beta$ is indistinguishable from a random element. Therefore, q and q^* are indistinguishable. $\square$

Complexity Analysis. We analyze the asymptotic complexity of $\Pi_{\mathsf{PaiCKSPIR}}$ from four aspects: communication, computation, bandwidth, and storage.

Let us first consider the offline phase. In the offline phase, the parties need to send $4n$ group elements, n plaintexts, and $3n$ ciphertexts. Hence, the communication cost is $\tilde{O}(n)$. In addition, the server needs to compute H and F n times, n Xors and exponentiations. Moreover, the client needs to compute $2n$ exponentiations, $2n$ encryptions, and n decryptions. Overall, the offline computational cost is $\tilde{O}(n)$. Furthermore, since each message sent by the parties contains $\sqrt{n}$ group elements

Table 4: Performance of PaiKSPIR and PaiCKSPIR on $n = 2^{20}, 2^{22}, 2^{24}$ database sizes and $64, 128, 256$ bytes entry sizes. “Comm.” stands for communication cost. Offline costs are the whole preprocessing. Online costs are amortized over 1000 queries. Data for PaiKSPIR are from Table 3.

		$n = 2^{20}$			$n = 2^{22}$			$n = 2^{24}$		
		64 bytes	128 bytes	256 bytes	64 bytes	128 bytes	256 bytes	64 bytes	128 bytes	256 bytes
Offline Comm. (MB)	PaiKSPIR	624.359	1008.581	1777.023	2496.443	4032.715	7105.260	9985.771	16130.862	28421.042
	PaiCKSPIR	432.0	688.0	1200.0	1728.0	2752.0	4800.0	6912.0	11008.0	19200.0
Offline Time (s)	PaiKSPIR	16.218	17.331	20.157	60.437	67.729	78.297	249.635	275.815	308.757
	PaiCKSPIR	33.311	33.525	35.030	119.743	123.880	128.781	470.224	487.166	516.738
Online Comm. (KB)	PaiKSPIR	0.391	0.578	0.953	0.391	0.578	0.953	0.391	0.578	0.953
	PaiCKSPIR	0.172	0.234	0.359	0.172	0.234	0.359	0.172	0.234	0.359
Online Time (ms)	PaiKSPIR	1.391	1.436	1.444	1.045	1.031	1.071	1.055	0.989	1.008
	PaiCKSPIR	0.883	0.906	0.878	0.671	0.673	0.670	0.634	0.649	0.617
Online Client-Side Storage (MB)	PaiKSPIR	4.931	5.161	4.950	4.760	4.990	4.778	4.758	4.988	4.776
	PaiCKSPIR	5.139	4.921	5.156	4.969	4.749	4.985	4.969	4.753	4.985

and $\sqrt{n}$ plaintexts or ciphertexts, the bandwidth of the protocol is $\tilde{O}(\sqrt{n})$. Finally, it is easy to see that the server-side and client-side storage are $\tilde{O}(n)$ and $\tilde{O}(\sqrt{n})$, respectively.

Now, we consider the online phase. Note that the OPRF protocol (described in Section 2.3) requires the parties to send two group elements and compute the hash function one time and three exponentiations, hence both the communication and computational costs are $\tilde{O}(1)$. In addition to executing the OPRF protocol, the parties need to send a group element and a ciphertext (as well as a bit b), which requires $\tilde{O}(1)$ communication. Moreover, the parties need to compute the hash function H one time and two exponentiations, which consumes $\tilde{O}(1)$ amount of computation. Finally, it is easy to see that the bandwidth of the online phase of the protocol is $\tilde{O}(1)$, and the server-side and client-side storage are $\tilde{O}(n)$ and $\tilde{O}(1)$, respectively.

A.1 Implementation and Evaluation

We also implement PaiCKSPIR. Since there is no known implementation of CKSPIR, and any KSPIR protocol can be converted to KSPIR by having the client send an additional bit, we directly compare our CKSPIR protocol to KSPIR. Since we are more concerned about the online complexity of the protocol, we only need to compare it with our PaiKSPIR (which has lower online complexity than PianoKSPIR and SpamKSPIR). The experimental results are shown in Table 4. We use the experimental setup that has been described in Section 6.

Evaluation. The experimental results show that for online communication, PaiCKSPIR is 2.3 - 2.7× better than PaiKSPIR, and for online time, PaiCKSPIR is 1.5 - 1.7× better than PaiKSPIR. The main reason why PaiCKSPIR has better online efficiency than PaiKSPIR is that in the general transformation from PIR to KSPIR (described in Section 5.2), a single keyword query requires the client to make $v = 3$ index queries. For the offline phase, since PaiKSPIR uses our PIR protocol with a database containing $b = 1.5n$ entries, PaiCKSPIR has a lower offline communication, which is 1.4 - 1.5× better than PaiKSPIR. Moreover, since that PaiCKSPIR uses much more public-key operations (i.e., exponentiations), the offline time of PaiCKSPIR is 1.6 - 2.1× higher than PaiKSPIR.