Incremental Single-Server Private Information Retrieval

Pengfei Lu[1,2], Guangwu Xu[1,2,3,4] (✉), Zengpeng Li[1,2] (✉), Mei Wang[1,2], and Haoyu Cui[1,2]

[1] School of Cyber Science and Technology, Shandong University, Qingdao 266237, China
{PengfeiLu,cuihaoyu}@mail.sdu.edu.cn
[2] State Key Laboratory of Cryptography and Digital Economy Security, Shandong University, Qingdao 266237, China
{gxu4sdq,wangmeiz}@sdu.edu.cn
[3] Shandong Institute of Blockchain, Jinan 250101, China
[4] Quan Cheng Laboratory, Jinan 250101, China
zengpengliz@gmail.com

Abstract. Incremental preprocessing in private information retrieval (PIR) schemes refers to handle insertions, modifications, and deletions to the database without requiring complete preprocessing after each update. This broadens the applicability of PIR in practical scenarios. However, two major issues remain: the concept of incremental preprocessing for the single-server PIR is still not established, and the row-level update strategy (iSimplePIR (Row-level)) introduces excessive unnecessary overhead. This paper aims to efficiently extend incremental preprocessing to the single-server setting. To our knowledge, we are the first to propose the formal definition of single-server incremental PIR. Besides, we construct an entry-level incremental scheme (iSimplePIR (Entry-level)) based on SimplePIR (USENIX '23). iSimplePIR (Entry-level) supports real-time updates of individual entries, as well as optimization of communication for scenarios with certain update cycles by incorporating a row aggregation mechanism. For a 1% column-major update in a 1GB database, iSimplePIR (Entry-level) achieves a 224× reduction in preprocessing computation overhead and a 4.2× reduction in both communication and monetary costs compared to iSimplePIR (Row-level). When applied to password breach detection with completely random entry updates, iSimplePIR (Entry-level) reduces preprocessing time by 86×. Meanwhile, our method can be combined with various SimplePIR-based schemes to reduce preprocessing costs, such as DoublePIR, Authenticated PIR (based on the LWE assumption), VeriSimplePIR, and YPIR.

Keywords: Single-server private information retrieval · Incremental preprocessing · Password breach detection.

1 Introduction

Personal privacy has become a fundamental concern across various online platforms and services. Private information retrieval (PIR) [15] is a cryptographic

primitive designed to protect individual privacy, allowing a client to retrieve an entry from a database without revealing to the database server which entry was accessed. PIR has been suggested for various applications, including anonymous communication [3, 39], privacy-preserving media streaming [24], private contact discovery [27], privacy-friendly advertising [4, 23], password check [28, 43] and private navigation [21, 47], among others.

Although PIR schemes are functionally powerful, they inherently incur high computational cost, as the server must process every entry in the database to respond to even a single client query. Otherwise, the scheme would leak information about which records the client is not interested in. To optimize online computation under this constraint, Beimel *et al.* [6] introduce the *preprocessing* model. SimplePIR [26] (concurrent work FrodoPIR [19]), the current state-of-the-art single-server PIR scheme based on cryptographic primitives, performs preprocessing on the database to generate a query-independent "*hint*". This preprocessing enables the PIR server to shift most of its computational workload to an offline phase that occurs prior to client queries, thereby significantly reducing the online computation cost.

Many applications that employ PIR naturally involve updates to their underlying content. For example, in password breach detection [30, 43, 44], the client aims to check whether their password appears in a leaked password database without revealing their query to the server. During the transition from an old version of the password database to a new one, new passwords may be added or existing ones modified. We observe that the hint depends on all entries in the database. As a result, once the database is updated, the previous hint becomes invalid for the PIR recover phase. This necessitates updating the hint whenever the database is changed.

Ma *et al.* [33] propose the concept of *incremental preprocessing*, which allows the server to preprocess only the updated portions of the database-rather than preprocessing the database from scratch-when existing entries are modified or deleted, or new entries are inserted. SimplePIR [26, Appendix C.3] mentions the idea of incremental preprocessing: when some entries within a row of the database are modified, the server directly multiplies the entire row with a public matrix and sends the result to the client, then the client obtains a new hint after performing local processing. Similarly, the single-server authenticated PIR scheme (APIR) [16, Section 5.2] based on the Learning with Errors (LWE) assumption [42] also discusses incremental updates in the case of content modifications. However, existing research has the following issues:

- The incremental construction in [33] imposes limitations on the number of modified and deleted entries and fails to retain the efficient online operations of the original non-incremental scheme. Notably, Ma *et al.* only provide the definition and construction of incremental preprocessing in a multi-server setting. Multi-server PIR relies on the assumption of multiple non-colluding servers, which is impractical in real-world applications [32]. This makes extending incremental preprocessing to single-server setting meaningful.

- SimplePIR [26] and APIR [16] mention incremental preprocessing as an extension not elaborating on it. Noting that, when a single entry requires immediate updating or the number of updated entries within a certain update cycle is less than a full row, the row-level preprocessing in SimplePIR results in significant unnecessary computational and communication overhead.
- Furthermore, in practical scenarios, modifications or deletions typically occur at the entry level rather than affecting an entire row at once or within a realistic time window. Consequently, treating incremental preprocessing at the granularity of entire rows possesses substantial limitations that greatly affect the practicality of the scheme.

1.1 Our Contributions

This paper extends the existing PIR scheme to support dynamically updatable databases by introducing the concept of incremental preprocessing in the single-server setting. We present a detailed description of the row-level incremental update strategy in SimplePIR, referred to as iSimplePIR (Row-level), and propose an entry-level incremental scheme called iSimplePIR (Entry-level)[1].

On one hand, iSimplePIR (Entry-level) supports real-time incremental preprocessing for individual entries, enabling efficient instant updates. On the other hand, for applications with periodic updates, where frequent small transmissions may incur high communication cost, we incorporate a row aggregation mechanism into iSimplePIR(Entry-level). In the incremental preprocessing of our scheme, the most time-consuming multiplication is performed between the entry-related difference and the corresponding row of the public matrix, rather than between an entire row and the full matrix. Furthermore, there is no limitation on the number of modified or deleted entries.

Our entry-level incremental construction seamlessly integrates with most SimplePIR-based schemes, such as DoublePIR (the recursive variant) [26], APIR (the authenticated version) [16], VeriSimplePIR (the verifiable version) [14], and YPIR [38], significantly improving their efficiency in handling database updates. By extending update granularity to entry-level, it provides a fundamental building block supporting diverse dynamic update patterns.

We conduct detailed and extensive experimental evaluations under three different scenarios. In the first scenario, entries are modified or deleted in a row-wise order (referred to as row-major), which clearly favors iSimplePIR (Row-level) and represents the worst-case scenario for our scheme. The second scenario involves updates in a column-wise order (column-major), which benefits iSimplePIR (Entry-level) and serves as its best case. For the third scenario, where entries are updated completely at random, we consider the application of password breach detection, modeling changes between two versions of the database as updates to a password database.

[1] The term "entry-level" refers to the granularity of individual database entries, not to the sophistication of the scheme.

Based on the online throughput and communication measurements across the three scenarios, our iSimplePIR (Entry-level) preserves the efficient online performance of SimplePIR. When 1% of a 1GB database is modified under the row-major pattern, iSimplePIR (Entry-level) achieves performance comparable to iSimplePIR (Row-level), demonstrating that our scheme incurs no extra overhead even in the worst case. In the column-major case, iSimplePIR (Entry-level) reduces the computation overhead of updating the hint by 224×, communication by 4.2×, and monetary cost by 4.2×. In the random-update setting modeled on the password breach detection application, iSimplePIR (Entry-level) achieves an 86× reduction in preprocessing computation cost and a 2.1× reduction in communication overhead compared to iSimplePIR (Row-level).

Our contributions are summarized as follows:

- We provide a formal definition of single-server incremental PIR.
- Our technique can be applied to reduce the preprocessing cost of most protocols in the SimplePIR family. These schemes cover aspects such as integrity and verifiability, and when combined with iSimplePIR (Entry-level), we can extend their applicability to a broader range of scenarios requiring stronger security and involving frequently updated databases.
- In both predefined settings, as well as in the application scenario of randomly updated entries in password leakage detection, our iSimplePIR (Entry-level) significantly outperforms prior row-level update strategies in terms of preprocessing computation and communication overhead.

1.2 Related Work

This section reviews relevant PIR schemes and various approaches to handling database updates.

Two classes of PIR schemes. PIR exists in two flavors: single-server PIR [1, 2, 8, 10, 11, 17, 19, 20, 22, 26, 29, 31, 34, 37, 40, 41, 45] and multi-server PIR [12, 15, 18, 25]. In multi-server PIR, the database is replicated across two or more non-colluding servers. These schemes are challenging to deploy due to their reliance on coordination among multiple independent infrastructure providers. Moreover, their security is vulnerable, as it depends entirely on the non-collusion assumption. In contrast, single-server PIR schemes are based on cryptographic hardness assumptions and do not rely on non-collusion.

SimplePIR-based schemes. The preprocessing model has been further explored in subsequent work [14, 18, 19, 26, 38, 48]. DoublePIR [26] invokes SimplePIR on a database formed by concatenating the hint matrix and response vector, reducing the client-side hint download size at the cost of increased server computation and query size. APIR (the construction based on the LWE assumption [16, Construction 2]) is an authenticated version of the SimplePIR scheme. VeriSimplePIR [14] is a verifiable version of the SimplePIR scheme. YPIR [38], based on SimplePIR, achieves high throughput while maintaining no offline communication. In the offline phase, the server still computes the hint to generate responses. A detailed description is given in Section 5.

Table 1: Comparison of different schemes for handling database updates. Update semantics refers to whether the scheme provides a formal definition of incremental PIR. Scheme applicability refers to the extent to which the incremental construction can be combined with other schemes.

Schemes	Servers	Update granularity	Update semantics	Scheme applicability
Checklist [28]	Multi-server	Bucketed database	✗	Low
Ma *et al.* [33]	Multi-server	Entry-level	✓	Medium
SimplePIR [26]	Single-server	Row-level	✗	Low
APIR [16]	Single-server	Row-level	✗	Low
iSimplePIR (Entry-level)	Single-server	Entry-level	✓	High

Schemes for handling database updates. Table 1 summarizes the differences between our scheme and several relevant prior schemes. Checklist [28] propose a scheme supporting insertions by dividing the database into buckets with exponentially growing capacities. The initial data is stored in the largest bucket, while new entries are added to the smallest bucket. Most updates only affect smaller buckets, but preprocessing still needs to be redone from scratch. Additionally, the scheme requires the client to query all $\log N$ buckets, where N is the size of database, significantly increasing online communication. Otherwise, it risks revealing which bucket contains the desired entry. To avoid complete preprocessing of the aforementioned buckets, Ma *et al.* [33] introduce the notion of *incremental preprocessing* in the multi-server setting. They leverages *pseudorandom set* (PRS) to extend the Corrigan-Gibbs and Kogan two-server scheme (CK) [18], obtaining incremental CK (iCK). However, the complex extension construction causes the iCK's online query phase to fail to inherit the characteristics of the original scheme. Specifically, the client sends plaintext indices instead of cryptographic keys, leading to asymptotically less succinct online communication. Moreover, in [33], the number of modified or deleted entries M is strictly limited (the product of M and the number of update sets must remain sublinear in N). These constraints limit the scalability of their approach in practical applications.

2 System Model and Definitions

Notations. For a finite set S, $x \xleftarrow{R} S$ denotes sampling an element x uniformly at random from S. For $x \in \mathbb{N}$, we use $[x]$ to represent the set $\{1, \ldots, x\}$. We denote the integers modulo p as $\mathbb{Z}_p$. A database DB contains N entries, where $\mathsf{DB}[i]$ represents the i-th entry in DB. Bold uppercase letters denote matrices and bold lowercase letters denote vectors. We use $u_{i,j}$ and $v_{i,j}$ to denote modified and deleted entries, respectively, where i and j represent the row and column indices. Let $\mathbf{A}[j]$ denote the j-th row of $\mathbf{A}$.

2.1 Single-Server Preprocessing PIR

We now revisit the formal definition of a single-server preprocessing PIR scheme [26]. The main difference between this type of PIR and standard PIR [15] lies in the offline phase, where the database can be preprocessed to produce the hint.

Definition 1 (Single-server preprocessing PIR [26]) *A single-server PIR scheme with preprocessing consists of the following four algorithms:*

- $h \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{DB})$*: On input a security parameter* λ*, and a database* DB*, the Setup algorithm outputs the hint* h*.*
- $(st, q) \leftarrow \mathsf{Query}(i)$*: On input an index* i*, the Query algorithm outputs a client secret* st *and a query* q*.*
- $a \leftarrow \mathsf{Answer}(\mathsf{DB}, q)$*: On input a database* DB *and a query* q*, the Answer algorithm outputs an answer* a*.*
- $d_i \leftarrow \mathsf{Recover}(st, h, a)$*: On input a client secret* st*, an answer* a*, and the hint* h*, the Recover algorithm outputs a database record* d_i*.*

A single-server preprocessing PIR scheme should satisfy the correctness, query privacy, and non-triviality.

Correctness. When the client and server honestly execute the PIR protocol, the client should retrieve the desired database entry with all but negligible probability. Formally, for λ, $N \in \mathbb{N}$ and any $i \in [N]$, we require that the following probability is at least $1 - \mathsf{negl}(\lambda)$:

$$Pr\left[d_i = \mathsf{DB}[i] : \begin{array}{r} h \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{DB}) \\ (st, q) \leftarrow \mathsf{Query}(i) \\ a \leftarrow \mathsf{Answer}(\mathsf{DB}, q) \\ d_i \leftarrow \mathsf{Recover}(st, h, a) \end{array}\right],$$

where $\mathsf{negl}(\cdot)$ is a negligible function and the probability is determined by the random choices of all algorithms.

Query Privacy. A query should not reveal any information about the item the client desires, meaning that any two queries should be computationally indistinguishable to the server. For λ, $N \in \mathbb{N}$ and all $i, j \in [N]$, we define the following distribution:

$$\mathsf{P}_i := \left\{ q_i : \begin{array}{r} h \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{DB}) \\ (st, q_i) \leftarrow \mathsf{Query}(i) \end{array} \right\}.$$

The query privacy property of a single-server preprocessing PIR holds if for any PPT adversary $\mathcal{A}$,

$$|Pr[\mathcal{A}(1^\lambda, \mathsf{P}_i) = 1] - Pr[\mathcal{A}(1^\lambda, \mathsf{P}_j) = 1]| \leq \mathsf{negl}(\lambda).$$

Non-triviality. The PIR scheme is non-trivial, meaning its communication overhead should be less than downloading the entire database. The combined size of the hint h, query q, and response a should be sublinear in N.

2.2 Defining Single-Server Incremental PIR

The key difference from the two-server incremental PIR definition [33] is that the number of additional algorithms is reduced from four to two. In single-server PIR, the preprocessing phase lacks an offline server role, so we adaptively reduce the number of interaction rounds between the client and server.

When the number of changes to the database within a given time is M (e.g., adding new entries, modifying, or deleting existing entries), the client and server do not need to rerun the Setup phase. The two additional algorithms perform incremental adjustments to the current hint, ensuring that the update cost is proportional to M rather than $N+M$. The incrementally adjusted hint integrates smoothly with the updated database, enabling efficient queries to proceed in the online phase.

Definition 2 (Single-server incremental PIR) *A single-server incremental PIR scheme consists of six algorithms. In addition to the four algorithms from Definition 1, two new algorithms are defined as follows:*

- $(\mathsf{DB}', \beta) \leftarrow \mathsf{DBUpdate}(\mathsf{DB}, \mathsf{op})$*: On input the original database* DB *and a series of operations* op*, the DBUpdate algorithm outputs an updated database* DB' *and a summary* β *of all the updates.*
- $h' \leftarrow \mathsf{StateUpdate}(h, \beta)$*: On input a hint* h *and a summary* β*, the StateUpdate algorithm outputs an updated hint* h'*.*

A single-server incremental PIR scheme with preprocessing should satisfy the correctness, query privacy and non-triviality.

Correctness. For λ, $N^{'}$ (the size of $\mathsf{DB}^{'}$) $\in \mathbb{N}$ and any $i \in [N^{'}]$, we require that the following probability is at least $1 - \mathsf{negl}(\lambda)$:

$$Pr\left[d_i = \mathsf{DB}^{'}[i] : \begin{array}{r} h \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{DB}) \\ (\mathsf{DB}^{'}, \beta) \leftarrow \mathsf{DBUpdate}(\mathsf{DB}, \mathsf{op}) \\ h^{'} \leftarrow \mathsf{StateUpdate}(h, \beta) \\ (st, q) \leftarrow \mathsf{Query}(i) \\ a \leftarrow \mathsf{Answer}(\mathsf{DB}^{'}, q) \\ d_i \leftarrow \mathsf{Recover}(st, h^{'}, a) \end{array} \right],$$

where $\mathsf{negl}(\cdot)$ is a negligible function and the probability is determined by the random choices of all algorithms.

Query Privacy. For λ, $N^{'} \in \mathbb{N}$ and all $i, j \in [N^{'}]$, we define the following distribution:

$$\mathsf{P}_i^{'} := \left\{ q_i : \begin{array}{r} h \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{DB}) \\ (\mathsf{DB}^{'}, \beta) \leftarrow \mathsf{DBUpdate}(\mathsf{DB}, \mathsf{op}) \\ h^{'} \leftarrow \mathsf{StateUpdate}(h, \beta) \\ (st, q_i) \leftarrow \mathsf{Query}(i) \end{array} \right\}.$$

The query privacy property of a single-server incremental PIR holds if for any PPT adversary $\mathcal{A}$,

$$|Pr[\mathcal{A}(1^\lambda, \mathsf{P}_i^{'}) = 1] - Pr[\mathcal{A}(1^\lambda, \mathsf{P}_j^{'}) = 1]| \leq \mathsf{negl}(\lambda).$$

Non-triviality. The size of the original hint should be sublinear in N. Similarly, the sizes of the updated hint, query q, response a, and update summary β should be sublinear in $N^{'}$. Crucially, the computational cost of incremental updates should be proportional to M, the number of update operations in op, while remaining sublinear in N.

2.3 Leakage Model

Leakage Function. We define the leakage function $\mathcal{L}$ to capture the information revealed by the update summary β during the incremental update process. For a sequence of update operations op applied to the database, we define:

$$\mathcal{L}(\mathsf{op}) = \Big(|\mathsf{op}|,\ \mathsf{type}(\mathsf{op}),\ \mathsf{pos}(\mathsf{op}),\ \mathsf{struct}(\mathsf{op})\Big)$$

where $|\mathsf{op}|$ denotes the number of updates, $\mathsf{type}(\mathsf{op})$ denotes the type of each update (insertion, modification, deletion), $\mathsf{pos}(\mathsf{op})$ denotes the positions of updated elements at the granularity revealed by the scheme, i.e., either entry indices (i, j) or row indices i after aggregation, $\mathsf{struct}(\mathsf{op})$ denotes the structural information of updates, including aggregated update vectors (e.g., $\mathbf{u}'_i$, $\mathbf{v}'_i$), among others.

Update Privacy. We require that the incremental update process does not reveal any information beyond the leakage function $\mathcal{L}$. Formally, for any PPT adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that:

$$\mathcal{A}(\beta) \approx_c \mathcal{S}(\mathcal{L}(\mathsf{op}))$$

where β is the update summary generated by $\mathsf{DBUpdate}$, and $\approx_c$ denotes computational indistinguishability.

Compared with row-level update mechanisms, the proposed entry-level construction reduces the granularity of leakage, revealing only fine-grained update information instead of entire row access patterns.

Discussion on Leakage Acceptability. The leakage captured by $\mathcal{L}$ is inherent to incremental preprocessing PIR schemes, as completely hiding update patterns would require reprocessing the entire database. In practical applications such as password breach detection and private blacklist queries, the leakage of update size and coarse-grained update patterns is typically acceptable [33, 43, 44]. This is because:

- The queried index is still protected by PIR query privacy,
- Update patterns often correlate with publicly observable system behavior (e.g., periodic database refresh).

Therefore, our leakage profile achieves a practical balance between efficiency and privacy, which is consistent with prior work on incremental PIR and update-efficient cryptographic systems [28, 33].

2.4 Types of Database Updates

In this paper, we focus on three types of database updates: the insertion of new entries, modification of existing entries, and deletion.

Insertion. Similar to the two-server incremental model, we also only support the insertion of new entries at the end of the database. This is because if new entries are added at arbitrary positions, the insertion of a single entry would affect the indices of all subsequent entries, requiring the entire database to be preprocessed from scratch.

Modification. Modification refers to changing the content of existing entries without altering the size of the database.

Deletion. If the client has retrieved the entry from the database before it is deleted, the client may have a copy, making it impossible to prevent further access to that entry. Therefore, in the deletion definitions below, we assume the client has not explicitly obtained the specific entry previously. We provide the definitions of strong and weak deletion.

- *Strong deletion.* Strong deletion refers to the situation where, after an entry is deleted, it is guaranteed that the client cannot retrieve the entry again.
- *Weak deletion.* Weak deletion refers to the situation where new clients cannot retrieve any deleted entries.

Strong deletion is the ideal property for practical applications. The server deletes certain entries and ensures that clients who have not previously retrieved them can no longer access them. In a preprocessing-based PIR scheme, an existing client stores a hint that implicitly encodes information about all database entries. Even if the client has not explicitly queried a deleted entry in the past, it can still retrieve the deleted entry indirectly by querying other entries and utilizing the existing hint. Consequently, achieving strong deletion is particularly challenging in the preprocessing model. We discuss in the following section that weak deletion is achievable.

We describe several direct methods for handling database updates, along with their potential limitations, in Appendix A.2.

3 Background: The SimplePIR Scheme

3.1 SimplePIR

We describe the SimplePIR [26] scheme, with its security relying on the hardness of the LWE problem [42]. A detailed explanation of the LWE problem is provided in Appendix A.1. The construction parameters of SimplePIR include the database size N, LWE parameters (n, q, χ), a plaintext modulus $p \ll q$, and a LWE matrix $\mathbf{A} \in \mathbb{Z}_q^{\sqrt{N} \times n}$. The database of N entries is treated as a matrix $\mathbf{D} \in \mathbb{Z}_p^{\sqrt{N} \times \sqrt{N}}$. Specifically, SimplePIR consists of the following algorithms:

- $\mathbf{H} \leftarrow \mathsf{Setup}(1^\lambda, \mathbf{D})$: The server computes $\mathbf{H} \leftarrow \mathbf{D} \cdot \mathbf{A} \in \mathbb{Z}_q^{\sqrt{N} \times n}$ and the client downloads and stores the hint $\mathbf{H}$.

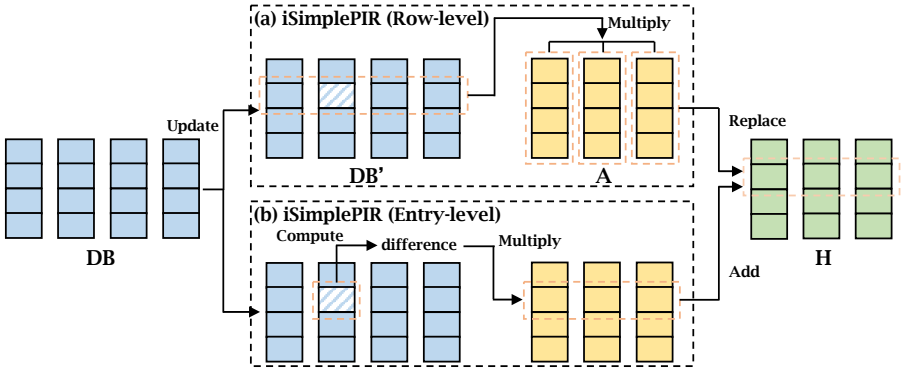


Fig. 1: An illustration of incremental preprocessing. Processes (a) and (b) depict row-level and entry-level incremental SimplePIR, respectively. The patterned cell in the figure indicates a modified entry.

- $(\mathsf{st}, \mathbf{q}) \leftarrow \mathsf{Query}(i)$: The client writes $i \in [N]$ as $(i_r, i_c) \in [\sqrt{N}]^2$, samples $\mathbf{s} \xleftarrow{R} \mathbb{Z}_q^n$ and $\mathbf{e} \leftarrow \chi^{\sqrt{N}}$, and computes $\mathbf{q} \leftarrow \mathbf{A}\mathbf{s} + \mathbf{e} + \Delta \cdot \mathbf{b}_{i_c} \in \mathbb{Z}_q^{\sqrt{N}}$, where $\Delta := \lfloor q/p \rfloor$ and $\mathbf{b}_{i_c}$ is the vector of all zeros but with 1 in the i_c^{th} position, and $(\mathsf{st}, \mathbf{q}) \leftarrow ((i_r, \mathbf{s}), \mathbf{q})$. The query $\mathbf{q}$ is sent to the server.
- $\mathbf{a} \leftarrow \mathsf{Answer}(\mathbf{D}, \mathbf{q})$: The server computes $\mathbf{a} \leftarrow \mathbf{D} \cdot \mathbf{q} \in \mathbb{Z}_q^{\sqrt{N}}$ and sends the response $\mathbf{a}$ to the client.
- $d_i \leftarrow \mathsf{Recover}(\mathsf{st}, \mathbf{H}, \mathbf{a})$: The client parses $(i_r, \mathbf{s}) \leftarrow \mathsf{st}$ and computes $\hat{d}_i \leftarrow \mathbf{a}[i_r] - \mathbf{H}[i_r] \cdot \mathbf{s} \in \mathbb{Z}_q$, where $\mathbf{a}[i_r]$ and $\mathbf{H}[i_r]$ represent component i_r of $\mathbf{a}$ and row i_r of $\mathbf{H}$, respectively. Finally, the client outputs $d_i \leftarrow \left\lfloor \hat{d}_i \right\rceil / \Delta \in \mathbb{Z}_p$.

3.2 Row-Level Update Strategy for SimplePIR

In [26, Appendices C.3], SimplePIR treats changes to certain entries at the level of their respective rows, observing that only the corresponding rows in the hint $\mathbf{H}$ need to be updated. The preprocessing workload to be redone is linearly proportional to the number of rows whose contents have been updated, and it is not necessary to re-execute the entire Setup algorithm. This reflects the idea of row-level incremental preprocessing (iSimplePIR(Row-level)). However, the explanation in [26] is overly brief and lacks detail. Here, we provide a concrete procedure using a modification example to illustrate the process. And we present the procedure as Process (a) in Figure 1.

If the modified entries in $\mathbf{D}$ involve M_r rows, which after modification become $(\mathbf{u}_{t_1}, \ldots, \mathbf{u}_{t_{M_r}}) \in (\mathbb{Z}_p^{\sqrt{N}})^{M_r}$, then for $j \in [M_r]$, the server computes $\mathbf{u}'_{t_j} \leftarrow \mathbf{u}_{t_j} \cdot \mathbf{A} \in \mathbb{Z}_q^n$ and obtains the tuples $(\mathbf{u}'_{t_1}, \ldots, \mathbf{u}'_{t_{M_r}}) \in (\mathbb{Z}_q^n)^{M_r}$ along with their corresponding row indices $(t_1, \ldots, t_{M_r})$. The server sends these to the client, who then replaces the corresponding rows in $\mathbf{H}$ with $(\mathbf{u}'_{t_1}, \ldots, \mathbf{u}'_{t_{M_r}})$ according to $(t_1, \ldots, t_{M_r})$ to obtain the updated hint.

4 iSimplePIR: Single-Server Incremental PIR

4.1 Entry-Level Incremental SimplePIR Construction

We extend and construct an entry-level incremental SimplePIR scheme, referred to as iSimplePIR (Entry-level). This scheme supports efficient real-time updates while enabling communication-optimized row aggregation preprocessing within a defined update cycle. Taking the modification of an entry in the database as an example, we illustrate the process in Process (b) of Figure 1.

Specifically, we present the construction of iSimplePIR (Entry-level) in Figure 2. The detailed explanation is provided below for each algorithm:

Setup. In the offline preprocessing phase, the Setup algorithm remains consistent with the original scheme described in Section 3.1.

DBUpdate and StateUpdate. After generating the hint, if a series of operations, op, lead to updates in the database, the server and client execute the DBUpdate and StateUpdate algorithms, respectively, to incrementally update the current hint. We incorporate versioned hint management, where the server generates an incremental update package with an associated version number k for each update. The client verifies the order of incremental updates based on k and maintains the corresponding version number of the hint. Below, we provide a detailed description and analysis of three different update scenarios:

Insertion. We observe that $\mathbf{H}$ is the product of $\mathbf{D}$ and $\mathbf{A}$. Since the new entries are inserted at the end of the database and padded with zeros when they do not fill a complete row, the insertion case is handled using row-level incremental preprocessing. Assume the M newly inserted entries occupy M'_r rows. First, the server incrementally computes the product of each of the M'_r rows with $\mathbf{A}$, obtaining M'_r hint-related rows, which are then sent to the client. Next, the client incrementally updates the current hint by simply appending the received hint-related rows to the end of the existing hint.

Modification. We discuss from the perspective of a single modified entry. In entry-level real-time updates, assume that the entry at row i, column j is updated from $u_{i,j}$ to $u'_{i,j} = u_{i,j} + \gamma$. The server first computes the difference and sends it to the client along with the corresponding row and column indices. The client then multiplies the difference with the j-th row of $\mathbf{A}$, and adds the resulting intermediate value to the i-th row of $\mathbf{H}$.

We further observe that in schemes with a certain update interval, the number of modified entries within the same row may accumulate over time. When the number of accumulated updates is large, transmitting differences along with their corresponding indices incurs high communication overhead (see Section 6.1 for a detailed comparison). Therefore, we introduce a communication optimized row aggregation mechanism. Noting that changes to the i-th row of the database only affect the i-th row in the current hint, we proceed as follows. Let M' be the number of modified entries in the i-th row. The server's communication cost is approximately $M' \log p + (M'+1) \log \sqrt{N}$ bits under the entry-level strategy and $n \log q + \log \sqrt{N}$ bits under the row-level strategy. Let $t = \lceil \frac{n \log q}{\log p + \log \sqrt{N}} \rceil$. When $M' > t$, the server computes the differences corresponding to all modified entries

Construction 1: iSimplePIR (Entry-level). The construction parameters are consistent with the original scheme described in Section 3.1.
Offline preprocessing.

- $\mathbf{H} \leftarrow \mathsf{Setup}(1^\lambda, \mathbf{D})$: **Server** computes and outputs the hint $\mathbf{H} \leftarrow \mathbf{D} \cdot \mathbf{A} \in \mathbb{Z}_q^{\sqrt{N} \times n}$.
- $(\mathbf{D}', \beta) \leftarrow \mathsf{DBUpdate}(\mathbf{D}, \mathsf{op})$: **Server** performs the operations op on $\mathbf{D} \in \mathbb{Z}_p^{\sqrt{N} \times \sqrt{N}}$ to produce $\mathbf{D}'$:
  1. For entry insertions, when the newly added entries form a complete row $\mathbf{w}_i$, compute $\mathbf{w}_i' \leftarrow \mathbf{w}_i \cdot \mathbf{A} \in \mathbb{Z}_q^n$, attach version number k and set $\beta_{add} \leftarrow (\mathbf{w}_i', k)$; if they are insufficient to form a full row, pad with zeros.
     Entry-level real-time updates:
  2. For modification, compute $\gamma_{edit} \leftarrow u_{i,j}' - u_{i,j} \in \mathbb{Z}_p$ and set $\beta_{edit} \leftarrow (i, j, \gamma_{edit}, k)$.
  3. For deletion, compute $\gamma_{del} \leftarrow r - v_{i,j} \in \mathbb{Z}_p$, where $r \xleftarrow{R} \mathbb{Z}_p$. Set $\beta_{del} \leftarrow (i, j, \gamma_{del}, k)$.
     Within a given update interval, for each $i \in [\sqrt{N}]$, if M', the number of modified/deleted entries in row $\mathbf{u}_i/\mathbf{v}_i \in \mathbb{Z}_p^{\sqrt{N}}$ is less than or equal to t, the server applies the entry-level real-time updates described above. If $M' > t$:
     Communication-optimized row aggregation:
  4. For all modified entries $u_{i,j}$ in $\mathbf{u}_i$, compute $\gamma_{edit,j} \leftarrow u_{i,j}' - u_{i,j} \in \mathbb{Z}_p$, $\mathbf{u}_i' \leftarrow \sum_j \gamma_{edit,j} \cdot \mathbf{A}[j] \in \mathbb{Z}_q^n$ and set $\beta_{edit} \leftarrow (i, \mathbf{u}_i', k)$.
  5. For all deleted entries $v_{i,j}$ in $\mathbf{v}_i$, compute $\gamma_{del,j} \leftarrow r - v_{i,j} \in \mathbb{Z}_p$, where $r \xleftarrow{R} \mathbb{Z}_p$, $\mathbf{v}_i' \leftarrow \sum_j \gamma_{del,j} \cdot \mathbf{A}[j] \in \mathbb{Z}_q^n$ and set $\beta_{del} \leftarrow (i, \mathbf{v}_i', k)$.
  6. Set $\beta \leftarrow (\beta_{edit}, \beta_{del}, \beta_{add})$. Output $(\mathbf{D}', \beta)$.
- $\mathbf{H}' \leftarrow \mathsf{StateUpdate}(\mathbf{H}, \beta)$: **Client** parses β as $(\beta_{edit}, \beta_{del}, \beta_{add})$, incrementally updates the current hint $\mathbf{H}$, and outputs the updated hint $\mathbf{H}'$:
  1. For insertions, parse β_{add} as $(\mathbf{w}_i', k)$ and verify the k. Append $\mathbf{w}_i'$ to the end of $\mathbf{H}$.
     Entry-level real-time updates:
  2. For modification/deletion, parse $\beta_{edit/del}$ as $(i, j, \gamma_{edit/del}, k)$ and verify the k. Compute $\mathbf{x}_i \leftarrow \gamma_{edit/del} \cdot \mathbf{A}[j] \in \mathbb{Z}_q^n$ and add $\mathbf{x}_i$ to the i-th row of $\mathbf{H}$.
     Communication-optimized row aggregation:
  3. For modifications/deletions, parse $\beta_{edit/del}$ as $(i, \mathbf{u}_i'/\mathbf{v}_i', k)$ and verify the k. Add $\mathbf{u}_i'/\mathbf{v}_i'$ to the i-th row of $\mathbf{H}$.

  Online phase.
- The online phase is identical to the original scheme in Section 3.1, except the server responds based on $\mathbf{D}'$ and the client recovers the entry using $\mathbf{H}'$.

Fig. 2: The iSimplePIR (Entry-level) scheme.

in the i-th row, multiplies them by the relevant row of $\mathbf{A}$, and sums the results into an intermediate vector. The client then adds this vector directly to the i-th row of $\mathbf{H}$. Otherwise, the entry-level real-time update is applied. In summary,

this row aggregation strategy optimizes communication while preserving entry-level incremental preprocessing on the server side, rather than multiplying the entire row with $\mathbf{A}$ as in row-level strategy.

Deletion. From the perspective of a single deleted entry, suppose the entry $v_{i,j}$ at row i and column j is deleted. Note that $v_{i,j}$ is an entry that the server intends to delete and does not want the client to query; therefore, information directly related to $v_{i,j}$ cannot be transmitted. In entry-level real-time updates, $v_{i,j}$ needs to be replaced with a unified random object r. The specific operations of entry-level and row aggregation updates are similar to those for modifications.

We consider two cases: deletion for an honest client who follows the protocol, and deletion for a malicious client who may deviate arbitrarily. In both cases, we assume that the client has not queried the entry intended for deletion prior to this. The above approach does not work against malicious clients because they can retain copies of the differences sent by the server during deletion. In this scenario, a malicious client can query the server for the entry at row i and column j and directly recover $v_{i,j}$ using the retained copy.[2] In fact, secure deletion against malicious clients is impossible in SimplePIR. Our method targeting honest clients remains effective for malicious clients who join the protocol after the entry has been deleted, as they do not have any copies related to the deleted entries. The deletion discussed here is not meant to represent a strong adversarial model, but rather to analyze the level of deletion that can be realistically achieved.

Query, Answer, and Recover. In the online phase, the Query algorithm remains the same as in the original SimplePIR scheme. The Answer and Recover algorithms, except for the inputs being updated to the new database and new hint, stay consistent with the original scheme. Overall, after the incremental update in the preprocessing phase, the algorithms in the online phase remain consistent with the original scheme.

Theorem 3. *Under the LWE assumption, when instantiated with database size N, LWE parameters (n, q, χ), and random matrix $\boldsymbol{A} \xleftarrow{R} \mathbb{Z}_q^{\sqrt{N} \times n}$, our single-server iSimplePIR (Entry-level) scheme satisfies correctness, query privacy, and non-triviality (i.e., Definition 2).*

Proof (Sketch). We separately prove non-triviality, correctness, query privacy and update privacy.

Non-triviality. When M entries are modified or deleted, the computational cost of iSimplePIR (Entry-level) is $O(nM)$. When M new entries are inserted across M'_r rows, the cost becomes $O(nM'_r\sqrt{N})$. The non-triviality of iSimplePIR (Entry-level) follows from the original scheme [26] described in Section 3.1 and the analysis above.

Correctness. We prove correctness from the following two aspects: (1) the hint should be correctly incrementally updated; (2) the client should retrieve the

[2] For the communication-optimized row aggregation, a malicious client may leverage the received update-related data and a copy of the original hint to iteratively query and verify entries in order to recover deleted items.

desired entries during the online phase. On the one hand, consider the case where an entry $u_{i,j}$ is modified to $u'_{i,j}$. Observing that $\mathbf{H} = \mathbf{D} \cdot \mathbf{A}$, $\mathbf{H}' = \mathbf{D}' \cdot \mathbf{A} = \mathbf{D} \cdot \mathbf{A} + (\mathbf{D}' - \mathbf{D}) \cdot \mathbf{A}$, and that the modification to the i-th row of $\mathbf{D}$ only affects the i-th row of $\mathbf{H}$, we analyze this update from the perspective of the modified entry. The term $\mathbf{D} \cdot \mathbf{A} + (\mathbf{D}' - \mathbf{D}) \cdot \mathbf{A}$ is equivalent to adding the product of the entry-wise difference and the corresponding row of $\mathbf{A}$ to the corresponding row of $\mathbf{H}$. In the entry-level real-time update, the server computes $\gamma_{edit} \leftarrow u'_{i,j} - u_{i,j} \in \mathbb{Z}_p$, the client computes $\mathbf{x}_i \leftarrow \gamma_{edit/del} \cdot \mathbf{A}[j] \in \mathbb{Z}_q^n$, and adds $\mathbf{x}_i$ to the i-th row of $\mathbf{H}$ to obtain $\mathbf{H}'$. The communication-optimized row aggregation aggregates the modified entries in the i-th row and adds the result to the corresponding row of $\mathbf{H}$ to obtain $\mathbf{H}'$. From the above analysis, it is evident that $\mathbf{H}'$ equals the product of the updated database $\mathbf{D}'$ and $\mathbf{A}$. The case of deletion can be proven similarly. For insertion, the client directly appends the new row $\mathbf{w}'_i \in \mathbb{Z}_q^n$ to the end of $\mathbf{H}$ to obtain $\mathbf{H}'$, which clearly equals $\mathbf{D}' \cdot \mathbf{A}$ as well. On the other hand, the correctness of the online query phase follows directly from the original scheme (as proven in the first part of Theorem C.1 in [26]).

Query privacy. We prove query privacy from the following two aspects: (1) the incremental update of the hint should not reveal any information about the query index; (2) the queries in the online phase should appear indistinguishable. First, the incremental update does not involve the query index i, meaning the entire process is independent of i. Then, for databases $\mathbf{D}$ and $\mathbf{D}'$, we have the following two distributions:

$$\mathsf{P}_i := \left\{ \mathbf{q}_i : \begin{array}{r} \mathbf{H}' \leftarrow \mathsf{Setup}(1^\lambda, \mathbf{D}') \\ (st, \mathbf{q}_i) \leftarrow \mathsf{Query}(i) \end{array} \right\},$$

$$\mathsf{P}'_i := \left\{ \mathbf{q}_i : \begin{array}{r} \mathbf{H} \leftarrow \mathsf{Setup}(1^\lambda, \mathbf{D}) \\ (\mathbf{D}', \beta) \leftarrow \mathsf{DBUpdate}(\mathbf{D}, \mathsf{op}) \\ \mathbf{H}' \leftarrow \mathsf{StateUpdate}(\mathbf{H}, \beta) \\ (st, \mathbf{q}_i) \leftarrow \mathsf{Query}(i) \end{array} \right\}.$$

According to Theorem 9 in [33], P_i and P'_i are computationally indistinguishable. Finally, Theorem C.1 (second half) in [26] proves that distribution P_i is indistinguishable from a random distribution. Using the triangle inequality, it follows that the probability of any PPT adversary distinguishing queries for any indices i and j is negligible. Furthermore, Corollary C.3 in [26] proves that encryption remains secure even when the same matrix $\mathbf{A}$ is reused to encrypt multiple messages, as long as each ciphertext is generated with an independent secret vector and noise vector. In SimplePIR, the online phase allows all clients to construct their queries using the same matrix $\mathbf{A}$, and the row-level strategy in the offline phase also reuses $\mathbf{A}$. Our iSimplePIR (Entry-level) inherits this reuse of $\mathbf{A}$ to support incremental preprocessing.

Update privacy. We show that the information revealed during the incremental update process is limited to the leakage function $\mathcal{L}$ defined in Section 2.3. Specifically, the update summary β consists only of structured information about the updates, such as the number of modified entries, their types, and their positions

at a certain granularity. No information about the query index i is revealed. We construct a simulator $\mathcal{S}$ that, given only $\mathcal{L}(\mathsf{op})$, can generate a simulated update transcript β^* that is computationally indistinguishable from the real update summary β. This follows from the fact that all elements in β: (1) are independent of the query index i, and (2) reveal only structural information about the updates as captured by $\mathcal{L}(\mathsf{op})$. Therefore, for any PPT adversary $\mathcal{A}$, we have $\mathcal{A}(\beta) \approx_c \mathcal{S}(\mathcal{L}(\mathsf{op}))$.

The proof includes an analysis of the security implications of reusing $\mathbf{A}$ during incremental preprocessing. For the theoretical analysis, assuming a single entry is modified, the preprocessing computational cost in iSimplePIR (Entry-level) is $O(n)$. In contrast, iSimplePIR (Row-level) requires $O(n\sqrt{N})$ preprocessing computation (Table 2).

Client-side offline computation. In iSimplePIR (Entry-level), the multiplication operations during real-time updates are concentrated on the client side. Notably, the client computation in our incremental preprocessing is independent of N. This contrasts with PIANO [48] which also performs client-side offline computation, but incurs N-related overhead. Furthermore, its experiments demonstrate that client computation becomes the preprocessing bottleneck.

4.2 Optimizations and Extensions

We briefly propose two storage optimizations: (1) the server and client can communicate and store only a small seed to generate matrix $\mathbf{A}$, and (2) if the client knows in advance that it will only perform $Q \ll n$ queries, it can store the product of the hint and the corresponding secret key instead of the entire hint. We also consider an extension to a single-round [9, 46] setting where the client's queries remain unchanged before and after database updates. After the update, the server can incrementally compute a new response—at the cost of some additional storage—and return it to the client, significantly reducing the response computation cost. Detailed descriptions appear in the full version.

5 Four Case Studies

Due to space limitations, we only present the setup phase of the four case study schemes based on SimplePIR, and briefly describe how they are integrated with iSimplePIR (Entry-level), as well as the improvements in computation and communication efficiency after the integration.

DoublePIR Scheme [26]. Since SimplePIR operates on a database of elements from $\mathbb{Z}_p$ with the hint matrix and response vector in $\mathbb{Z}_q$, the server in DoublePIR computes the decomposition of these entries before invoking SimplePIR [26] (using Decomp for base-p decomposition). DoublePIR treats the database as a matrix $\mathbf{D}_{dou} \in \mathbb{Z}_p^{l \times m}$ with two LWE matrices $\mathbf{A}_1 \in \mathbb{Z}_q^{m \times n}$ and $\mathbf{A}_2 \in \mathbb{Z}_q^{l \times n}$, along with a scalar $k := \lceil \log q / \log p \rceil$.

Table 2: Comparison of operation counts for different schemes before and after combination with iSimplePIR (Entry-level). We count the number of integer multiplications '×' and additions '+'. The specific update scenario assumes that a single entry in one row of the database is modified.

Schemes	Before combination (#operations)	After combination (#operations)
SimplePIR [26]	$n\sqrt{N}\times$, $n\sqrt{N}+$	$n\times$, $n+$
DoublePIR [26]	$nm\times$, $nm+$	$n\times$, $n+$
APIR [16]	$n\sqrt{N}\times$, $n\sqrt{N}+$	$n\times$, $n+$
VeriSimplePIR [14]	$(nm+\lambda l)\times$, $(nm+\lambda l)+$	$(n+\lambda)\times$, $(n+\lambda)+$
YPIR [38]	$d_1 l_1\times$, $d_1 l_1+$	$d_1\times$, d_1+

– $(\mathbf{H}_s, \mathbf{H}_c) \leftarrow \mathsf{Setup}(1^\lambda, \mathbf{D}_{dou})$: The server computes $\mathbf{H}_s \leftarrow \mathsf{Decomp}(\mathbf{A}_1^\top \cdot \mathbf{D}_{dou}^\top) \in \mathbb{Z}_q^{kn\times l}$ and $\mathbf{H}_c \leftarrow \mathbf{H}_s \cdot \mathbf{A}_2 \in \mathbb{Z}_q^{kn\times n}$. And the client downloads and stores $\mathbf{H}_c$.

APIR Scheme [16]. In traditional PIR, the server is (implicitly) equivalent to the data owner. With the rise of cloud computing, it has become a trend to outsource storage/computation tasks to rented servers and explicitly separate the two. To defend against *selective-failure attacks*, where a malicious server deliberately corrupts specific entries to infer the client's query, the data owner in APIR (the construction based on the LWE assumption) generates a database digest as a commitment to its contents. After *rebalancing*, the database is viewed as a matrix $\mathbf{D}_{aut} \in \mathbb{Z}_2^{\sqrt{N}\times\sqrt{N}}$.

– $\mathbf{D}_c \leftarrow \mathsf{Setup}(1^\lambda, \mathbf{D}_{aut})$: The data owner computes the database digest $\mathbf{D}_c \leftarrow \mathbf{D}_{aut} \cdot \mathbf{A} \in \mathbb{Z}_q^{\sqrt{N}\times n}$ and the client downloads and stores $\mathbf{D}_c$.

VeriSimplePIR Scheme [14]. VeriSimplePIR constructs the verification procedure using an extractable Short Integer Solution (SIS)-based commitment [5]. The database is viewed as a matrix $\mathbf{D}_{ver} \in \mathbb{Z}_p^{l\times m}$. The hash function Hash is modeled as a random oracle, and its detailed implementation can be found in [14].

– $(\mathbf{H}_{ver}, \mathbf{Z}_{ver}) \leftarrow \mathsf{Setup}(1^\lambda, \mathbf{D}_{ver})$: The server computes the commitment $\mathbf{H}_{ver} \leftarrow \mathbf{D}_{ver} \cdot \mathbf{A}_1 \in \mathbb{Z}_q^{l\times n}$, $\mathbf{C}_{ver} = \mathsf{Hash}(\mathbf{A}_1, \mathbf{H}_{ver}) \in \{0,1\}^{\lambda\times l}$ and $\mathbf{Z}_{ver} \leftarrow \mathbf{C}_{ver} \cdot \mathbf{D}_{ver}$. The client downloads and stores $\mathbf{H}_{ver}, \mathbf{Z}_{ver}$.

YPIR Scheme [38]. The database is viewed as a matrix $\mathbf{D}_y \in \mathbb{Z}_p^{l_1\times l_2}$, where $d_i, m_i \in \mathbb{N}$ and $l_i = m_i d_i$ for $i \in \{1,2\}$. $\mathcal{R}_{d_j} := \mathbb{Z}[x]/\langle x^{d_j}+1\rangle$ and $\mathcal{R}_{d_j,q} := \mathcal{R}_{d_j}/q\mathcal{R}_{d_j}$ for $j \in \{1,2\}$. For integers $q > p$, let $\lfloor\cdot\rceil_{q,p}$ denote the rounding map that lifts $x \in \mathbb{Z}_q$ to its representative $x' \in (-q/2, q/2]$ and outputs $\lfloor (p/q)x' \rceil \in$

$\mathbb{Z}_p$. This operation is extended component-wise to matrices. For $f = \sum_{i=0}^{d-1} \alpha_i x^i$,

$$\mathsf{NCyclicMat}(f) := \begin{bmatrix} \alpha_0 & \alpha_1 & \cdots & \alpha_{d-1} \\ -\alpha_{d-1} & \alpha_0 & \cdots & \alpha_{d-2} \\ \vdots & \vdots & \ddots & \vdots \\ -\alpha_1 & -\alpha_2 & \cdots & \alpha_0 \end{bmatrix}.$$

NCyclicMat is extended to operate on vectors in a component-wise manner.

- $(\mathbf{H}_1, \mathbf{H}_2) \leftarrow \mathsf{Setup}(1^\lambda, \mathbf{D}_y)$: For $j \in \{1, 2\}$, the server samples $\mathbf{a}_j \xleftarrow{R} \mathcal{R}_{d_j, q_j}^{m_j}$ and sets $\mathbf{A}_j = \mathsf{NCyclicMat}(\mathbf{a}_j^\top) \in \mathbb{Z}_{q_j}^{d_j \times l_j}$. The server computes and outputs $\mathbf{H}_1 = \mathsf{Decomp}(\lfloor \mathbf{A}_1 \cdot \mathbf{D}_y \rceil_{q_1, \hat{q_1}}) \in \mathbb{Z}^{kd_1 \times l_2}$, $\mathbf{H}_2 = \mathbf{A}_2 \cdot \mathbf{H}_1^\top \in \mathbb{Z}_{q_2}^{d_2 \times kd_1}$.

Combination. We briefly describe how four case-study schemes are combined with iSimplePIR (Entry-level) and analyze the communication and computational costs before and after the integration (Table 2).

- *DoublePIR-iSimplePIR (Entry-level)*: It is observed that $\mathbf{H}_s$ is the product of the database and the public matrix. When the database is updated, we can apply the entry-level real-time updates and communication-optimized row aggregation from iSimplePIR (Entry-level) to the offline phase of DoublePIR.
- *APIR-iSimplePIR (Entry-level)*: The incremental update is applied directly to the digest.
- *VeriSimplePIR-iSimplePIR (Entry-level)*: Noticed that both $\mathbf{H}_{ver}$ and $\mathbf{Z}_{ver}$ are products of two matrices. After the database undergoes an update, they are directly combined with iSimplePIR (Entry-level), where the incremental preprocessing is first applied to obtain the updated $\mathbf{H}_{ver}$, followed by the calculation of $\mathbf{C}_{ver}$, and similarly, the incremental update is applied to $\mathbf{Z}_{ver}$.
- *YPIR-iSimplePIR (Entry-level)*: After an update, the server locally integrates iSimplePIR (Entry-level) to efficiently preprocess and obtain the updated $\mathbf{H}_1$.

As shown in Table 2, after the update, iSimplePIR (Entry-level) achieves a multiplicative reduction—dependent on the database dimension—in the numbers of multiplications and additions required in the preprocessing of the original SimplePIR, DoublePIR, APIR, and YPIR. For VeriSimplePIR, computational efficiency is improved for both matrices produced in the offline phase.

6 Implementation and Evaluation

Our experimental evaluation aims to answer the following four questions:

1. Under both row-major and column-major update scenarios, what are the computation and communication costs of iSimplePIR (Entry-level) compared to iSimplePIR (Row-level)?

2. How do the computation and communication costs of the iSimplePIR scheme vary with different update ratios and database sizes?
3. Under random update scenarios, how does iSimplePIR (Entry-level) perform when applied to password breach detection?

To address the above questions, we implement and evaluate both iSimplePIR (Entry-level) and iSimplePIR (Row-level). The underlying SimplePIR protocol [26] serves as the foundation of our work. We use the publicly available implementation of SimplePIR from [13] and extend it by incorporating the actual hint generation process.

We consider the following three scenarios to conduct a comprehensive and detailed comparison of the two schemes: (1) entries are updated in row-major order, which favors the iSimplePIR (Row-level) scheme due to updates filling complete rows; (2) entries are updated in column-major order, which favors the iSimplePIR (Entry-level) scheme as updates span across rows; and (3) entries are updated completely at random, as described in Section 6.2 in the context of the password breach detection application.

Baselines. Furthermore, we compare our iSimplePIR (Entry-level) with the following baselines: DoublePIR [26], the Spiral family [37], and MulPIR [1], which are recent single-server PIR schemes applied to password breach detection. We run their publicly available implementations [1, 26, 36].

Experimental setup. We implement iSimplePIR(Row-level) and iSimplePIR (Entry-level) in C/C++ with approximately 1000 lines of code[3]. We run our experiments on a single thread of a machine running Ubuntu 24.04, equipped with an Intel i7 processor operating at 3.3 GHz and 16 GB of RAM. All experiments are performed in the same benchmarking environment, and the iSimplePIR code is compiled using the `clang++` compiler (version 18) with the `-O3` option. We run each experiment 10 times and report averages from those 10 trials. Standard deviations are less than 10% of the reported means.

Parameters. Both row-level and entry-level incremental SimplePIR schemes are benchmarked with lattice dimension $n = 2^{10}$, modulus $q = 2^{32}$, and a choice of p to minimize online communication. It is observed that the original SimplePIR experiments [26, Table 8] use one bit database entries. All database entries in these microbenchmark experiments are set to one bit, while larger entries in application scenarios are handled using the extension method described in [26, Section 4.3]. As analyzed in [14], under compact packing, entry size has negligible impact, whereas total database size significantly affects parameters. Thus, our microbenchmarks are based on total database size.

6.1 Microbenchmarks

After database updates, we conduct microbenchmarks to compare the offline and online performance of iSimplePIR (Entry-level) under both row-major and column-major settings against iSimplePIR (Row-level). Following the two-server

[3] The full source code link is available on GitHub later.

Table 3: Microbenchmarks of iSimplePIR (Row-level) and iSimplePIR (Entry-level) under two different scenarios, where 1% of a 1GB database is modified. Each database entry is one bit. "comp. (comm.)" denotes computation (communication) overhead. Throughput is the ratio of the database size to the server's query response time. We estimate the service costs based on prevailing AWS pricing: \$0.09 per GB of outbound data and $\$1.5 \cdot 10^{-5}$ per core-second [26, 38].

Scenario	Metric	iSimplePIR(Row-level)	**iSimplePIR(Entry-level)**
Row-major	Offline comp. (s)	1.77	1.70
	Offline comm. (MB)	1.26	1.26
	Online comm. (KB)	241	240
	Server comp. (ms)	95.73	94.62
	Throughput (GB/s)	10.4	10.6
	Server cost	\$0.000133	\$0.000134
Col-major	Offline comp. (s)	172.45	0.77
	Offline comm. (MB)	120.75	28.50
	Online comm. (KB)	240	240
	Server comp. (ms)	97.23	95.63
	Throughput (GB/s)	10.3	10.5
	Server cost	\$0.010635	\$0.002527

model [33], we use a representative update scenario where 1% of a 1GB database is modified. The results are presented in Table 3. The monetary costs of outbound communication and server computation are reported based on prevailing AWS pricing, noting that only outbound communication is charged [26, 38].

Online throughput and communication. The hint generated after incremental preprocessing is compatible with the updated database. Therefore, the online phase operations of iSimplePIR (Row-level) and iSimplePIR (Entry-level) remain consistent with the original SimplePIR scheme, ensuring efficient online computation. Specifically, for a 1GB database, the online server computation time of iSimplePIR(Entry-level) is approximately 95ms, which is comparable to that of SimplePIR [26, Table 8].

Offline performance. From Table 3, for modifying 1% of the database content, the preprocessing computation and communication overhead of iSimplePIR (Entry-level) is comparable to that of iSimplePIR (Row-level) under the row-major setting. This demonstrates that even under conditions unfavorable to entry-level incremental preprocessing, iSimplePIR (Entry-level) incurs no additional overhead compared to the row-level scheme, owing to communication-efficient row aggregation. Such efficiency further highlights our method's adaptability to diverse operational scenarios. Under the column-major setting, the advantages of iSimplePIR (Entry-level) become more pronounced. Specifically, compared to iSimplePIR (Row-level), it achieves 224× less computation, 4.2× smaller communication, and is 4.2× cheaper in terms of monetary cost. The sub-

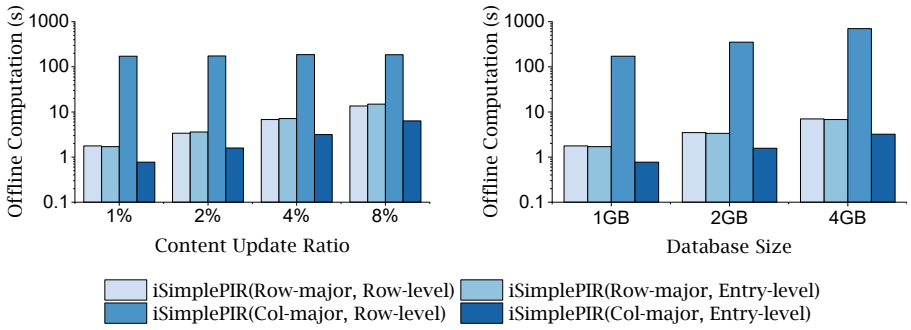


Fig. 3: Offline computation of iSimplePIR (Entry-level) and iSimplePIR (Row-level) under different update ratios and database sizes in two scenarios. Each database entry is one bit. The left figure is for a 1GB database, and the right figure is for a 1% update ratio.

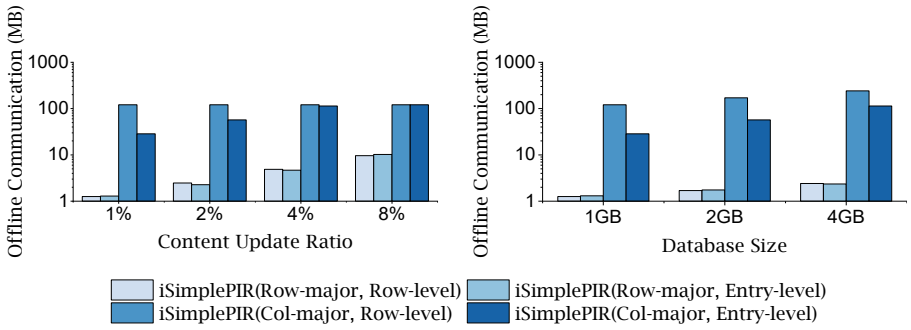


Fig. 4: Offline communication of iSimplePIR (Entry-level) and iSimplePIR (Row-level) under different update ratios and database sizes in two scenarios. Each database entry is one bit. The left figure is for a 1GB database, and the right figure is for a 1% update ratio.

stantial computational savings—beyond the expected theoretical ratio—primarily result from two architectural factors: the simplified loop structure of the entry-level incremental design significantly improves cache locality, thereby reducing memory bandwidth pressure; and the computation exhibits higher instruction-level parallelism and more effective SIMD vectorization.

Figures 3 and 4 illustrate the trends in preprocessing computation and communication costs of iSimplePIR (Row-level) and iSimplePIR (Entry-level) under varying update ratios and database sizes for two different scenarios. Overall, in the row-major setting, both schemes exhibit comparable computation and communication overhead under identical conditions, with costs increasing as the update ratio and database size grow.

In the column-major setting, iSimplePIR (Entry-level) consistently outperforms iSimplePIR (Row-level) in both computation and communication. As

shown in the right subfigures of Figures 3 and 4, the communication advantage of iSimplePIR (Entry-level) diminishes with increasing database size-from a 4.2× reduction at 1GB to a 2.1× reduction at 4GB-due to the number of modified entries growing faster than the number of database rows. Notably, in the left subfigures of Figures 3 and 4, when 8% of the data is modified, the preprocessing communication of iSimplePIR (Entry-level) approach those of iSimplePIR (Row-level). This is because, at higher update ratios, the number of modified entries per row exceeds t, triggering the row aggregation that enables communication optimization. Specifically, for an 8% content update, the monetary cost of iSimplePIR (Entry-level) without communication-optimized row aggregation is \$0.02003. With optimization, the cost is reduced to \$0.01063, yielding a 1.9× reduction. In summary, iSimplePIR (Entry-level) achieves more significant savings for smaller databases and lower update ratios, where the database size remains within a practical range without compromising the applicability of the scheme.

6.2 Password Checkup Benchmark

We conduct experiments on uniformly random updates in the context of password checkup. We use a subset of the database from the Blyss implementation [7, 35] as an example, consisting of approximately 4GB of SHA256 hashes of leaked passwords. Clients can hash their query password using a public hash function and map it to a candidate index in the database. We model updates to the password database as differences between versions, where approximately 1.3 million 32-byte hashes are updated. In the aforementioned environment, we evaluate the offline performance of iSimplePIR (Entry-level) compared to iSimplePIR (Row-level), as well as the online performance of iSimplePIR (Entry-level) compared to various baselines. The experimental results are presented in Figure 5. Note that the underlying SimplePIR scheme requires clients to store a 241MB hint locally. This preprocessing cost can be amortized across multiple subsequent queries. This is particularly suitable for desktop computer scenarios where clients seeking faster online queries may have sufficient local storage for the hint. In contrast, other schemes similarly applied to password checkup also require storing client-side hints, with sizes exceeding 241MB [14].

Online throughput and communication. From the right subfigure of Figure 5, the throughput of iSimplePIR (Entry-level) is 8-29× higher than the Spiral family and 133× higher than MulPIR, due to the inheritability of its online operations. Its communication cost is 1.4× lower than DoublePIR but 7× higher than Spiral. Compared to Checklist [28], our iSimplePIR (Entry-level) achieves better communication efficiency. Checklist stores newly inserted entries in different buckets, requiring queries to all buckets during the online phase.

Offline performance. In the left subfigure of Figure 5, after database updates, iSimplePIR (Entry-level) achieves 86× lower computation and 2.1× smaller communication for offline hint preprocessing compared to iSimplePIR (Row-level). iSimplePIR (Entry-level) leverages its per-entry real-time update capability when the number of updated entries within a row is less than or equal to t; when this number exceeds t, its communication cost becomes comparable to

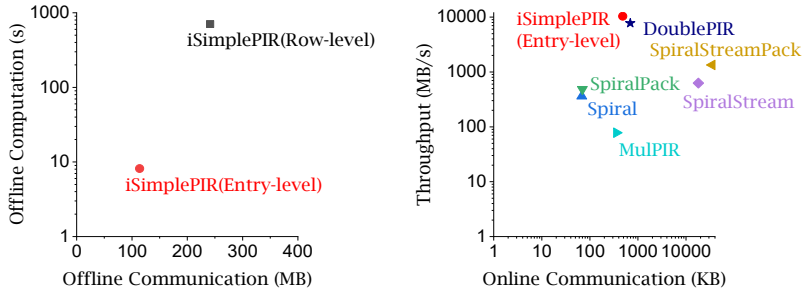


Fig. 5: Offline and online performance of different PIR protocols for password breach detection. The left figure shows the offline performance after database updates, where points closer to the lower left indicate better performance. The right figure shows the online performance, where points closer to the upper left are better. The database size in the figure is 4GB. For detailed experimental settings, see the first paragraph of Section 6.2.

that of iSimplePIR (Row-level). Overall, iSimplePIR (Entry-level) remains more efficient than iSimplePIR (Row-level). Compared to the improvements observed under the column-major configuration in Section 6.1, the computational benefits under fully random updates are reduced. This reduction is due to: diminished cache and vectorization advantages as the database grows, and increased server-side computation caused by row aggregation under random update patterns.

7 Conclusion

This paper gives the first formal definition of single-server incremental PIR, proposes an entry-level incremental construction of SimplePIR, and demonstrates the application of our scheme to password breach detection. Future research directions include constructing entry-level incremental preprocessing for sublinear server-side computation schemes [17, 18, 48].

Acknowledgments. This study was funded by the National Natural Science Foundation of China (grant numbers 12271306, 62472255, 6230227), the Youth Innovation Team of Shandong Province (grant number 2024KJH79), and the Natural Science Foundation of Shandong Province, China (grant numbers ZR2026QB26, ZR2023MF045, ZR2023QF088), and CCF-Tencent Rhino-Bird Open Research Fund (CCF-Tencent RAGR20250129). Guangwu Xu and Zengpeng Li are co-corresponding authors.

A Supplementary Material

A.1 Learning with Errors Assumption

The security of SimplePIR scheme is based on the decisional version of the Learning with Errors (LWE) problem [42]. Let $\mathbf{A} \xleftarrow{R} \mathbb{Z}_q^{m \times n}$ be a uniformly random matrix, let $\mathbf{e} \leftarrow \chi^m$ be a short error vector sampled from a discrete Gaussian distribution χ, and let $\mathbf{s} \xleftarrow{R} \mathbb{Z}_q^n$ be a uniformly random secret vector. The LWE problem states that the distributions $(\mathbf{A}, \mathbf{As}+\mathbf{e})$ and $(\mathbf{A}, \mathbf{r})$ are computationally indistinguishable, where $\mathbf{r}$ is uniformly sampled from $\mathbb{Z}_q^m$.

A.2 Simple Approaches and Potential Attacks

When faced with database updates, there are some straightforward approaches. For instance, when new entries are added to the database, the simplest method is for the client to download and store all the newly inserted entries. However, this approach presents two significant issues: first, downloading and storing large entries directly can substantially increase communication and storage costs; second, it may inadvertently reveal information about the client. We abstract one potential attack as follows:

- *Query frequency attack.* If the entry the client wishes to retrieve is among the newly inserted entries, the client will significantly reduce the frequency of subsequent queries after downloading it directly. A malicious server could infer whether the client is interested in newly added entries by observing changes in query frequency.

Another approach, similar to [28], places all newly inserted entries in a new PIR database. Specifically, during the offline phase, the server preprocesses the new database and provides the client with new hints. In the online phase, the client must send queries to all databases and receive multiple responses; otherwise, it would reveal which database contains the desired entry. Clearly, the multi-database setup increases communication overhead.

Our single-server incremental solution avoids directly storing complete entries. After incremental updates, the client can continue to perform indistinguishable queries, effectively resisting the query frequency attack. Additionally, in our approach, the client does not need to send multiple queries, and the server is not required to compute multiple responses.

References

1. Ali, A., Lepoint, T., Patel, S., Raykova, M., Schoppmann, P., Seth, K., Yeo, K.: Communication-Computation trade-offs in PIR. In: USENIX Security. pp. 1811–1828 (2021)
2. Angel, S., Chen, H., Laine, K., Setty, S.: Pir with compressed queries and amortized query processing. In: S&P. pp. 962–979. IEEE (2018)

3. Angel, S., Setty, S.: Unobservable communication over fully untrusted infrastructure. In: OSDI. pp. 551–569 (2016)
4. Backes, M., Kate, A., Maffei, M., Pecina, K.: Obliviad: Provably secure and practical online behavioral advertising. In: S&P. pp. 257–271. IEEE (2012)
5. Baum, C., Bootle, J., Cerulli, A., Del Pino, R., Groth, J., Lyubashevsky, V.: Sub-linear lattice-based zero-knowledge arguments for arithmetic circuits. In: CRYPTO. pp. 669–699. Springer (2018)
6. Beimel, A., Ishai, Y., Malkin, T.: Reducing the servers computation in private information retrieval: Pir with preprocessing. In: CRYPTO. pp. 55–73. Springer (2000)
7. Blyss: Blyss private password checker. `https://playground.blyss.dev/passwords`, last accessed 2025/4/25
8. Boyle, E., Ishai, Y., Pass, R., Wootters, M.: Can we access a database both locally and privately? In: TCC. pp. 662–693. Springer (2017)
9. Brakerski, Z., Kalai, Y.: Witness indistinguishability for any single-round argument with applications to access control. In: PKC. pp. 97–123. Springer (2020)
10. Cachin, C., Micali, S., Stadler, M.: Computationally private information retrieval with polylogarithmic communication. In: EUROCRYPT. pp. 402–414. Springer (1999)
11. Canetti, R., Holmgren, J., Richelson, S.: Towards doubly efficient private information retrieval. In: TCC. pp. 694–726. Springer (2017)
12. Cao, Q., Tran, H.Y., Dau, S.H., Yi, X., Viterbo, E., Feng, C., Huang, Y.C., Zhu, J., Kruglik, S., Kiah, H.M.: Committed private information retrieval. In: ESORICS. pp. 393–413. Springer (2024)
13. de Castro, L.: VeriSimplePIR. `https://github.com/leodec/VeriSimplePIR`, last accessed 2025/5/25
14. de Castro, L., Lee, K.: VeriSimplePIR: Verifiability in SimplePIR at no online cost for honest servers. In: USENIX Security. pp. 5931–5948 (2024)
15. Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private information retrieval. In: FOCS. pp. 41–41. IEEE Computer Society (1995)
16. Colombo, S., Nikitin, K., Corrigan-Gibbs, H., Wu, D.J., Ford, B.: Authenticated private information retrieval. In: USENIX Security. pp. 3835–3851 (2023)
17. Corrigan-Gibbs, H., Henzinger, A., Kogan, D.: Single-server private information retrieval with sublinear amortized time. In: EUROCRYPT. pp. 3–33. Springer (2022)
18. Corrigan-Gibbs, H., Kogan, D.: Private information retrieval with sublinear online time. In: EUROCRYPT. pp. 44–75. Springer (2020)
19. Davidson, A., Pestana, G., Celi, S.: Frodopir: Simple, scalable, single-server private information retrieval. Proc. Priv. Enhancing Technol. **2023**(1), 365–383 (2023)
20. Dong, C., Chen, L.: A fast single server private information retrieval protocol with low communication cost. In: ESORICS. pp. 380–399. Springer (2014)
21. Fung, E., Kellaris, G., Papadias, D.: Combining differential privacy and pir for efficient strong location privacy. In: SSTD. pp. 295–312. Springer (2015)
22. Gentry, C., Ramzan, Z.: Single-database private information retrieval with constant communication rate. In: ICALP. pp. 803–815. Springer (2005)
23. Green, M., Ladd, W., Miers, I.: A protocol for privately reporting ad impressions at scale. In: CCS. pp. 1591–1601 (2016)
24. Gupta, T., Crooks, N., Mulhern, W., Setty, S., Alvisi, L., Walfish, M.: Scalable and private media consumption with popcorn. In: NSDI. pp. 91–107 (2016)
25. Hayata, J., Schuldt, J.C., Hanaoka, G., Matsuura, K.: On private information retrieval supporting range queries. In: ESORICS. pp. 674–694. Springer (2020)

26. Henzinger, A., Hong, M.M., Corrigan-Gibbs, H., Meiklejohn, S., Vaikuntanathan, V.: One server for the price of two: Simple and fast single-server private information retrieval. In: Usenix Security. pp. 3889–3905 (2023)
27. Kales, D., Rechberger, C., Schneider, T., Senker, M., Weinert, C.: Mobile private contact discovery at scale. In: USENIX Security. pp. 1447–1464 (2019)
28. Kogan, D., Corrigan-Gibbs, H.: Private blocklist lookups with checklist. In: USENIX Security. pp. 875–892 (2021)
29. Kushilevitz, E., Ostrovsky, R.: Replication is not needed: Single database, computationally-private information retrieval. In: FOCS. pp. 364–373. IEEE (1997)
30. Li, L., Pal, B., Ali, J., Sullivan, N., Chatterjee, R., Ristenpart, T.: Protocols for checking compromised credentials. In: CCS. pp. 1387–1403 (2019)
31. Lin, C., Liu, Z., Malkin, T.: Xspir: Efficient symmetrically private information retrieval from ring-lwe. In: ESORICS. pp. 217–236. Springer (2022)
32. Liu, J., Li, J., Wu, D., Ren, K.: PIRANA: Faster multi-query pir via constant-weight codes. In: S&P. pp. 4315–4330. IEEE (2024)
33. Ma, Y., Zhong, K., Rabin, T., Angel, S.: Incremental offline/online pir. In: USENIX Security. pp. 1741–1758 (2022)
34. Melchor, C.A., Barrier, J., Fousse, L., Killijian, M.O.: XPIR: Private information retrieval for everyone. Proc. Priv. Enhancing Technol. **2016**(2), 155–174 (2016)
35. Menon, S., Movva, N., Gurkan, K.: Blyss SDK. `https://github.com/blysspriv acy/sdk`, last accessed 2025/5/15
36. Menon, S.J.: Spiral-rs. `https://github.com/menonsamir/spiral-rs`, last accessed 2025/5/15
37. Menon, S.J., Wu, D.J.: Spiral: Fast, high-rate single-server pir via fhe composition. In: S&P. pp. 930–947. IEEE (2022)
38. Menon, S.J., Wu, D.J.: YPIR: High-throughput single-server pir with silent preprocessing. In: USENIX Security. pp. 5985–6002 (2024)
39. Mittal, P., Olumofin, F., Troncoso, C., Borisov, N., Goldberg, I.: PIR-Tor: Scalable anonymous communication using private information retrieval. In: USENIX Security (2011)
40. Mughees, M.H., Chen, H., Ren, L.: OnionPIR: Response efficient single-server pir. In: CCS. pp. 2292–2306 (2021)
41. Park, J., Tibouchi, M.: Shecs-pir: somewhat homomorphic encryption-based compact and scalable private information retrieval. In: ESORICS. pp. 86–106. Springer (2020)
42. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. J.ACM **56**(6), 1–40 (2009)
43. Thomas, K., Pullman, J., Yeo, K., Raghunathan, A., Kelley, P.G., Invernizzi, L., Benko, B., Pietraszek, T., Patel, S., Boneh, D., Bursztein, E.: Protecting accounts from credential stuffing with password breach alerting. In: USENIX Security. pp. 1556–1571 (2019)
44. Wang, K.C., Reiter, M.K.: Detecting stuffing of a user's credentials at her own accounts. In: USENIX Security. pp. 2201–2218 (2020)
45. Wang, S., Ding, X., Deng, R.H., Bao, F.: Private information retrieval using trusted hardware. In: ESORICS. pp. 49–64. Springer (2006)
46. Williams, P., Sion, R.: Single round access privacy on outsourced storage. In: CCS. pp. 293–304 (2012)
47. Wu, D.J., Zimmerman, J., Planul, J., Mitchell, J.C.: Privacy-preserving shortest path computation. In: NDSS (2016)
48. Zhou, M., Park, A., Zheng, W., Shi, E.: Piano: extremely simple, single-server pir with sublinear server computation. In: S&P. pp. 4296–4314. IEEE (2024)