Distributional Private Information Retrieval

by

Ryan Lehmkuhl

B.S., University of California Berkeley (2021)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2024

Authored by:	Ryan Lehmkuhl Department of Electrical Engineering and Computer Science August 20, 2024
Certified by:	Henry Corrigan-Gibbs Assistant Professor of Computer Science Thesis Supervisor
Accepted by:	Leslie A. Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Distributional Private Information Retrieval

by

Ryan Lehmkuhl

Submitted to the Department of Electrical Engineering and Computer Science
on August 20, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

ABSTRACT

A private-information-retrieval (PIR) scheme lets a client fetch a record from a remote database without revealing which record it has fetched. Classic PIR schemes treat all database records the same but, in practice, some database records are much more popular (i.e., commonly fetched) than others. We introduce *distributional private information retrieval*, a new type of PIR that can run faster than classic PIR—both asymptotically and concretely—when the popularity distribution is heavily skewed. Distributional PIR provides exactly the same cryptographic privacy notion as classic PIR. The speedup comes from providing a relaxed form of correctness: distributional PIR guarantees reliable retrieval for PIR queries that follow the popularity distribution, but only "best-effort" retrieval for out-of-distribution queries.

We give several constructions of distributional-PIR schemes that make black-box use of existing standard PIR protocols. On a popularity distribution drawn from real-world Twitter data, distributional PIR reduces compute costs by 5.1–77× compared to existing techniques. Finally, we build CrowdSurf, an end-to-end system for privately streaming social-media posts, and show that our PIR schemes reduce the end-to-end server cost by 8×.

Thesis supervisor: Henry Corrigan-Gibbs
Title: Assistant Professor of Computer Science

Contents

The material in this thesis is based on joint work with Alexandra Henzinger and Henry Corrigan-Gibbs.

Chapter 1

Introduction

Today, the tweets we read on Twitter, the queries we make to Google, and the videos we watch on YouTube all reveal sensitive information about us—letting each service learn our interests and our activities. Cryptographic protocols for private information retrieval (PIR) [1], [2] protect this query data: PIR lets a user fetch a record from a remote database without revealing which record it is fetching to the server hosting the database. The performance of PIR has improved dramatically over the last decade [3]–[13], showing the potential viability of private web search [12], private media delivery [14], [15], and metadata-hiding messaging [16], [17].

Unfortunately, a longstanding barrier to the deployment of PIR has been its computational overhead. An information-theoretic lower bound [18] shows that, if the server learns nothing about which database record a user is fetching, then it must compute over *every* record in the database to answer the user's query. As a result, to answer a PIR query to an N-bit database, the server cannot do better than to run in $\Omega(N)$ time. In contrast, a non-private lookup on a RAM machine requires just $O(1)$ operations. Additionally, non-private database systems further optimize lookups by utilizing caching and prefetching popular elements, leveraging information about which records a user queried previously and which records are being queried by other users [19], [20].

In this work, we aim to translate these performance optimizations into the world of privacy-preserving databases. We introduce a new approach to bypassing PIR's compute lower bound in both theory and practice: taking advantage of the fact that some records in a database are much more *popular* (i.e., likely for clients to query) than others. For example, the top 1% of Twitter users have the vast majority of followers on the platform [21], the top Google search queries are made much more frequently than others [22], and the most popular YouTube videos have orders of magnitude more views than the average video [23]. We show that, even without learning anything about the record that a user is fetching, a server can take advantage of this skewed access pattern to answer user queries much faster than a standard PIR server can.

We formalize this notion as *distributional PIR*. A distributional-PIR scheme is defined relative to a popularity distribution $\mathcal{P}$ over the set of database records, which encodes how likely a user is to fetch each record. Distributional PIR provides exactly the same security guarantee as standard PIR: that is, after answering a PIR query, the server knows no more information about which record the user was fetching than before the interaction—whether or not the user's query pattern follows the popularity distribution $\mathcal{P}$. The performance gains of distributional PIR come from relaxing the scheme's correctness guarantee, ensuring that

1. if the user's queries follow access distribution $\mathcal{P}$, then they will recover their desired record with good probability, but
2. if the user deviates from $\mathcal{P}$, then they will recover their record with lower probability.

This relaxed notion of correctness, which provides reliable retrieval for in-distribution queries and "best effort" retrieval for out-of-distribution queries, is exactly the source of distributional PIR's power over standard PIR.

Distributional PIR constructions. We present new constructions of distributional PIR schemes that make black-box use of any standard PIR scheme. Our constructions work by replicating the "popular" database entries into a separate, small database. When generating a query, with some probability clients query the popular database instead of the original one. The more skewed the access distribution $\mathcal{P}$, the smaller the popular database is, allowing the server to run in much less time. We provide lower bounds that show that, for any access distribution $\mathcal{P}$, the runtime of our new distributional PIR protocols asymptotically matches the optimal server runtime achievable (without preprocessing the database) up to log factors.

To give an example: if on a database with N-entries, 90% of the queries are expected to hit only 10% of the database records, we replicate the $0.1N$ most-popular entries into a "popular" database. Our distributional PIR scheme then routes 90% of queries to the popular database and 10% of queries to the original database. All in all, this scheme can run in time $0.9 \cdot 0.1N + 0.1 \cdot N = 0.19 \cdot N$–over 5× faster than standard PIR. On a real-world Twitter popularity distribution, our microbenchmarks show that distributional PIR successfully returns 19 of 24 queried tweets on average, while reducing server work by 5–77× and communication by 2–117× compared to existing batch-PIR schemes.

Along the way, we present new optimizations to state-of-the-art classical PIR protocols, improving their performance through a new application of existing cryptographic techniques and the use of different hardware architectures. In particular, we improve the running time of client encryption in SimplePIR [8] by 116–351× by working with a more structured cryptographic assumption ("ring learning-with-errors") and efficiently translating these ciphertexts to those expected by SimplePIR via a new application of known techniques ("modulus switching" [24]). In addition, we utilize fast matrix-multiplication kernels on modern GPUs to speed up the per-query server work in SimplePIR [8] by upwards of 3× compared to a cluster of CPUs (maintaining the same dollar-cost per query).

Private Twitter feeds with CrowdSurf. To demonstrate that distributional PIR's theoretical performance gains translate into practice, we design and implement CrowdSurf, a system for privately generating Twitter feeds. Analyzing a 2014 crawl of the Twitter social graph spanning 505 million accounts, we observe that the distribution of follower counts is heavily skewed, with a tail following a power-law distribution [25]. Our CrowdSurf server runs a distributional PIR scheme (parametrized by this real-world Twitter distribution) to let users privately retrieve the recent tweets of accounts that they follow. Our end-to-end evaluation shows that a user can privately fetch, on average, 19 tweets from a 38GB database with 500ms of latency and 34MB of traffic—costing the server 0.0057 cents in total, 8× less than existing techniques.

Limitations. While it provides large improvements in server runtime, distributional PIR comes with two main drawbacks. First, the PIR server must have a good approximation of the access distribution $\mathcal{P}$. While some applications naturally reveal such an access distribution to the server (e.g., Twitter publishes follower counts, Amazon publishes how often each product is purchased), collecting these statistics in a privacy-preserving manner may be more challenging in other applications. Second, distributional PIR provides a weaker correctness notion than standard PIR as clients may not have all of their queries successfully answered. Though this is reasonable for applications that can work with "best effort" retrieval (e.g., Twitter feeds), it may not be suitable for other deployment scenarios.

Notation. Throughout this thesis, we model all computation as being performed on a random-access machine. For some input database of size N, we assume the word length of the machine is $\geq \Omega(\log N)$ when analyzing runtime costs.

We denote $[m]$ as the set of integers $\{0, 1, \cdots, m-1\}$. We define a probability distribution $\mathcal{P}$ over $[N]$ as a list of N floating point numbers $(p_1, p_2, \ldots p_N)$. We will generally assume that the distribution $\mathcal{P}$ is sorted by popularity, i.e., $p_1 \geq p_2 \geq \cdots \geq p_N$. $F_{\mathcal{P}}(\cdot)$ denotes the CDF of $\mathcal{P}$ and $F_{\mathcal{P}}^{-1}(\cdot)$ the inverse CDF. We use $\widetilde{O}(\cdot)$ to hide poly-logarithmic factors in the input. For some list L, we denote the sub-list containing the i-th to j-th elements (inclusive) as $L_{i..j}$ or $L[i..j]$.

Chapter 2

Related Work

Batch PIR. Ishai et al. [26] introduced batch codes to construct PIR schemes that let a client make B queries at close to the cost of one. Followup works by Angel et al. [4], [16] explored probabilistic batch codes, i.e., batch codes that fail with some probability, and showed that they are significantly more efficient in certain parameter regimes. While batch PIR can be achieved via black-box use of any PIR scheme and batch code, recent works [27]–[29] have additionally explored schemes that combine the two in a non-black-box manner in order to improve performance. All of these works are complementary to our distributional-PIR constructions since we make black-box use of any batch-PIR scheme.

There is a large body of work on PIR codes [30] which, in multi-server setting, allow each of the PIR servers to store only a fraction of the entire database. This task is orthogonal from our own.

PIR with popularity distributions. Recent work by Lam et al. [31], explored how to improve the cost of batch PIR in the two-server setting. They observe that the database contains "hot indices" that clients access more often and propose splitting the database up into two distinct buckets that clients query with different frequency. At the time of writing, the authors don't provide an implementation we could compare against or details on how they parameterize their scheme. However, because their scheme satisfies a much stronger notion of correctness than what we target, we expect it to perform worse than our constructions.

Several works in the information-theory community have studied PIR in a similar setting to ours, in which the client and server know the relative popularity of the database elements [32], [33]. These works are orthogonal to our own in that they focus only on *communication*: minimizing the number of bits the servers must send to the client in an information-theoretic sense. In contrast, our focus is on server *computation*, which these prior works do not address at all.

Sublinear-time PIR protocols. Beimel, Ishai, and Malkin [18] prove that a PIR server must run in at least N time to answer queries to an N-record database. Recent work has suggested other models of PIR in which their lower bound does not apply: letting the server preprocess the database [18], [34] or interact with the user ahead of time [35]–[39]. These works are complementary to our own as our main construction makes black-box use of an arbitrary PIR scheme as its cryptographic core.

Lattice-based homomorphic encryption schemes. There is a long line of working exploring how to improve the efficiency of lattice-based homomorphic encryption schemes [7], [8], [11], [12], [40]–[46]. Most relevant to our optimizations, Li et al. [11] and Henzinger et al. [12] both construct "hybrid" schemes that combine LWE and RingLWE-based encryption schemes–their techniques are complementary to our own. Similar to us, Menon et al. [9] designs a hybrid scheme that uses RingLWE-based assumptions to speed up preprocessing. Our approach outperforms their preprocessing techniques (Section 6.0.2) and additionally enables fast encryption, but slightly decreases the number of homomorphic operations that can be performed (see Chapter 6 for more details).

Chapter 3

Defining Distributional PIR

In this chapter, we introduce distributional PIR, a new type of private-information-retrieval scheme targeted for applications in which (1) some elements of the database are more often queried than others and (2) a relaxed correctness guarantee is acceptable. In this case, distributional PIR schemes require sublinear computation in the database size.

A distributional PIR scheme on a database of N items takes as input a probability distribution $\mathcal{P}$ over the indices $\{1, \dots, N\}$. With each query, the client can request a batch of B items from the server, as in a batch-PIR scheme [26], [27], [47]. We sketch the properties that a distributional-PIR scheme provides here, and then we state these properties formally.

- *Privacy.* No matter what database entries the client fetches, the server learns no information about the client's query. The privacy guarantee here is exactly the same as in standard PIR.

The correctness properties for a distributional PIR scheme are slightly different from prior ones:

- *Explicit correctness.* Clients are always aware if their query wasn't answered successfully. In other words, no matter what set of database records a client requests, for each record they receive either the corresponding database entry or a failure symbol–never an incorrect database record.
- *Average-case correctness.* If the client i.i.d. samples a set of records to fetch from the popularity distribution $\mathcal{P}$, the client on average recovers a certain fixed fraction of the records.
- *Worst-case correctness.* No matter what set of database records the client requests, it recovers each record with some fixed probability.

Finally, the main advantage of using a distributional PIR is:

- *Low expected server time.* If the client's query distribution $\mathcal{P}$ is far from uniform, the server's Answer routine runs in time less than the database size.

3.1 Background: Batch private information retrieval

Before defining distributional PIR, we provide some background on a related primitive, batch private information retrieval (batch PIR) [48]. A batch-PIR scheme allows a client to privately fetch a list of elements from a server's database. In more detail, a batch-PIR scheme defined over some plaintext space $\mathcal{M}$ and batch size B is defined by the following routines:

– $\mathsf{Setup}(1^N) \to \mathsf{pp}$. Given a database size N expressed in unary, output public parameters pp.
– $\mathsf{Encode}(\mathsf{pp}, D) \to D_{\mathsf{code}}$. Given public parameters pp and a database $D \in \mathcal{M}^N$ as input, output an encoded database D_{code}.
– $\mathsf{Query}(\mathsf{pp}, I) \to (\mathsf{st}, q)$: Given public parameters pp and a list of query indices $I \in [N]^B$, output client state st and a query q.
– $\mathsf{Answer}^{D_{\mathsf{code}}}(q) \to a$: Given oracle access to the records of an encoded database D_{code} and client query q, output an answer a.
– $\mathsf{Recover}(\mathsf{st}, a) \to (\mathcal{M} \cup \{\perp\})^B$: Given client state st and answer a, output a list of B items, each of which can either be a database record or a failure symbol $\perp$.

Batch-PIR schemes use the standard notion of PIR correctness and security.

Correctness. A client should be able to recover their database indices of interest with overwhelming probability. Concretely, we say that a batch-PIR scheme over message space $\mathcal{M}$ and batch size B has *correctness error* ϵ if the following holds for all database sizes $N \in \mathbb{N}$, databases $D \in \mathcal{M}^N$ and list of indices $I \in [N]^B$:

$$\Pr\left[\forall j \in [B],\ m_j = D_{I_j} \;\middle|\; \begin{array}{rl} \mathsf{pp} & \leftarrow \mathsf{Setup}(1^N) \\ D_{\mathsf{code}} & \leftarrow \mathsf{Encode}(\mathsf{pp}, D) \\ (\mathsf{st}, q) & \leftarrow \mathsf{Query}(\mathsf{pp}, I) \\ a & \leftarrow \mathsf{Answer}^{D_{\mathsf{code}}}(q) \\ (m_1, \dots, m_B) & \leftarrow \mathsf{Recover}(\mathsf{st}, a) \end{array} \right] \geq 1 - \epsilon.$$

Security. The client's query should leak no information about their requested database indices. Since the security definition of batch PIR is the same as distributional PIR (up to syntax), we defer the full security description to Appendix A.

Server runtime. For a given database size N, we say that a batch-PIR scheme has *server runtime* T if for all databases $D \in \mathcal{M}^N$ and list of indices $I \in [N]^B$, the Answer routine runs in time at most T. We typically measure the running time in terms of the number of probes that Answer makes to D_{code}.

Communication cost. For a given database size N, we say that a batch-PIR scheme has *communcation cost* C if for all databases $D \in \mathcal{M}^N$, list of indices $I \in [N]^B$, and randomness of the PIR algorithms:

$$\max_{D,I} \left(|\mathsf{pp}| + |q| + |a| \;:\; \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^N) \\ D_{\mathsf{code}} \leftarrow \mathsf{Encode}(\mathsf{pp}, D) \\ (\_, q) \leftarrow \mathsf{Query}(\mathsf{pp}, I) \\ a \leftarrow \mathsf{Answer}^{D_{\mathsf{code}}}(q) \end{array} \right) \leq C.$$

3.2 Definition

We now formally define distributional PIR.

Syntax. The syntax for a distributional-PIR scheme is a strict generalization of the standard syntax for batch-PIR schemes, only taking a popularity distribution $\mathcal{P}$ as an additional input. Setting the popularity distribution $\mathcal{P}$ to be arbitrary recovers the syntax of a standard batch-PIR scheme.

In more detail, a distributional-PIR scheme is parameterized by a message space $\mathcal{M}$, a database size $N \in \mathbb{N}$, and a batch size $B \in \mathbb{N}$. All algorithms are randomized and implicitly take as input a security parameter. Such a scheme consists of the following routines:

- $\mathsf{Dist.Setup}(\mathcal{P}) \to \mathsf{pp}$. Given a popularity distribution $\mathcal{P}$, output public parameters pp.
- $\mathsf{Dist.Encode}(\mathsf{pp}, \mathcal{P}, D) \to D_{\mathsf{code}}$. Given public parameters pp, a popularity distribution $\mathcal{P}$, and database $D \in \mathcal{M}^N$, output an encoded database D_{code}.
- $\mathsf{Dist.Query}(\mathsf{pp}, I) \to (\mathsf{st}, q)$. Given public parameters pp and a list of indices $I \in [N]^B$, output client state st and a query q.
- $\mathsf{Dist.Answer}^{D_{\mathsf{code}}}(q) \to a$. Given oracle access to the records of an encoded database D_{code} and client query q, output an answer a.
- $\mathsf{Dist.Recover}(\mathsf{st}, a) \to (\mathcal{M} \cup \{\perp\})^B$. Given client state st and answer a, output a list of B items, each of which can either be a database record or a failure symbol $\perp$.

Since the popularity distribution $\mathcal{P}$ may be as large as the database D itself, we have both Setup and Encode separately take it as input: this allows the public parameters pp—which clients would download in a real PIR deployment—to be fairly small even if Answer requires much more information about the distribution $\mathcal{P}$.

3.2.1 Security

Our notion of security for distributional PIR is nearly identical to that of traditional batch PIR. In particular, clients' queries reveal nothing about their desired indices. Since security of a distributional-PIR scheme can either be informational-theoretic or computational, we handle both cases by bounding the advantage of an adversary $\mathcal{A}$ by some value δ. In the computational setting, δ is negligible in some security parameter λ, and the runtime of $\mathcal{A}$, $N, \log M$, and B are all polynomial in λ. In the information-theoretic setting, the runtime of the $\mathcal{A}$, $\delta, N, \log M$, and B can be any constants.

Security only holds if the client's query pattern is independent of its query indices or the success/failure of past queries. For example, if a client makes a batch query, an item in that batch fails, and the client immediately re-queries as a result, the server may be able to deduce information about the client's queries. As a result, if failure is unacceptable one must choose parameters so that failure occurs with negligible probability. However, in many applications of interest (e.g. Chapter 7), "best-effort" retrieval is acceptable and the client can simply ignore failures.

We define security using Experiment 3.2.1. Let $\mathsf{Sec}_\Pi(\cdot, \cdot)$ denote the output of the experiment, then for some adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$, define their *advantage* with respect to Π as:

$$\mathsf{DistAdv}[\mathcal{A}, \Pi] = \left|\Pr[\mathsf{Sec}_\Pi(\mathcal{A}, 0) = 1] - \Pr[\mathsf{Sec}_\Pi(\mathcal{A}, 1) = 1]\right|.$$

We say that a distributional-PIR scheme Π is δ-secure iff for all adversaries $\mathcal{A}$:

$$\mathsf{DistAdv}[\mathcal{A}, \Pi] \le \delta.$$

Experiment 3.2.1 (Distributional PIR: Security experiment)**.** The experiment is parameterized by (1) a distributional PIR scheme $\Pi = (\mathsf{Dist.Setup}, \mathsf{Dist.Encode}, \mathsf{Dist.Query}, \mathsf{Dist.Answer}, \mathsf{Dist.Recover})$ with message space $\mathcal{M}$, database size N, and batch size B, (2) an adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$, (3) a bit $b \in \{0, 1\}$. We compute the output of the experiment as:

$\underline{\mathsf{Sec}_\Pi(\mathcal{A}, b) :}$

$$\begin{aligned}
(\mathsf{st}, \mathcal{P}, I_0, I_1) &\leftarrow \mathcal{A}_0() \\
\mathsf{pp} &\leftarrow \mathsf{Dist.Setup}(\mathcal{P}) \\
(\_, q) &\leftarrow \mathsf{Dist.Query}(\mathsf{pp}, I_b) \\
\text{Output } b' &\leftarrow \mathcal{A}_1(\mathsf{st}, \mathsf{pp}, q)
\end{aligned}$$

While our security definition only explicitly reasons about single queries, security holds for a client making multiple queries using the same set of public parameters. To see why, assume that some distributional-PIR scheme Π is δ-secure; imagine an extension of Experiment 3.2.1 for $k > 1$ queries where $\mathcal{A}_0$ outputs $((I_{0,0}, I_{1,1}), \ldots, (I_{k,0}, I_{k,1}))$ and $\mathcal{A}_1$ receives queries for $(I_{0,b}, \ldots, I_{k,b})$ for some $b \in \{0, 1\}$. We can define k hybrid distributions where in the i-th hybrid we replace the i-th query with a query for $I_{i,1-b}$. Since the client is stateless and queries are independent of eachother, the δ-security of Π implies that $\mathcal{A}$ has at most advantage δ in distinguishing any pair of these hybrids. Thus, by the triangle equality $\mathcal{A}$ has at most advantage $k\delta$ against the multi-query experiment.

3.2.2 Correctness

In traditional PIR, the server knows nothing about clients' access patterns, motivating a strict form of correctness that allows the client to fetch any entry with overwhelming probability. Distributional PIR fundamentally diverges from this: the popularity distribution provides additional information about clients' queries, enabling a server to make informed guesses about the locations of clients' queries in order to improve performance. However, in order to take full advantage of this, the server should be allowed to occasionally guess incorrectly.

Defining correctness in the presence of occasional errors takes some care. First, it becomes crucial to distinguish between *silent* and *explicit* failures: an implicit error occurs if the $\mathsf{Dist.Recover}$ returns an incorrect database element, an explicit error occurs when $\mathsf{Dist.Recover}$ returns a $\perp$ symbol. Below we provide several notions of correctness that bound the frequency of these two types of failures.

We define all of our correctness notions using Experiment 3.2.2. The experiment simulates the process of a distributional-PIR client interacting with a server, returning the indices of the records that the simulated client successfully recovered. We let $\mathsf{Correct}_\Pi(\cdot, \cdot, \cdot)$ denote the output of Experiment 3.2.2. Similar to our security definition, we present single-query correctness definitions that imply multi-query ones.

Experiment 3.2.2 (Distributional PIR: Correctness experiment)**.** The experiment is parameterized by (1) a distributional PIR scheme $\Pi = (\mathsf{Dist.Setup}, \mathsf{Dist.Encode}, \mathsf{Dist.Query}, \mathsf{Dist.Answer}, \mathsf{Dist.Recover})$ with message space $\mathcal{M}$, database size N, and batch size B, (2) a popularity distribution $\mathcal{P}$ over $[N]$, (3) a database $D = (D_1, \ldots, D_N) \in \mathcal{M}^N$, and (4) a list of indices $I \in [N]^B$. We compute the output of the experiment as:

$\underline{\mathsf{Correct}_\Pi(\mathcal{P}, D, I):}$

$$\begin{aligned} \mathsf{pp} &\leftarrow \mathsf{Dist.Setup}(\mathcal{P}) \\ (\mathsf{st}, q) &\leftarrow \mathsf{Dist.Query}(\mathsf{pp}, I) \\ a &\leftarrow \mathsf{Dist.Answer}^{D_{\mathsf{code}}}(q) \\ (m_1, \ldots, m_B) &\leftarrow \mathsf{Dist.Recover}(\mathsf{st}, a) \end{aligned}$$

– If $\exists j \in [B]$ such that $m_j \neq \bot$ and $m_j \neq D_{I_j}$, return the failure symbol $\bot$.
– Else, return $\{j \in [B] \mid m_j = D_{I_j}\}$.

Explicit correctness For most applications of interest, frequent implicit failures are unacceptable. As a result we first introduce a notion of correctness–similar to standard PIR correctness–that bounds the probability of these occurring. We say that a distributional-PIR scheme has explicit correctness if, no matter which database records the client wants to fetch, each index they recover either is the desired record or a failure symbol with probability negligible in the implicit correctness parameter. In other words, while a distributional-PIR scheme is able to fail regularly, it should almost never return an incorrect database record.

Formally, let Π be a distributional-PIR scheme over message space $\mathcal{M}$, database size N, and batch size B. We say that Π has explicit correctness error ϵ if for all popularity distributions $\mathcal{P}$, databases $D \in \mathcal{M}^N$, and lists of indices $I \in [N]^B$:

$$\Pr\left[\mathsf{Correct}_\Pi(\mathcal{P}, D, I) = \bot\right] \leq \epsilon.$$

We now bound the frequency of non-silent failures through two new notions of correctness.

Average-case correctness. First, we consider clients whose query pattern follows the popularity distribution. In particular, we model clients as sampling their indices i.i.d. from the popularity distribution $\mathcal{P}$. While this is a simplification of real-word behavior, in Section 8.1 we demonstrate that it accurately approximates clients' behavior on real-world data. If, for a particular application, the true query distribution is far from this model, we later provide tight bounds on how correctness is affected (Proposition 3.3.3) and show how to boost the correctness of any distributional-PIR scheme accordingly (Lemma 4.2.2).

To allow for occasional failures we bound *average-case* behavior, requiring that, on average, each client should recover a fixed fraction of their queries. In particular, we say that a distributional-PIR scheme has average-case correctness μ_{avg} on a probability distribution $\mathcal{P}$ if, when the client queries for a list of B indices sampled i.i.d. from the distribution $\mathcal{P}$, the client recovers a μ_{avg} fraction of its desired records, in expectation over the random draws from $\mathcal{P}$ and the randomness of the PIR algorithms.

Formally, let Π be a distributional PIR scheme over message space $\mathcal{M}$, database size N, and batch size B. We say that the distributional PIR scheme Π has *average-case correctness* μ_{avg} on a probability distribution $\mathcal{P}$ over $[N]$ if, for all databases $D \in \mathcal{M}^N$:

$$\mathbb{E}\left[\frac{|\mathsf{Correct}_{\Pi}(\mathcal{P}, D, I)|}{B} \;:\; I \xleftarrow{\mathrm{R}} \mathcal{P}^B\right] \geq \mu_{\mathsf{avg}}$$

Worst-case correctness. While relaxing correctness to only bound average-case behavior helps enable the performance gains of distributional PIR, it may have undesirable effects. For example, given a skewed popularity distribution, the server could completely ignore low-probability elements in the tail of the distribution, which would be unacceptable for some applications. To combat this, we introduce an additional notion of worst-case correctness that ensures the server successfully answers *every* possible query with some probability. One subtlety here is that we enforce this success probability over each *individual* element in a given batch rather than the entire batch itself (as in average-case correctness); this ensures that for any query, each index is recovered with some fixed probability.

A distributional-PIR scheme has worst-case correctness μ_{worst} if, no matter which database records the client wants to fetch, the client recovers each of its desired records with probability at least μ_{worst}, where the probability is over the randomness of the PIR algorithms. When the batch size $B = 1$, this is exactly the standard correctness notion for PIR schemes.

Formally, let Π be a distributional PIR scheme over message space $\mathcal{M}$, database size N, and batch size B. We say that the distributional PIR scheme Π has *worst-case correctness* μ_{worst} on a popularity distribution $\mathcal{P}$ if, for all databases $D \in \mathcal{M}^N$, all lists of indices $I = (I_1, \ldots, I_B) \in [N]^B$, and all $j \in [B]$:

$$\Pr[j \in \mathsf{Correct}_{\Pi}(\mathcal{P}, D, I)] \geq \mu_{\mathsf{worst}}$$

Note that worst-case correctness is a strictly stronger notion than average-case correctness. In particular, if a scheme has worst-case correctness μ_{worst}, then it also has average-case correctness μ_{worst}.

Expected server time. For a probability distribution $\mathcal{P}$ over $[N]$ we say that a distributional-PIR scheme has *expected server time* T on distribution $\mathcal{P}$ if for all databases $D \in \mathcal{M}^N$:

$$\mathbb{E}\left[\mathsf{Time}(\mathsf{Correct}_{\Pi}(\mathcal{P}, D, I)) \;:\; I \xleftarrow{\mathrm{R}} \mathcal{P}^B\right] \leq T,$$

where $\mathsf{Time}(\cdot)$ denotes the running time of the $\mathsf{Dist.Answer}$ algorithm in Experiment 3.2.2. We typically measure the running time in terms of the number of probes that $\mathsf{Dist.Answer}$ makes to the encoded database.

Expected communication cost. For a probability distribution $\mathcal{P}$ over $[N]$ we say that a distributional-PIR scheme has *expected communication cost* C on distribution $\mathcal{P}$ if for all databases $D \in \mathcal{M}^N$:

$$\mathbb{E}\left[\begin{array}{c|r} & I \xleftarrow{\mathrm{R}} \mathcal{P}^B \\ & \mathsf{pp} \leftarrow \mathsf{Dist.Setup}(\mathcal{P}) \\ |\mathsf{pp}| + |q| + |a| \;:\; & D_{\mathsf{code}} \leftarrow \mathsf{Dist.Encode}(\mathsf{pp}, \mathcal{P}, D) \\ & (\_, q) \leftarrow \mathsf{Dist.Query}(\mathsf{pp}, I) \\ & a \leftarrow \mathsf{Dist.Answer}(D_{\mathsf{code}}, q) \end{array}\right] \leq C$$

Fact 3.2.3 (PIR schemes are distributional PIR schemes)**.** *A batch-PIR scheme with correctness error ϵ is a distributional PIR scheme, in the sense of Section 3.2 with explicit correctness error ϵ and both worst- and average-case correctness $1-\epsilon$. Moreover,* Dist.Setup *takes no popularity distribution as input.*

3.3 Robustness against distribution shift

In practice, the estimated query-popularity distribution $\mathcal{P}$ that the PIR service provider uses to generate its public parameters may be slightly different from the *true* popularity distribution $\hat{\mathcal{P}}$ from which clients sample their queries. An important question is how a distributional PIR scheme behaves when distributions $\mathcal{P}$ and $\hat{\mathcal{P}}$ are not identical—i.e., under distribution shift.

Fortunately, we can show that as long as the distributions $\mathcal{P}$ and $\hat{\mathcal{P}}$ are "close" in statistical distance, the average-case performance of a distributional PIR scheme under the true query distribution $\hat{\mathcal{P}}$ is "close" to that under the estimated query distribution $\mathcal{P}$.

To make this precise, we must first define the notion of statistical distance. Recall that for probability distributions $\mathcal{D}$ and $\hat{\mathcal{D}}$ over a finite set S, the *statistical distance* between the distributions $\Delta(\mathcal{D}, \hat{\mathcal{D}})$ is defined as:

$$\Delta(\mathcal{D}, \hat{\mathcal{D}}) \overset{\text{def}}{=} \frac{1}{2} \sum_{x \in S} \left| \Pr_{X \sim \mathcal{D}}[X = x] - \Pr_{\hat{X} \sim \hat{\mathcal{D}}}[\hat{X} = x] \right|,$$

where $X \sim \mathcal{D}$ denotes the random variable X being distributed according to $\mathcal{D}$. Statistical distance gives a limit on how well an algorithm (even a computationally unbounded one) can distinguish two distributions. Formally:

Fact 3.3.1 ([49, Theorem 3.11])**.** *Let S and S' be finite sets and let $\mathcal{D}$ and $\hat{\mathcal{D}}$ be probability distributions over S, and let $f\colon S \to S'$ be a function. If we let $f(\mathcal{D})$ denote the distribution induced by taking a sample from $\mathcal{D}$ and applying f to it, we have:*

$$\Delta(f(\mathcal{D}), f(\hat{\mathcal{D}})) \leq \Delta(\mathcal{D}, \hat{\mathcal{D}}).$$

When sampling batches of elements from a distribution, statistical distance gives the following bound:

Fact 3.3.2 ([50])**.** *Let $\mathcal{D}^B$ denote the B-fold product distribution (i.e., the distribution induced by taking B i.i.d. samples from each distribution). Then,*

$$\Delta(\mathcal{D}^B, \hat{\mathcal{D}}^B) \leq B \cdot \Delta(\mathcal{D}, \hat{\mathcal{D}}).$$

Using these facts, we can bound the effect of distribution shift on a distributional PIR scheme's correctness:

Proposition 3.3.3. *If a distributional PIR scheme with batch size B has average-case correctness μ_{avg} under query distribution $\mathcal{P}$, then it has average-case correctness at least $\hat{\mu_{\mathsf{avg}}} = \mu_{\mathsf{avg}} - B \cdot \Delta(\mathcal{P}, \hat{\mathcal{P}})$ under query distribution $\hat{\mathcal{P}}$.*

Proof of Proposition 3.3.3. For a distributional PIR scheme Π, popularity distribution $\mathcal{P}$ and database $D \in \mathcal{M}^N$, let $f_{\Pi,\mathcal{P},D} \colon [N]^B \to [B]$ denote the function that takes as input a set of query indices I, and outputs the result of the PIR correctness experiment $\mathsf{Correct}_\Pi(\mathcal{P}, D, I)$ (i.e., it outputs the indices of all successful retrievals). Then Facts 3.3.1 and 3.3.2 imply that the output of $f_{\Pi,\mathcal{P},D}$ is almost the same, up to a statistical distance of at most $B \cdot \Delta(\mathcal{P}, \hat{\mathcal{P}})$, when the queries are sampled from true distribution $\hat{\mathcal{P}}$ instead of estimated distribution $\mathcal{P}$. Thus, the expected size of the output of $\mathsf{Correct}_\Pi(\mathcal{P}, D, I)$ differs by at most $\kappa = B \cdot \Delta(\mathcal{P}, \hat{\mathcal{P}})$. By Definition 3.2.2, this means that if the distributional PIR scheme has average-case correctness μ_{avg} under query distribution $\mathcal{P}$, then it has average-case correctness at least $\mu_{\mathsf{avg}} - \kappa$ under query distribution $\hat{\mathcal{P}}$. $\square$

In the following chapter, we provide a generic technique for boosting the correctness of a distributional-PIR scheme (Lemma 4.2.2), which can be used to mitigate the effect of distribution shift.

Chapter 4

Distributional PIR Constructions

In this chapter, we construct a distributional-PIR scheme from any batch-PIR scheme [48]. In particular, we show that:

Theorem 4.0.1. *For all constants $\mu_{\mathsf{avg}}, \mu_{\mathsf{worst}} \in [0, 1]$, database sizes N, batch sizes B, and probability distributions $\mathcal{P}$, if there exists an errorless, δ-secure batch-PIR scheme Π_{batch} such that:*

– Π_{batch} *is parameterized by a message space M, and batch size B. On input a database of size K, Π_{batch} has server runtime $\widetilde{O}(K)$ and communication cost $C(K)$.*

Then there exists a 2δ-secure distributional-PIR scheme Π with:
– *explicit correctness error* 0
– *average-case correctness μ_{avg},*
– *worst-case correctness μ_{worst},*
– *expected server runtime:* $\widetilde{O}\left(F_{\mathcal{P}}^{-1}\left(\frac{\mu_{\mathsf{avg}}-\mu_{\mathsf{worst}}}{1-\mu_{\mathsf{worst}}}\right)\cdot(1-\mu_{\mathsf{worst}}) + N\cdot\mu_{\mathsf{worst}}\right)$
– *expected communication* $C\left(F_{\mathcal{P}}^{-1}\left(\frac{\mu_{\mathsf{avg}}-\mu_{\mathsf{worst}}}{1-\mu_{\mathsf{worst}}}\right)\right)\cdot(1-\mu_{\mathsf{worst}}) + C(N)\cdot\mu_{\mathsf{worst}}$.

Our construction is conceptually simple: we start by instantiating a distributional-PIR scheme that is fast but has a high failure probability, then show how to combine it with a standard batch-PIR scheme to boost correctness at the cost of performance.

Throughout the rest of the chapter, we assume that $\mu_{\mathsf{avg}} > \mu_{\mathsf{worst}}$, since otherwise the distributional-PIR scheme degrades to one with just worst-case correctness.

4.1 A fast, but errorful, distributional PIR scheme.

We start by constructing a fast distributional PIR scheme that can achieve any value of average-case correctness (i.e., $\mu_{\mathsf{avg}} \in [0, 1]$), but has no worst-case guarantees (i.e., $\mu_{\mathsf{worst}} = 0$). It works by choosing k elements of the database whose total probability mass under $\mathcal{P}$ is at least μ_{avg}, then it runs standard batch-PIR over those elements. In order to minimize k (equiv. the server running time), it greedily selects the most popular database elements. We give the full details in Construction 4.1.1.

Construction 4.1.1 (A fast distributional PIR scheme)**.** The construction is parameterized by constants $\mu_{\mathsf{avg}}, \mu_{\mathsf{worst}} \in [0,1]$, database size N, message space $\mathcal{M}$, batch size B, and a standard batch-PIR scheme (Section 3.1) $(\mathsf{Setup}, \mathsf{Encode}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Recover})$ for message space $\mathcal{M}$ and batch size B.

```
$\mathsf{Dist.Setup}(\mathcal{P}) \to \mathsf{pp}$.
– Compute $k \leftarrow F_{\mathcal{P}}^{-1}(\mu_{\mathsf{avg}})$
– Run $\mathsf{pp}_{\mathsf{batch}} \leftarrow \mathsf{Setup}(1^k)$
– Output $(\mathsf{pp}_{\mathsf{batch}}, k)$

$\mathsf{Dist.Encode}(\mathsf{pp}, \mathcal{P}, D) \to D_{\mathsf{code}}$.
– Parse $\mathsf{pp} \to (\mathsf{pp}_{\mathsf{batch}}, k)$
– Let $D' = (d'_1, d'_2, \ldots, d'_N)$ be $D$ sorted according to $\mathcal{P}$
– Run $D_{\mathsf{code}} \leftarrow \mathsf{Encode}((d'_1, d'_2, \ldots, d_k))$
– Output $D_{\mathsf{code}}$

$\mathsf{Dist.Query}(\mathsf{pp}, I) \to (\mathsf{st}, q)$.
– Parse $\mathsf{pp} \to (\mathsf{pp}_{\mathsf{batch}}, k)$
– For all $j \in [B]$, compute $\mathcal{E} \leftarrow \{j \,:\, I_j \notin [k]\}$
– For all $j \in \mathcal{E}$, set $I_j = 1$
– Compute $(\mathsf{st}, q) \leftarrow \mathsf{Query}(\mathsf{pp}_{\mathsf{batch}}, I)$
– Output $((\mathsf{st}, \mathcal{E}), q)$

$\mathsf{Dist.Answer}^{D_{\mathsf{code}}}(q) \to a$.
– Output $\mathsf{Answer}^{D_{\mathsf{code}}}(q)$

$\mathsf{Dist.Recover}(\mathsf{st}, a) \to (m_1, \ldots, m_B)$.
– Parse $\mathsf{st} \to (\mathsf{st}, \mathcal{E})$
– Compute $(m_1, \ldots, m_B) \leftarrow \mathsf{Recover}(\mathsf{st}, a)$.
– For all $j \in \mathcal{E}$, set $m_j = \bot$
– Output $(m_1, \ldots, m_B)$
```

Because $\mathcal{P}$ is a discrete distribution, it may be the case that $\mu_{\mathsf{avg}} \neq F_{\mathcal{P}}(F_{\mathcal{P}}^{-1}(\mu_{\mathsf{avg}}))$. For simplicity of exposition, we assume that this relation holds for any μ_{avg} and $\mathcal{P}$ that we instantiate Construction 4.1.1 with. If this assumption doesn't hold, then to get the optimal runtime Construction 4.1.1 should always build queries over the top $k-1$ elements and only include the k-th element with some probability < 1 needed to satisfy correctness.

Lemma 4.1.2. *For all constants $\mu_{\mathsf{avg}} \in [0,1]$, message space M, batch size B, and δ-secure, errorless batch-PIR scheme Π_{batch} such that:*

- Π_{batch} *is parameterized by a message space M, and batch size B. On input a database of size K, Π_{batch} has server runtime $\widetilde{O}(K)$ and communication cost $C_{\mathsf{batch}}(K)$.*

Construction 4.1.1 is a δ-secure distributional PIR scheme that, on any database $D \in M^*$ and probability distribution $\mathcal{P}$, has

- *explicit correctness error* 0
- *worst-case correctness 0*
- *average-case correctness μ_{avg}*
- *expected server running time $\widetilde{O}(F_{\mathcal{P}}^{-1}(\mu_{\mathsf{avg}}))$, and*
- *expected communication $C_{\mathsf{batch}}(F_{\mathcal{P}}^{-1}(\mu_{\mathsf{avg}}))$.*

Proof. Construction 4.1.1 simply runs Π_{batch} on the $k := F_{\mathcal{P}}^{-1}(\mu_{\mathsf{avg}})$ most popular elements

Construction 4.2.1 (Combining distributional and batch PIR schemes)**.** The construction is parameterized by a constant $\alpha \in [0,1]$, a distributional-PIR scheme Π_{dist}, and a batch-PIR scheme Π_{batch}, both defined over a database size N, message space $\mathcal{M}$, and batch size B.

```
$\mathsf{Dist.Setup}(\mathcal{P}) \to \mathsf{pp}$.
– Compute $\mathsf{pp}_{\mathsf{dist}} \leftarrow \Pi_{\mathsf{dist}}.\mathsf{Setup}(\mathcal{P})$
– Compute $\mathsf{pp}_{\mathsf{batch}} \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Setup}(1^N)$
– Output $(\mathsf{pp}_{\mathsf{dist}}, \mathsf{pp}_{\mathsf{batch}})$

$\mathsf{Dist.Encode}(\mathsf{pp}, \mathcal{P}, D) \to D_{\mathsf{code}}$.
– Parse $\mathsf{pp} \to (\mathsf{pp}_{\mathsf{dist}}, \mathsf{pp}_{\mathsf{batch}})$
– Compute $D_{\mathsf{dist}} \leftarrow \Pi_{\mathsf{dist}}.\mathsf{Encode}(\mathsf{pp}_{\mathsf{dist}}, \mathcal{P}, D)$
– Compute $D_{\mathsf{batch}} \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Encode}(\mathsf{pp}_{\mathsf{batch}}, D)$
– Output $(D_{\mathsf{dist}}, D_{\mathsf{batch}})$

$\mathsf{Dist.Query}(\mathsf{pp}, I) \to (\mathsf{st}, q)$.
– Parse $\mathsf{pp} \to (\mathsf{pp}_{\mathsf{dist}}, \mathsf{pp}_{\mathsf{batch}})$
– Sample a bit $b \leftarrow \mathsf{Bernoulli}(1-\alpha)$
  ∘ If $b = 0$, $(\mathsf{st}, q) \leftarrow \Pi_{\mathsf{dist}}.\mathsf{Query}(\mathsf{pp}_{\mathsf{dist}}, I)$
  ∘ Else, $(\mathsf{st}, q) \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Query}(\mathsf{pp}_{\mathsf{batch}}, I)$
– Output $((b, \mathsf{st}), (b, q))$

$\mathsf{Dist.Answer}^{D_{\mathsf{code}}}(q) \to a$.
– Parse $D_{\mathsf{code}} \to (D_{\mathsf{dist}}, D_{\mathsf{batch}})$, $q \to (b, q)$
  ∘ If $b = 0$, $a \leftarrow \Pi_{\mathsf{dist}}.\mathsf{Answer}^{D_{\mathsf{dist}}}(q)$
  ∘ Else, $a \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Answer}^{D_{\mathsf{batch}}}(q)$
– Output $a$

$\mathsf{Dist.Recover}(\mathsf{st}, a) \to (m_1, \ldots, m_B)$.
– Parse $\mathsf{st} \to (b, \mathsf{st})$
  ∘ If $b = 0$, $(m_1, \ldots, m_B) \leftarrow \Pi_{\mathsf{dist}}.\mathsf{Recover}(\mathsf{st}, a)$
  ∘ Else, $(m_1, \ldots, m_B) \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Recover}(\mathsf{st}, a)$
– Output $(m_1, \ldots, m_B)$
```

of the database. As a result, explicit correctness, expected server-time and communication follow directly from the properties of Π_{batch}. When $k < N$–the regime where the scheme is useful–some database elements are never probed (i.e. the server can't answer some queries), so the worst-case correctness is 0. By construction, the probability mass of the popular subset is $\sum_{i=1}^{k} p_i \geq \mu_{\mathsf{avg}}$. Since queries are drawn i.i.d. from $\mathcal{P}$, in expectation $\geq \mu_{\mathsf{avg}} \cdot B$ queries fall into the popular subset and are recovered successfully, thus, the average-case correctness is μ_{avg}. Finally, security reduces directly to the underlying security of Π_{batch}. □

4.2 Trading performance for correctness

Here we show how to reduce the correctness error of a distributional-PIR scheme by combining it with a standard batch-PIR scheme. The idea is to randomly choose between the two schemes: for each query, with probability α the client uses the distributional-PIR scheme, and with probability $1-\alpha$ they use a standard batch-PIR scheme. This results in a scheme whose parameters are a linear combination of the two schemes: effectively trading off the performance of the distributional-PIR scheme for the correctness of the batch-PIR scheme. We give the full details in Construction 4.2.1.

Lemma 4.2.2. *Let Π_{dist} a distributional-PIR scheme and Π_{batch} be a batch-PIR scheme, both defined over a common message space $\mathcal{M}$, database size N, and batch size B where:*

- Π_{dist} *is* δ_{dist}*-secure with explicit correctness error* ϵ*, worst-case correctness* μ_{worst}*, average-case correctness error* μ_{avg}*, expected server runtime* T_{dist}*, and expected communication* C_{dist}*.*
- Π_{batch} *is* δ_{batch}*-secure with correctness error* $\epsilon_{\mathsf{batch}}$*, server runtime* T_{batch}*, and communication cost* C_{batch}*.*

Then, for every $\alpha \in [0,1] \subseteq \mathbb{R}$*, Construction 4.2.1 is a* $(\delta_{\mathsf{dist}} + \delta_{\mathsf{batch}})$*-secure distributional-PIR scheme over message space* $\mathcal{M}$*, database size* N*, and batch size* B *with*

- *explicit correctness error* $(\alpha\epsilon + (1-\alpha)\epsilon_{\mathsf{batch}})$*,*
- *worst-case correctness* $\alpha\mu_{\mathsf{worst}} + (1-\alpha)\cdot(1-\epsilon_{\mathsf{batch}})$*,*
- *average-case correctness* $\alpha\mu_{\mathsf{avg}} + (1-\alpha)\cdot(1-\epsilon_{\mathsf{batch}})$*,*
- *expected server time* $\alpha T_{\mathsf{dist}} + (1-\alpha)T_{\mathsf{batch}}$*, and*
- *expected communication cost* $\alpha C_{\mathsf{dist}} + (1-\alpha)C_{\mathsf{batch}}$*.*

Proof. Explicit correctness follows directly from the correctness properties of the underlying PIR schemes. Average-case correctness, expected server time, and expected communication cost all follow directly by linearity of expectation. Worst-case correctness is derived by taking a union-bound over either PIR scheme failing.

To show security, fix some constant $\alpha \in [0,1]$. We show that Π is $(\delta_{\mathsf{dist}} + \delta_{\mathsf{batch}})$-secure via a hybrid argument. Let $\mathsf{Sec}_{\Pi}(\cdot,\cdot)$ denote the output of the distributional-PIR security Experiment 3.2.1, and let $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ be any valid adversary for the experiment. Define the following distributions:

- H_0: This is $\mathsf{Sec}_{\Pi}(\mathcal{A}, 0)$.
- H_1: The same as H_0 except, for each query (b,q), if $b=0$, then the query q is computed for I_1 rather than I_0. We highlight the difference between H_0 and H_1 below:

$$\mathrm{H}_1 := \left\{ \begin{array}{l} (\mathsf{st}, \mathcal{P}, D, I_0, I_1) \leftarrow \mathcal{A}_0() \\ (\mathsf{pp}_{\mathsf{dist}}, \mathsf{pp}_{\mathsf{batch}}) \leftarrow \Pi.\mathsf{Setup}(\mathcal{P}) \\ b \leftarrow \mathsf{Bernoulli}(1-\alpha) \\ \quad \circ \text{ If } b=0,\ (\_, q) \leftarrow \Pi_{\mathsf{dist}}.\mathsf{Query}(\mathsf{pp}_{\mathsf{dist}}, I_1) \\ \quad \circ \text{ Else, } (\_, q) \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Query}(\mathsf{pp}_{\mathsf{batch}}, I_0) \\ \text{Output } \mathcal{A}_1(\mathsf{st}, (\mathsf{pp}_{\mathsf{dist}}, \mathsf{pp}_{\mathsf{batch}}), (b,q)) \end{array} \right\}$$

- H_2: This is $\mathsf{Sec}_{\Pi}(\mathcal{A}, 1)$.

For $i \in [2]$, let W_i be the event that H_i outputs 1.

Claim 4.2.3. $|\Pr[W_0] - \Pr[W_1]| \le \delta_{\mathsf{dist}}$.

Proof. Using $\mathcal{A}$ we construct an adversary $\mathcal{B} = (\mathcal{B}_0, \mathcal{B}_1)$ against the security of Π_{dist}. $\mathcal{B}$ works as follows:
Observe that:

- when $b=1$, $\mathcal{B}$ always outputs 0, and
- when $b=0$, then $\mathcal{B}$ perfectly simulates either H_0 or H_1 depending on whether it receives a query for I_0 or I_1.

```
$\mathcal{B}_0$
– $(\mathsf{st}_{\mathcal{A}}, \mathcal{P}, I_0, I_1) \leftarrow \mathcal{A}_0()$
– Output $((\mathsf{st}_{\mathcal{A}}, I_0), \mathcal{P}, I_0, I_1)$
```

```
$\mathcal{B}_1(\mathsf{st}, \mathsf{pp}_{\mathsf{dist}}, q)$
– Parse $\mathsf{st} \to (\mathsf{st}_{\mathcal{A}}, I_0)$
– Sample a bit $b \leftarrow \mathsf{Bernoulli}(1-\alpha)$
– Compute $\mathsf{pp}_{\mathsf{batch}} \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Setup}(1^N)$
– If $b = 0$:
    ∘ Output $\mathcal{A}_1(\mathsf{st}, (\mathsf{pp}_{\mathsf{batch}}, \mathsf{pp}_{\mathsf{dist}}), (0, q))$
– Else:
    ∘ Compute $(\_, q') \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Query}(\mathsf{pp}_{\mathsf{batch}}, I_0)$
    ∘ Run $\mathcal{A}_1(\mathsf{st}, (\mathsf{pp}_{\mathsf{batch}}, \mathsf{pp}_{\mathsf{dist}}), (1, q'))$
    ∘ Output 0
```

As a result:

$$\begin{aligned}\mathsf{DistAdv}[\mathcal{B}, \Pi_{\mathsf{dist}}] &= \left|\Pr[\mathsf{Sec}_{\Pi_{\mathsf{dist}}}(\mathcal{B}, 0) = 1] - \Pr[\mathsf{Sec}_{\Pi_{\mathsf{dist}}}(\mathcal{B}, 1) = 1]\right| \\ &= \left|\Pr[W_0 \mid b = 0]\Pr[b = 0] - \Pr[W_1 \mid b = 0]\Pr[b = 0]\right|.\end{aligned}$$

Conditioned on $b = 1$, H_0 and H_1 are identical by construction, so the following holds:

$$\begin{aligned}\left|\Pr[W_0] - \Pr[W_1]\right| &= |(\Pr[W_0 \mid b = 0]\Pr[b = 0] + \Pr[W_0 \mid b = 1]\Pr[b = 1]) \\ &\quad - (\Pr[W_1 \mid b = 0]\Pr[b = 0] + \Pr[W_1 \mid b = 1]\Pr[b = 1])| \\ &= \left|\Pr[W_0 \mid b = 0]\Pr[b = 0] - \Pr[W_1 \mid b = 0]\Pr[b = 0]\right|.\end{aligned}$$

Thus, $\mathsf{Dist}[\mathcal{B}, \Pi_{\mathsf{dist}}] = |\Pr[W_0] - \Pr[W_1]|$. Since Π_{dist} is δ_{dist}-secure, then this implies that

$$|\Pr[W_0] - \Pr[W_1]| < \delta_{\mathsf{dist}}.$$ □

By an analogous argument, it also holds that $|\Pr[W_1] - \Pr[W_2]| \leq \delta_{\mathsf{batch}}$. Thus, by the triangle inequality, we have $\mathsf{DistAdv}[A, \Pi] \leq \delta_{\mathsf{dist}} + \delta_{\mathsf{batch}}$. □

4.3 Main Construction

We are now ready to construct a distributional-PIR scheme Π that satisfies Theorem 4.0.1. Recall, that Theorem 4.0.1 assumes the existence of some δ-secure, errorless batch-PIR scheme Π_{batch}. Using Π_{batch}, we construct Π in two steps:

1. We build a fast distributional-PIR scheme Π_{dist} by instantiating Construction 4.1.1 using Π_{batch} with average-case correctness $\frac{\mu_{\mathsf{avg}} - \mu_{\mathsf{worst}}}{1 - \mu_{\mathsf{worst}}}$.
2. We instantiate Construction 4.2.1 using Π_{dist} and Π_{batch} with $\alpha = 1 - \mu_{\mathsf{worst}}$.

We now justify the choice of parameters for each step. In step (2), setting $\alpha = 1 - \mu_{\mathsf{worst}}$ ensures that the resulting scheme satisfies worst-case correctness μ_{worst}. For step (1), we want to find the minimum number of popular elements k that Construction 4.1.1 should probe to achieve the target average-case correctness μ_{avg}. From the definition of Construction 4.1.1

and Construction 4.2.1, the probability that an element sampled from $\mathcal{P}$ is answered correctly by our construction is $\mu_{\mathsf{worst}} \cdot 1 + (1 - \mu_{\mathsf{worst}}) \cdot F_{\mathcal{P}}(k)$. So we need:

$$\begin{aligned}
\mu_{\mathsf{worst}} \cdot 1 + (1 - \mu_{\mathsf{worst}}) \cdot F_{\mathcal{P}}(k) &\geq \mu_{\mathsf{avg}} \\
F_{\mathcal{P}}(k) &\geq \frac{\mu_{\mathsf{avg}} - \mu_{\mathsf{worst}}}{1 - \mu_{\mathsf{worst}}} \\
k &\geq F_{\mathcal{P}}^{-1}(\frac{\mu_{\mathsf{avg}} - \mu_{\mathsf{worst}}}{1 - \mu_{\mathsf{worst}}})
\end{aligned}$$

Thus, setting the average-case correctness of Construction 4.1.1 to be $\frac{\mu_{\mathsf{avg}} - \mu_{\mathsf{worst}}}{1 - \mu_{\mathsf{worst}}}$ minimizes the number of popular elements probed.

The full details of Π are specified in Construction 4.3.2. Note that the specified batch-PIR scheme Π_{batch} can be constructed via standard techniques, e.g., by combining a batch code [48] with a standard PIR scheme.

Claim 4.3.1. Construction 4.3.2 is a 2δ-secure distributional PIR scheme with
– explicit correctness error 0,
– average-case correctness μ_{avg},
– worst-case correctness μ_{worst},
– expected communication cost $C_{\mathsf{batch}}\left(F_{\mathcal{P}}^{-1}\left(\frac{\mu_{\mathsf{avg}} - \mu_{\mathsf{worst}}}{1 - \mu_{\mathsf{worst}}}\right)\right) \cdot (1 - \mu_{\mathsf{worst}}) + C_{\mathsf{batch}}(N) \cdot \mu_{\mathsf{worst}}$, and
– expected server runtime $\widetilde{O}\left(F_{\mathcal{P}}^{-1}\left(\frac{\mu_{\mathsf{avg}} - \mu_{\mathsf{worst}}}{1 - \mu_{\mathsf{worst}}}\right) \cdot \epsilon_{\mathsf{worst}} + N \cdot \mu_{\mathsf{worst}}\right)$.

Proof. The proof follows directly from the properties of Π_{batch} and **??** and Lemma 4.2.2. □

Construction 4.3.2 (Construction from Theorem 4.0.1)**.** The construction is parameterized by constants $\mu_{\mathsf{avg}}, \mu_{\mathsf{worst}} \in [0,1]$, a database size N, and a batch-PIR scheme Π_{batch} defined over a message space $\mathcal{M}$, and batch size B.

```
Dist.Setup(P) → pp.
– Compute $k \leftarrow F_{\mathcal{P}}^{-1}\left(\frac{\mu_{\mathsf{avg}}-\mu_{\mathsf{worst}}}{1-\mu_{\mathsf{worst}}}\right)$
– Compute $\mathsf{pp}_1 \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Setup}(1^k)$
– Compute $\mathsf{pp}_2 \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Setup}(1^N)$
– Output $(\mathsf{pp}_1, \mathsf{pp}_2, k)$

Dist.Encode(pp, P, D) → D_code.
– Parse $\mathsf{pp} \to (\mathsf{pp}_1, \mathsf{pp}_2, k)$
– Let $D' = (d'_1, d'_2, \ldots, d'_N)$ be $D$ sorted according to $\mathcal{P}$
– $D^1_{\mathsf{code}} \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Encode}((d'_1, d'_2, \cdots, d'_k))$
– $D^2_{\mathsf{code}} \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Encode}(D)$
– Output $(D^1_{\mathsf{code}}, D^2_{\mathsf{code}})$

Dist.Query(pp, I) → (st, q).
– Parse $\mathsf{pp} \to (\mathsf{pp}_1, \mathsf{pp}_2, \_)$
– Sample a bit $b \leftarrow \mathsf{Bernoulli}(\mu_{\mathsf{worst}})$
  ∘ If $b = 0$, $(\mathsf{st}, q) \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Query}(\mathsf{pp}_1, I)$
  ∘ Else, $(\mathsf{st}, q) \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Query}(\mathsf{pp}_2, I)$
– Output $((b, \mathsf{st}), (b, q))$

Dist.Answer(D_code, q) → a.
– Parse $D_{\mathsf{code}} \to (D_{\mathsf{dist}}, D_{\mathsf{batch}})$, $q \to (b, q)$
  ∘ If $b = 0$, $a \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Answer}(D_{\mathsf{dist}}, q)$
  ∘ Else, $a \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Answer}(D_{\mathsf{batch}}, q)$
– Output $a$

Dist.Recover(st, a) → (m_1, ..., m_B).
– Parse $\mathsf{st} \to (b, \mathsf{st})$
  ∘ If $b = 0$, $(m_1, \ldots, m_B) \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Recover}(\mathsf{st}, a)$
  ∘ Else, $(m_1, \ldots, m_B) \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Recover}(\mathsf{st}, a)$
– Output $(m_1, \ldots, m_B)$
```

Chapter 5

Distributional PIR: Deployment Considerations

In this section we discuss practical issues that arise when using distributional-PIR schemes.

5.1 Measuring the popularity distribution

To implement a distributional-PIR scheme, the PIR server must have a good approximation of the popularity distribution $\mathcal{P}$. When the server has access to a log of client queries to a dataset—as we have for Twitter—the server can use the query log to estimate $\mathcal{P}$.

In a world in which PIR is ubiquitous, however, the server learns no information about which client is querying which record, and thus the server learns no information about the popularity distribution via client queries. We sketch ways for the server to measure the distribution, even when it does not see client queries; the best one to use depends on the application.

External information. In some cases, the PIR server can use external information to surmise the distribution $\mathcal{P}$. For example, when using PIR in the context of auditing in Certificate Transparency [51], the database contains one record per website (specifically one per signed certificate timestamp). Clients request records in proportion to how often they visit a particular website. If the server can externally obtain information about which websites are popular (e.g., via the Alexa list of top websites), then it can use this to estimate the distribution $\mathcal{P}$.

Private measurement. A second option is for the PIR server to use existing schemes for private aggregation [52] to gather statistics on which database entries are popular. This would allow the server to learn the aggregate distribution $\mathcal{P}$ without learning anything about the behavior of any individual client. To ensure that the computational overhead of the measurement infrastructure does not end up being a bottleneck, clients could participate in the private-measurement protocol only occasionally—e.g., once per day or week.

Setups	*Expected Communication*		*Expected Server Runtime*
	Params	Per-Query	Per-Query
Naive	$N\ell$	0	0
Top-k Entries	$(\log N + \ell) \cdot F_{\mathcal{P}}^{-1}(\frac{\mu_{\mathsf{avg}} - \mu_{\mathsf{worst}}}{1 - \mu_{\mathsf{worst}}})$	$Q - (1 - \mu_{\mathsf{worst}}) \cdot C(F_{\mathcal{P}}^{-1}(\frac{\mu_{\mathsf{avg}} - \mu_{\mathsf{worst}}}{1 - \mu_{\mathsf{worst}}}), \ell)$	$R - (1 - \mu_{\mathsf{worst}}) \cdot \ell \cdot F_{\mathcal{P}}^{-1}(\frac{\mu_{\mathsf{avg}} - \mu_{\mathsf{worst}}}{1 - \mu_{\mathsf{worst}}})$
Top-k Indices	$\log N \cdot F_{\mathcal{P}}^{-1}(\frac{\mu_{\mathsf{avg}} - \mu_{\mathsf{worst}}}{1 - \mu_{\mathsf{worst}}})$	Q	R
Sorted	$\log N$	Q	R
Recursive PIR	$\log N$	$Q + C(N, \log N)$	$R + N \log N$

Table 1: Big-O asymptotic costs associated with Construction 4.3.2 on a message space $\mathcal{M}$, database size N, popularity distribution $\mathcal{P}$, batch size $B = 1$, and constants $\mu_{\mathsf{avg}}, \mu_{\mathsf{worst}} \in [0, 1]$ when using the various setups described in Section 5.2. Let $\ell = \log \mathcal{M}$ denote the size of a database entry. $C(p, q)$ returns the per-query communication cost of a standard PIR scheme on a database with p entries each consisting of q bits. We abbreviate $Q = \mu_{\mathsf{worst}} \cdot C(N, \ell) + (1 - \mu_{\mathsf{worst}}) \cdot C(k, \ell)$ and $R = \mu_{\mathsf{worst}} \cdot N \cdot \ell + (1 - \mu_{\mathsf{worst}}) \cdot k \cdot \ell$.

5.2 How the client learns the distribution

A second implementation concern is that distributional-PIR schemes have some public parameters that the client must obtain before making any PIR queries. The public parameters can potentially be as large as the representation of the entire distribution $\mathcal{P}$, which could be as large as the database itself.

Below, we sketch several choices of public parameters for Construction 4.3.2 and in Table 1 compare their asymptotic tradeoffs for batch size $B = 1$. For each of the setups, we assume that the client's indices correspond to the original database before it's been sorted by popularity unless stated otherwise.

In several of these schemes, the client downloads some database-dependent information. For simplicity, we describe this information as part of the public parameters—in practice it would be communicated as a one-time cost via $\Pi.\mathsf{Answer}$ since $\Pi.\mathsf{Setup}$ doesn't take the database as input.

Setup 1 (Naive). The most naive PIR scheme sets $\mathsf{pp} = D$. This allows the client to answer all of their queries locally without communicating with the server but requires large amounts of communication and storage.

Setup 2 (Download the top-k database entries). For databases that are too large to download, the client can instead download the k most-popular entries of the database and their corresponding indices. In other words, if D_{code} is D sorted according to $\mathcal{P}$, then $\mathsf{pp} = (\mathcal{P}[1..k], D_{\mathsf{code}}[1..k])$. After this, the client only needs to communicate with the server to answer queries to the entire database. Note that, in order to retain security, the client must still follow the protocol honestly (e.g. query the server with some probability even if all of their queries fall into the top-k entries).

Setup 3 (Download the top-k database indices). To reduce the download even more, the client can just download the *indices* of the k most-popular entries. In other words, if $\mathcal{P}$ is a popularity distribution, $\mathsf{pp} = \mathcal{P}[1..k]$. If the server keeps the database unsorted, then this information is sufficient for the client to query any index.

Setup 4 (Database is sorted by popularity). If the database is sorted by popularity *before* the client chooses their indices of interest, then the public parameters only need to specify the cutoff point k. In other words, $\mathsf{pp} = k$.

Setup 5 (Recursive PIR). Finally, if the entries of the database D are $\ell \gg \log N$ bits long, then the client can use standard batch-PIR [48] to fetch the position of their desired indices after sorting for much cheaper than a standard batch-PIR query to the entire database. In other words, the server encodes the popularity distribution $\mathcal{P}$ as another database that a client queries using PIR before building their distributional-PIR query.

Moreover, in settings such as our Twitter application (Chapter 7), the popularity distribution $\mathcal{P}$ changes slowly and the client repeatedly fetches the ith entry of the database—the tweets from user i. In this case, the client can make one batch query to the popularity distribution and reuse these bits over many future queries.

Chapter 6

Optimizations to existing PIR schemes

Our distributional PIR construction (Construction 4.3.2) makes black-box use of a standard PIR scheme. In this section, we describe new optimizations to lattice-based single-server private-information-retrieval schemes.

We develop a new approach to *linearly homomorphic encryption with preprocessing* [8], the core building block behind many of the fastest PIR schemes [7], [8], [11], [12]. Using a linearly homomorphic encryption scheme, a client can encrypt a matrix $\mathbf{X}$ and a server can multiply the encrypted value by a matrix $\mathbf{D}$ under encryption. By preprocessing the matrix $\mathbf{D}$, the server can speed up the encrypted matrix-matrix product. In the context of PIR, the client represents its query as a matrix $\mathbf{X}$, which it encrypts and sends to the server. The server represents the database as a matrix $\mathbf{D}$, computes the matrix-matrix product $\mathbf{D} \cdot \mathsf{Enc}(\mathbf{X})$ under encryption, and returns the result to the client.

The most important cost metrics in PIR are:

– the communication cost—i.e., the ciphertexts size, and
– the server computation cost—i.e., how much work the server has to do to compute the encrypted matrix-matrix product.

Prior linearly homomorphic encryption schemes either have small communication cost *or* small computation cost. We construct an encryption scheme that attempts to get the "best of both."

6.0.1 Background: encryption with lattices

The most computationally efficient linearly homomorphic encryption schemes are lattice-based and typically come in one of two flavors:

– **Learning-with-errors (LWE) based.** Computing a homomorphic matrix-vector product with LWE-based schemes [7], [8], [53] under encryption can just require computing a matrix-vector product on 32- or 64-bit integers modulo 2^{32} or 2^{64}. (That is, the "ciphertext modulus" is 2^{32} or 2^{64}.)
– **RingLWE based.** RingLWE-based encryption systems have much more compact ciphertexts—roughly $n \approx 2^{11}\times$ smaller than Regev LWE-based ciphertexts, where n is the size of the Regev secret key. These schemes also have faster encryption and decryption operations (again roughly $n\times$ faster). In addition, the fastest single-server PIR schemes [7], [8] re-

quire the server to perform some database-dependent preprocessing—this is also roughly $n\times$ faster with ring-based encryption schemes.

We construct a "hybrid" linearly homomorphic encryption scheme that achieves the benefits of both RingLWE systems—small ciphertext and fast encryption, decryption, and preprocessing— while allowing the server to perform the homomorphic matrix-matrix product with an LWE-style modulus of 2^{32} or 2^{64}. A number of recent works on PIR [9], [11], [12] also mix LWE- and RingLWE-based encryption schemes, but only achieve a subset of these features.

At a high-level, our scheme works as follows:

1. The client encrypts its query matrix $\mathbf{X}$ using a RingLWE-based encryption scheme with a modulus one-bit larger than 2^{32} or 2^{64}.
2. **Our contribution:** The client converts the RingLWE ciphertext to an LWE ciphertext, switches the modulus to 2^{32} or 2^{64}, and sends it to the server.
3. The server performs the homomorphic matrix-matrix product on the client's ciphertext modulo 2^{32} or 2^{64}. The server compresses the LWE ciphertext back into an RingLWE ciphertext, as in prior work [9], [11], [12], [43], to return to the client.
4. The client uses fast RingLWE-based decryption to recover the plaintext.

Step 2 of the above outline is the key new step. To implement it, we use modulus switching [24], [54], a standard technique to transform an LWE/RingLWE ciphertext encrypted using some modulus q_1 to a new smaller modulus q_2.

Our use of modulus switching departs from prior work. Prior work uses modulus switching *after* homomorphic evaluation, to reduce communication or to shave off the low-order bits of the plaintext [55]. In contrast, we have the client modulus switch *before* homomorphic evaluation takes place: after encryption under a RingLWE-based scheme using prime modulus q_1, the client converts the ring ciphertext to a set of LWE ciphertexts using standard techniques [56] and modulus switches the resulting ciphertext to a new modulus q_2. By setting q_2 to be a word-size, e.g. $q_2 \in \{2^{32}, 2^{64}\}$, the resulting scheme enjoys all the performance benefits of the pure LWE-based scheme with a few small costs:

1. *Modulus-switching time:* For a 32-bit ciphertext modulus, modulus switching consists of a single floating-point multiplication and rounding operation for each input element. Concretely, in our experiments the per-element cost is tens of nanoseconds, amounting to a total $< 5\%$ increase in the running time of encryption compared to pure RingLWE-based encryption.
2. *Noise growth:* Modulus switching before evaluation slightly *increases* the total noise in the ciphertext. Practically, the added noise is negligible: in our experiments, it decreases the size of the usable plaintext space by at most 0.1 bits.

Offloading work to a GPU. When using our encryption scheme for PIR, the server's main work is computing the homomorphic matrix-matrix product. After applying our modulus-switching optimization, the server's computation is so cheap that the bottleneck ends up being the speed with which the processor can read in the database from main memory.

After our modulus-switching approach, the PIR server's work is just to compute a very large matrix-matrix product of integers modulo 2^{32} or 2^{64}. Our idea then is simple: offload the computation to a GPU, which has orders of magnitude more memory bandwidth than

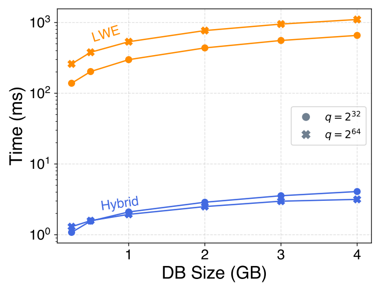

Figure 1: Client time to encrypt a vector for linear evaluation on a database of increasing size.

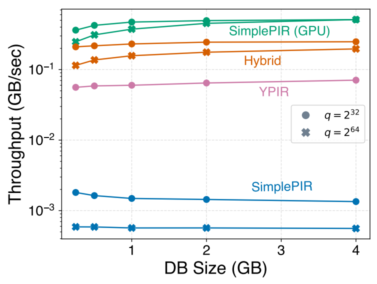

Figure 2: Server throughput when preprocessing a hint for a database of increasing size. The GPU instance is 8× more expensive to run than the other schemes. YPIR doesn't support 64-bit ciphertext moduli.

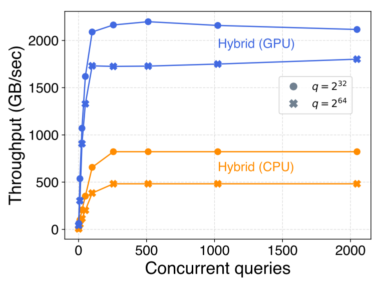

Figure 3: Server throughput when answering batches of client queries on a 4GB database. The CPU throughput is a cluster of 8 machines each with eight cores. The deployment cost of the CPU-based cluster and singular GPU is the same.

a CPU does. The LWE homomorphic-evaluation scheme we use makes black box use of existing, highly-optimized, matrix-multiplication libraries.

6.0.2 Microbenchmarks

We evaluate the performance of our optimizations on several microbenchmarks using ciphertext moduli $q = 2^{32}$ and $q = 2^{64}$ with respective lattice security parameters $n = 2048$ and $n = 4096$. In our hybrid scheme, we set the ciphertext modulus before modulus switching to be one bit larger than the target modulus. We evaluate multiple choices of moduli as they are useful in different settings: $q = 2^{32}$ is almost always faster to compute on, but $q = 2^{64}$ allows for a much larger plaintext space. Many applications, such as secure inference [41], [42] or private web search [12], necessitate the larger plaintext space.

We run on `c7.2xlarge` AWS instances (8 vCPUs and 16GB RAM) for CPU-based computation, and `p3.2xlarge` AWS instances (NVIDIA V100 w/ 16GB RAM) for GPU-based computation. The GPU instance costs $\approx 8\times$ more than the CPU instance.

Query Latency. In Figure 1 we compare the latency of generating a query using our scheme compared to LWE-based alternatives. Concretely, our scheme is 128–161× faster for 32-bit ciphertext moduli and 200–349× faster for 64-bit ciphertext moduli, with the gap monotonically increasing with database size. The performance of query generation for 32-bit and 64-bit moduli is similar in our scheme due to some implementations details of the Microsoft SEAL library [57].

Preprocessing Throughput. In Figure 2, we evaluate the preprocessing efficiency of our scheme compared against three alternatives: LWE-based schemes on a CPU, LWE-based schemes on a GPU, and the hybrid preprocessing techniques of YPIR [58]. YPIR requires working over a large group that ensures the preprocessing computation doesn't wrap around the modulus. For a 32-bit ciphertext modulus, the only size that YPIR's implementation supports, this leads to a 3.5–3.9× slowdown compared to our scheme.

Similar to query generation, when compared to LWE-based alternatives, our scheme's preprocessing is 116–184× faster for a 32-bit modulus and 195–351× for a 64-bit modulus. While running on a single CPU, our scheme is only 1.7–2.6× slower than LWE-based preprocessing run on a GPU (which costs 8× as much).

Batching requests with GPUs. We evaluate the effectiveness of using GPUs to handle large number of concurrent queries (possibly from different clients). Since GPUs are significantly more expensive than CPUs, we ensure that the deployments have similar costs: comparing the throughput of a single GPU vs. a simulated cluster of eight eight-core CPU-based machines, where each CPU-based machine parallelizes requests across all of its cores and uses cross-client caching techniques [58].

The results of the experiment are shown in Figure 3. For a batch of 50 concurrent requests, one GPU can process roughly 3× more requests per second than the 64-core CPU cluster. As the cost of GPUs falls relative to CPUs, using GPUs for these homomorphic operations will only become a more attractive alternative.

Chapter 7

Private Twitter Feeds with CrowdSurf

We now introduce CrowdSurf, a system that allows a Twitter user to fetch tweets from other users without the Twitter learning who is following whom.

To accomplish this, CrowdSurf combines our distributional PIR construction (Chapter 4) with our optimized PIR scheme (Chapter 6). We demonstrate that CrowdSurf is more cost-effective than alternative approaches.

Architecture. A CrowdSurf deployment consists of:

– a set of users who publish tweets,

– a set of infrastructure servers, which hold the database of all tweets, and

– a set of followers who read tweets.

The goal of CrowdSurf it to allow the followers to fetch tweets from particular user accounts without revealing to the servers which tweets they fetched. Twitter only learns the approximate number of accounts that each user follows and the times at which the user loads their feed.

The principle of CrowdSurf is simple: the infrastructure servers serve as PIR servers, where the PIR database is the set of recent tweets; with one user's tweets in each database record. The client fetches these tweets using PIR. We demonstrate that our techniques reduce the AWS dollar-cost of such a deployment by 8× compared to existing techniques.

Applying distributional PIR. CrowdSurf exploits the fact that some Twitter users are much more popular than others (Chapter 8). This allows us to reduce the server-side computational cost using distributional PIR.

In more detail, for a given batch size and corresponding popularity distribution, CrowdSurf splits the database up into two distinct buckets: a small bucket containing the most popular users' tweets, and a second bucket with the remaining users' tweets. When a client wants to follow a new set of people, they generate a distributional PIR query for the second bucket and send it to the server, who stores it. When a client asks the server to refresh their feed, the server will respond by sending over the entire first bucket in plaintext, and a distributional PIR answer on the second bucket.

PIR optimizations. For a PIR scheme, CrowdSurf uses the linearly homomorphic encryption scheme from Chapter 6 along with the hint-compression techniques of [12], [59] to enable lightweight clients. Since the server computes over RingLWE-based ciphertexts for hint-compression and LWE-based ciphertexts for distributional PIR, we separate this computation into two different clusters: a cluster of CPUs for hint-compression, and a cluster of GPUs for distributional PIR.

Learning the distribution. As we discuss in Section 5.2, the client must learn some information about the popularity distribution to make its PIR queries. For our evaluation, we choose to recursively apply PIR (see Section 5.2) which requires one round of keyword PIR to determine which bucket the client's desired user's tweets lie in. Since the user only needs to make this query when following a new user, and since this metadata database is more than $100\times$ smaller than the tweet database, we ignore this cost in our evaluation.

Chapter 8

Evaluation

We evaluate our main distributional-PIR construction (Construction 4.3.2) using real-world data and demonstrate that it:
– Improves PIR performance on real-world popularity distributions (Section 8.1).
– Provides correctness for real-world clients (Section 8.1).
– Compares favorably to existing batch-PIR schemes (Section 8.1.1).
– Enables private Twitter feeds for 8× less cost than existing approaches (Section 8.2).

8.1 Distributional PIR: Microbenchmarks

Data set. We use real-world data from a 2014 crawl of the Twitter social graph [21] spanning 505 million accounts and 23 billion social connections. We consider a database consisting of users' tweets and a distribution over the popularity of each user (Figure 1). These results demonstrate that distributional PIR improve PIR performance on a real-world dataset.

In the data set, the popularity distribution heavily depends on how many accounts a particular user follows. In particular, users who follow only a few accounts are much more likely to only follow popular accounts compared to those who follow many accounts. To account for this, we build a distinct popularity distribution for each number of accounts a user follows—i.e., for each batch size. (CrowdSurf does not try to hide the number of accounts a user follows from the servers, so the client can reveal this number to the server who can answer using the correct popularity distribution.) For example, when building a distribution for a batch size of 16, we only consider data from users following 8–16 accounts.

Parameter selection. In Figure 3 we demonstrate how the correctness parameters effect performance. Throughout the evaluation we use Construction 4.3.2 with correctness values $\mu_{\mathsf{worst}} = 0.01, \mu_{\mathsf{avg}} = 0.8$. We opt to use a fairly low value of worst-case correctness since very few of the clients in our dataset vary significantly from our popularity distribution.

Validating average-case correctness. We empirically verify that real users from the Twitter dataset obtain the average-case correctness that we predict. In particular, for any choice of parameters for Construction 4.3.2 we sampled a random subset of one million Twitter users and recorded the observed correctness for these users. In all cases, we found

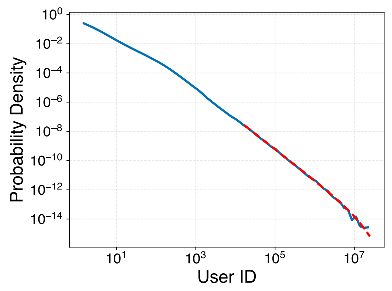

Figure 1: The distribution of Twitter followers per-user (shown in blue) follows a truncated power-law distribution [25] with $\alpha = 2.1$ (shown in red).

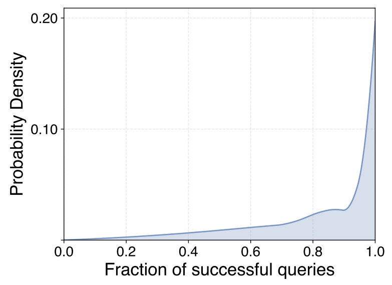

Figure 2: Construction 4.3.2 empirically provides the expected correctness for one million Twitter users making 24–32 queries. For $\mu_{\text{avg}} = 0.8$ and $\mu_{\text{worst}} = 0.01$, the resulting correctness distribution has a mean of 0.8 and a standard deviation of 0.22.

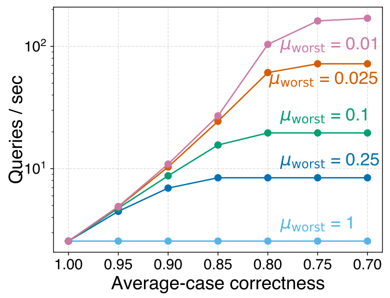

Figure 3: Decreasing correctness allows a distributional-PIR scheme to run in less time. The plot displays queries-per-second as average-case correctness varies when making 24 queries with Construction 4.3.2 on a 4 GB database that follows the Twitter popularity distribution.

that our scheme precisely achieved the desired accuracy. Figure 2 displays the result of one of these experiments.

8.1.1 Comparison to batch-codes

We compare the performance of distributional PIR, against existing techniques for batch PIR from standard batch codes [4], [26]. Batch PIR allows a client to make B queries to an N-record database with total server-side cost $N \cdot \text{polylog}(N)$, rather than the NB cost that naïve repetition would give. We evaluate against two popular codes, one from Ishai et al. [26] that hashes database records into buckets ("Hash") and one from Angel et al. [4] that uses cuckoo hashing instead ("Cuckoo").

Experimental Setup. We evaluate the cost of answering 1–64 queries from a client whose queries follow the Twitter popularity distribution. This batch range accounts for ∼83% of users in our dataset.

Since our distributional-PIR construction is agnostic to the underlying PIR scheme, we experiment with two different choices:

- SimplePIR [8]: this is the fastest single-server PIR scheme, but requires high communication and client state.
- Respire [28]: this is the most communication-efficient single-server PIR scheme for the batch sizes we consider that achieves reasonable performance without client state.

For SimplePIR, we evaluate queries on a 4 GB database with client storage capped at 200MB (5% of the total database size) across all schemes and batch sizes. For Respire, limitations in the implementation at the time of writing restricted us to run on a 1 GB database. Additionally, we only run on batch sizes that are a power-of-two due to some lower-level details of their scheme. In each setting, we instantiate Construction 4.3.2 with the best-performing batch-PIR scheme.

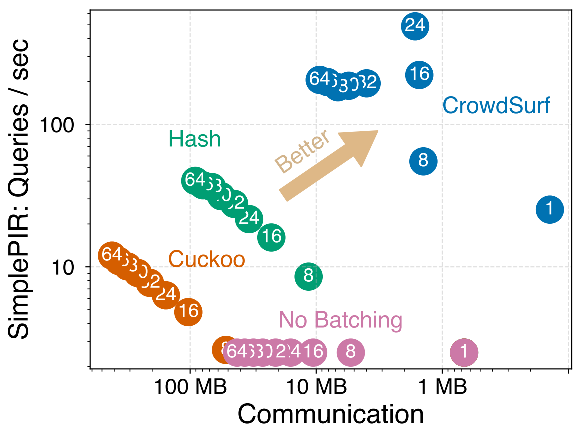

Figure 4: Average queries-per-second vs. total communication when answering a batch of client queries on a 4 GB database that follows the Twitter popularity distribution. All schemes are instantiated with SimplePIR [8] and use ∼200 MB of client storage. The number in each bubble represents the batch size.

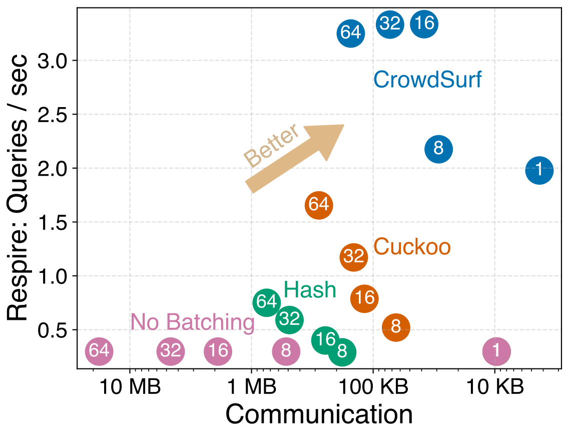

Figure 5: Average queries-per-second vs. total communication when answering a batch of client queries on a 1 GB database that follows the Twitter popularity distribution. All schemes are instantiated with Respire [28]. The number in each bubble represents the batch size.

Since both Construction 4.3.2 and the Hash batch code only recover a fraction of queries on average (∼80% and ∼90% respectively), all of our reported numbers only count *successful* queries. An important caveat to recall is that distributional PIR give weaker correctness guarantees than standard batch-PIR do: *our scheme only recovers* 80% *of queries when a user's queries follow the Twitter popularity distribution*, while the standard batch-PIR schemes achieve correctness no matter the distribution of client's queries.

We run all experiments on an `r7i.4xlarge` AWS instance (16 vCPUs, 128GB RAM). Such a large instance is necessary since the encoding outputted by the Cuckoo-based batch code [4] is upwards of 40 GB.

Performance results. In Figures 4 and 5 we plot the average queries-per-second vs. communication for the two experiments. In summary, when using SimplePIR, CrowdSurf increases the queries-per-second by 10–195× and reduces communication by 4.8–9.7× compared to the baseline that doesn't use batch codes. Compared against batch codes, CrowdSurf increases the queries-per-second by 5.1–77× and reduces communication by 8.1–95×. When using Respire, CrowdSurf increases the queries-per-second by 6.7–12.8× and reduces communication by 2.3–117× compared to the baseline that doesn't use batch codes. Compared against batch codes, CrowdSurf increases the queries-per-second by 2–8.5× and reduces communication by 1.8–9.73×. In both experiments, CrowdSurf improves performance over the baseline even for single queries, unlike batch codes.

Note that CrowdSurf's performance gains don't scale linearly with the batch size because we use a different distribution for each batch size: while the *tails* of these distributions follow a similarly-distributed power-law, the heads exhibit different behavior. For example, the cutoff point of Construction 4.3.2 for users making 16–24 queries is 8× smaller than the cutoff point for users making 56–64 queries. This further demonstrates how the performance

	Hint Compression		***PIR***			
	CPU (core-s)	AWS Costs (US cents)	CPU (core-s)	GPU (s)	AWS Costs (US cents)	**Total Cost**
Batch PIR	3.17	0.034	1.19	–	0.012	0.046
CrowdSurf	0.54	0.0053	–	0.004	0.0003	0.0057

Table 6: Per-request server costs for a single client following 24 users in the 38 GB Twitter database. Each request takes ∼500ms of latency. CrowdSurf clients use 65 MB of server storage and download 21 MB per-request. Batch-PIR clients use 78 MB of server storage and download 34 MB per-request. PIR costs are amortized over batches of simultaneous client requests.

of distributional PIR heavily relies on the underlying distribution.

To better understand the performance of the batch-PIR schemes in Figure 4, recall that in SimplePIR, the hint size scales with the number of rows in the database. In order to keep the total hint size constant after splitting the database into a number of buckets, the server must squish the dimensions of the matrices to be "short", increasing both communication and runtime. While this issue affects both batch codes, it's much more impactful for the Cuckoo batch code since it a) outputs an encoding $3\times$ the size of the original database and b) uses a large number of buckets ($1.5B$). All of that being said, the Cuckoo batch code correctness error is very near zero, so in many settings the lower performance may be acceptable.

In Figure 5, the Cuckoo batch-code outperforms the Hash batch-code thanks to some additional optimizations in the Respire implementation. All of these optimizations are compatible with the Hash batch-code, however we did not implement them ourselves; if one were to do this, we expect that the Hash batch-code would slightly outperform the Cuckoo batch-code in both query throughput and communication (but achieves a weaker notion of correctness).

8.2 CrowdSurf: End-to-end evaluation

We implemented CrowdSurf (Chapter 7) in approximately 3000 lines of Go and 1000 lines of C++. To evaluate its performance, we benchmarked a deployment serving users who follow 16–24 people according to the Twitter popularity distribution Chapter 8 and compared it to the best-performing batch-PIR baseline from **??**.

Experimental Setup. We use `c7i.2xlarge` AWS instances (8 vCPUs, 16 GB RAM) for CPU-based machines, and `p3.2xlarge` AWS instances (NVIDIA V100 w/ 16 GB RAM) for GPU-based machines. Using current cost estimates from AWS, each instance costs \$0.36/hour and \$3.06/hour respectively. To simulate clients we use a `c7i.2xlarge` instance in a separate AWS region, with a 12ms RTT between machines in either region. For estimating costs, we assume that any job we run fully utilizes the machine for its runtime.

Parameters. We parameterize our linearly homomorphic encryption scheme and hint-compression to satisfy 128-bits of computational security and 40-bits of statistical correctness. We use the same distributional PIR correctness values from earlier in the evaluation, so CrowdSurf recovers $\sim 80\%$ of queries on average. For the batch-PIR scheme we use the Hash-based batch code which recovers $\sim 90\%$ of tweets on average.

We set the tweet size to a loose upper-bound of 560 bytes [60]. Constraining the Twitter popularity distribution to users who follow 16–24 users results in a distribution with a support 73 million unique users and a corresponding 38 GB database. We set the popular bucket of the database to be 15 MB. For a user making 24 queries, 16 of their queries fall into the popular bucket on average.

Per-request costs. In our deployment, hint-compression is the dominant cost due its use of RingLWE-based encryption. To deal with this, we squish the database in each chunk to reduce the hint size; this has the additional effect of reducing the download size for the client but increasing server storage. To balance out costs, we parallelize hint-compression across multiple CPUs while batching multiple concurrent PIR requests on a single GPU. For our baseline, we do the same thing for hint-compression but batch PIR requests across CPUs rather than GPUs.

In Table 6 we give the concrete costs associated with a single client making a request. CrowdSurf populates a client's feed in half a second using 21 MB of communication and for a cost of 0.0057 cents. Note that, while it takes 500ms for all queries to be processed, on average, 16 of the 24 tweets a client requests are in the first database chunk and arrive in 50ms of latency. Thus, a client wouldn't have any observed latency using CrowdSurf compared to non-private Twitter. Compared to the baseline, CrowdSurf uses 20% less server storage, reduces the cost of hint compression by 6.4×, reduces the cost of PIR by 40×, and reduces the total cost by 8×.

Note that the gains from our techniques our heavily outweighed by the cost of hint-compression. As techniques for performing hint-compression continue to improve, these will immediately increase the relative improvement of CrowdSurf compared to the baseline.

Chapter 9

Conclusion

In this thesis, we introduced a new cryptographic primitive, distributional PIR, and demonstrated its ability to reduce computational cost compared to standard PIR. There are a number of exciting directions for future work to explore:

1. Our constructions only need to know a few bits of information about the popularity distribution. A natural question to ask is whether utilizing more fine-grained information about the distribution can improve performance.
2. Our constructions make black-box use of a standard batch-PIR scheme. It would be interesting to explore the effect of using a distributional-PIR scheme to instantiate the construction. For example, it might be possible that taking the popularity distribution into account could produce more efficient batch codes, improving the overall performance of our schemes.
3. At a high-level, distributional PIR utilizes auxillary-information about the inputs to the protocol to increase performance. Can a similar principle be applied to a broader class of cryptographic problems?

References

[1] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," in *FOCS*, 1995.

[2] E. Kushilevitz and R. Ostrovsky, "Replication is not needed: Single database, computationally-private information retrieval," in *FOCS*, 1997.

[3] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian, "XPIR: Private information retrieval for everyone," *PoPETs*, 2016.

[4] S. Angel, H. Chen, K. Laine, and S. Setty, "PIR with Compressed Queries and Amortized Query Processing," in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2018, pp. 962–979. DOI: 10.1109/SP.2018.00062.

[5] M. H. Mughees, H. Chen, and L. Ren, *OnionPIR: Response Efficient Single-Server PIR*, 2021.

[6] S. J. Menon and D. J. Wu, "Spiral: Fast, high-rate single-server PIR via FHE composition," in *S&P*, 2022.

[7] A. Davidson, G. Pestana, and S. Celi, *Frodopir: Simple, scalable, single-server private information retrieval*, Cryptology ePrint Archive, Paper 2022/981, 2022.

[8] A. Henzinger, M. M. Hong, H. Corrigan-Gibbs, S. Meiklejohn, and V. Vaikuntanathan, "One server for the price of two: Simple and fast single-server private information retrieval," in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/henzinger.

[9] S. J. Menon and D. J. Wu, "YPIR: High-throughput single-server PIR with silent preprocessing," in *USENIX Security Symposium*, 2024.

[10] L. de Castro, K. Lewi, and E. Suh, *Whispir: Stateless private information retrieval with low communication*, Cryptology ePrint Archive, Paper 2024/266, https://eprint.iacr.org/2024/266, 2024. URL: https://eprint.iacr.org/2024/266.

[11] B. Li, D. Micciancio, M. Raykova, and M. Schultz-Wu, "Hintless single-server private information retrieval," in *CRYPTO*, 2024.

[12] A. Henzinger, E. Dauterman, H. Corrigan-Gibbs, and N. Zeldovich, "Private web search with Tiptoe," in *SOSP*, Koblenz, Germany, Oct. 2023.

[13] I. Ahmad, D. Agrawal, A. E. Abbadi, and T. Gupta, "Pantheon: Private retrieval from public key-value store," *Proceedings of the VLDB Endowment*, vol. 16, no. 4, pp. 643–656, 2022.

[14] T. Gupta, N. Crooks, W. Mulhern, S. Setty, L. Alvisi, and M. Walfish, "Scalable and private media consumption with Popcorn," in *NSDI*, 2016.

[15] I. Ahmad, L. Sarker, D. Agrawal, A. El Abbadi, and T. Gupta, "Coeus: A system for oblivious document ranking and retrieval," in *SOSP*, 2021, pp. 672–690.

[16] S. Angel and S. Setty, "Unobservable communication over fully untrusted infrastructure," in *OSDI*, 2016.

[17] I. Ahmad, Y. Yang, D. Agrawal, A. E. Abbadi, and T. Gupta, "Addra: Metadata-private voice communication over fully untrusted infrastructure," in *OSDI*, 2021.

[18] A. Beimel, Y. Ishai, and T. Malkin, "Reducing the servers' computation in private information retrieval: PIR with preprocessing," *J. Cryptol.*, 2004.

[19] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," ser. SOSP '95.

[20] S. Li, J. Xu, M. van der Schaar, and W. Li, "Popularity-driven content caching," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*.

[21] M. Gabielkov, A. Rao, and A. Legout, "Studying Social Networks at Scale: Macroscopic Anatomy of the Twitter Social Graph," in *ACM Sigmetrics 2014*, 2014.

[22] Google, *Google Trends*, 2024. URL: https://trends.google.com/.

[23] X. Cheng, C. Dale, and J. Liu, "Statistics and social network of youtube videos," in *2008 16th Interntional Workshop on Quality of Service*.

[24] Z. Brakerski and V. Vaikuntanathan, "Efficient Fully Homomorphic Encryption from (Standard) LWE," in *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, IEEE, 2011, pp. 97–106. DOI: 10.1109/FOCS.2011.12.

[25] A. Clauset, C. R. Shalizi, and M. E. J. Newman, "Power-law distributions in empirical data," *SIAM Review*, vol. 51, no. 4, pp. 661–703, 2009.

[26] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai, "Batch codes and their applications," in *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, ACM, 2004, pp. 262–271. DOI: 10.1145/1007352.1007396.

[27] M. H. Mughees and L. Ren, "Vectorized Batch Private Information Retrieval," in *2023 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2023, pp. 437–452. DOI: 10.1109/SP46215.2023.10179329.

[28] A. Burton, S. J. Menon, and D. J. Wu, *Respire: High-rate PIR for databases with small records*, Cryptology ePrint Archive, Paper 2024/1165, 2024. URL: https://eprint.iacr.org/2024/1165.

[29] J. Liu, J. Li, D. Wu, and K. Ren, "Pirana: Faster multi-query pir via constant-weight codes," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024. URL: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00039.

[30] A. Fazeli, A. Vardy, and E. Yaakobi, "Codes for distributed pir with low storage overhead," in *2015 IEEE International Symposium on Information Theory (ISIT)*.

[31] M. Lam, J. Johnson, W. Xiong, *et al.*, *GPU-based Private Information Retrieval for On-Device Machine Learning Inference*, 2023. arXiv: 2301.10904 `[cs]`.

[32] S. Vithana, K. Banawan, and S. Ulukus, "Semantic private information retrieval: Effects of heterogeneous message sizes and popularities," in *GLOBECOM 2020*.

[33] A. Gomez-Leos and A. Heidarzadeh, "Single-server private information retrieval with side information under arbitrary popularity profiles," in *2022 IEEE Information Theory Workshop (ITW)*.

[34] W.-K. Lin, E. Mook, and D. Wichs, "Doubly efficient private information retrieval and fully homomorphic ram computation from ring lwe," in *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, 2023, pp. 595–608.

[35] H. Corrigan-Gibbs and D. Kogan, "Private information retrieval with sublinear online time," in *EUROCRYPT*, 2020.

[36] H. Corrigan-Gibbs, A. Henzinger, and D. Kogan, "Single-server private information retrieval with sublinear amortized time," in *EUROCRYPT*, 2022.

[37] D. Kogan and H. Corrigan-Gibbs, "Private blocklist lookups with checklist," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 875–892.

[38] M. Zhou, A. Park, E. Shi, and W. Zheng, "Piano: Extremely simple, single-server pir with sublinear server computation," 2024.

[39] A. Lazzaretti and C. Papamanthou, "Treepir: Sublinear-time and polylog-bandwidth private information retrieval from ddh," in *Annual International Cryptology Conference*, Springer, 2023, pp. 284–314.

[40] S. Halevi and V. Shoup, *Bootstrapping for HElib*, 2014.

[41] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, *GAZELLE: A Low Latency Framework for Secure Neural Network Inference*, 2018.

[42] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, *Delphi: A Cryptographic Inference Service for Neural Networks*, 2020.

[43] Z. Huang, W.-j. Lu, C. Hong, and J. Ding, "Cheetah: Lean and Fast Secure {Two-Party} Deep Neural Network Inference," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 809–826.

[44] M. Hao, H. Li, H. Chen, P. Xing, G. Xu, and T. Zhang, "Iron: Private Inference on Transformers," in *Advances in Neural Information Processing Systems*, 2022.

[45] J. Zhang, J. Liu, X. Yang, Y. Wang, K. Chen, X. Hou, K. Ren, and X. Yang, *Secure Transformer Inference Made Non-interactive*, 2024.

[46] H. Roh, J. Yeo, Y. Ko, G.-Y. Wei, D. Brooks, and W.-S. Choi, *Flash: A Hybrid Private Inference Protocol for Deep CNNs with High Accuracy and Low Latency on CPU*, https://arxiv.org/abs/2401.16732v1, 2024.

[47] S. Angel and S. Setty, "Unobservable Communication over Fully Untrusted Infrastructure," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 551–569.

[48] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai, "Batch codes and their applications," in *STOC*, 2004.

[49] D. Boneh and V. Shoup, *A Graduate Course in Applied Cryptography*. 2024. URL: https://crypto.stanford.edu/~dabo/cryptobook/BonehShoup_0_6.pdf.

[50] R. Renner, "On the variational distance of independently repeated experiments," eprint archive: http://arxiv.org/abs/cs.IT/0509013, Manuscript, Nov. 2003.

[51] S. Meiklejohn, J. DeBlasio, D. O'Brien, C. Thompson, K. Yeo, and E. Stark, "SoK: SCT auditing in Certificate Transparency," in *PETS*, 2022.

[52] H. Corrigan-Gibbs and D. Boneh, "Prio: Private, robust, and scalable computation of aggregate statistics," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*.

[53] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of the ACM*, vol. 56, no. 6, 34:1–34:40, 2009. DOI: 10.1145/1568318.1568324.

[54] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "Fully Homomorphic Encryption without Bootstrapping," 2011.

[55] L. de Castro, C. Juvekar, and V. Vaikuntanathan, *Fast Vector Oblivious Linear Evaluation from Ring Learning with Errors*, 2020.

[56] C. Peikert and Z. Pepin, "Algebraically structured lwe, revisited," in 2019.

[57] *Microsoft SEAL (release 4.1)*, Microsoft, 2024.

[58] S. J. Menon and D. J. Wu, *YPIR: High-Throughput Single-Server PIR with Silent Preprocessing*, 2024.

[59] B. Li, D. Micciancio, M. Raykova, and M. Schultz-Wu, *Hintless Single-Server Private Information Retrieval*, 2023.

[60] X. Corp., *Counting characters*, 2024. URL: https://developer.x.com/en/docs/counting-characters.

Appendix A

Additional material from Chapter 3

A.1 Batch PIR: Security

Since security of a batch-PIR scheme can either be informational-theoretic or computational, we handle both cases by bounding the advantage of an adversary $\mathcal{A}$ by some value δ. In the computational setting, δ is negligible in some security parameter λ, and the runtime of $\mathcal{A}$, $|D|, \log M$, and B are all polynomial in λ. In the information-theoretic setting, the runtime of the $\mathcal{A}$, $\delta, |D|, \log M$, and B can be any constants.

We define security using Experiment A.1.1. Let $\mathsf{Sec}_{\Pi,N}(\cdot,\cdot)$ denote the output of the experiment, then for some database size N and adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$, define their *advantage* with respect to Π as:

$$\mathsf{BatchAdv}[\mathcal{A}, \Pi, N] = |\Pr[\mathsf{Sec}_{\Pi,N}(\mathcal{A}, 0) = 1] - \Pr[\mathsf{Sec}_{\Pi,N}(\mathcal{A}, 1) = 1]| .$$

We say that a batch-PIR scheme Π is δ-secure iff for all database sizes N and adversaries $\mathcal{A}$:

$$\mathsf{BatchAdv}[\mathcal{A}, \Pi, N] \leq \delta.$$

Experiment A.1.1 (Batch PIR: Security experiment)**.** The experiment is parameterized by (1) a batch-PIR scheme $\Pi = (\mathsf{Setup}, \mathsf{Encode}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Recover})$ with message space $\mathcal{M}$ and batch size B, (2) a database size N (3) an adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$, (4) a bit $b \in \{0, 1\}$. We compute the output of the experiment as:

$\underline{\mathsf{Sec}_{\Pi,N}(\mathcal{A}, b) :}$

$$\begin{aligned} (\mathsf{st}, I_0, I_1) &\leftarrow \mathcal{A}_0() \\ \mathsf{pp} &\leftarrow \mathsf{Setup}(1^N) \\ (\_, q) &\leftarrow \mathsf{Query}(\mathsf{pp}, I_b) \\ \text{Output } b' &\leftarrow \mathcal{A}_1(\mathsf{st}, \mathsf{pp}, q) \end{aligned}$$