Distributional Private Information Retrieval

Ryan Lehmkuhl
MIT

Alexandra Henzinger
MIT

Henry Corrigan-Gibbs
MIT

Abstract

A private-information-retrieval (PIR) scheme lets a client fetch a record from a remote database without revealing which record it 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 PIR, a new type of PIR that can run faster than classic PIR—both asymptotically and concretely—when the popularity distribution is skewed. Distributional PIR provides exactly the same cryptographic privacy as classic PIR. The speedup comes from a relaxed form of correctness: distributional PIR guarantees that in-distribution queries succeed with good probability, while out-of-distribution queries succeed with lower probability. Because of its relaxed correctness, distributional PIR is best suited for applications where "best-effort" retrieval is acceptable. Moreover, for security, a client's decision to query the server must be independent of whether its past queries were successful.

We construct a distributional-PIR scheme that makes black-box use of classic PIR protocols, and prove a lower bound on the server runtime of a natural class of distributional-PIR schemes. On two real-world popularity distributions, our construction reduces compute costs by 5-77× compared to existing techniques. Finally, we build CrowdSurf, an end-to-end system for privately fetching tweets, and show that distributional-PIR reduces the end-to-end server cost by 8×.

1 Introduction

Today, the tweets we read on Twitter, the websites we visit, and the videos we watch on TikTok all reveal sensitive information about us—letting each service learn our interests and activities. Cryptographic protocols for private information retrieval (PIR) [14, 47] protect this query data: PIR lets a user fetch a record from a remote database without revealing to the server hosting the database which record the user is fetching. The performance of PIR has improved dramatically over the last decade [1, 2, 6, 21, 23, 40, 41, 55, 64, 65, 69], demonstrating the potential viability of private web search [40], private media delivery [3, 35], and metadata-hiding messaging [4, 7].

Unfortunately, a longstanding barrier to the deployment of PIR has been its server-side computational overhead. An information-theoretic lower bound [9] shows that, if the server learns nothing about which database record a user is fetching, then the server 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 must run in $\Omega(N)$ time. (Schemes that use preprocessing [9, 18, 19, 46, 57] can theoretically circumvent the $\Omega(N)$-time lower bound. To date, these schemes only provide meaningful speedups as either the database, server storage, or client storage becomes extremely large.) In contrast, a non-private lookup on a RAM machine requires just $O(1)$ operations. Additionally, non-private database systems further optimize lookups by caching and prefetching popular elements, leveraging information about which records a user queried previously and which records are being queried by other users [56, 72].

In this work, we design analogous optimizations for private database lookups. In particular, we introduce a new approach to bypass 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 [30], the top 1% of web domains account for more than 95% of all web-browsing activity [54, 76], and the top 1% of accounts on TikTok have orders of magnitude more followers than other users on the platform [78]. We show that, without learning anything about the record that a particular user is fetching, a server can take advantage of this skewed access pattern to answer user queries much faster than a standard PIR server.

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

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

In practice, distributional-PIR schemes produce the largest speedups when failures occur with some constant probability, e.g., clients recover 80% of queries on average. As a result, distributional-PIR schemes are best suited for applications where either (a) clients can tolerate occasional failures, or (b) clients make queries with a regular frequency, so they can retry failed queries at a later time. A number of applications fit these constraints. For example, clients of social-media platforms such as Twitter or TikTok fetch large batches of content at once (e.g., from people or topics that they follow); in this setting, failing to retrieve a few posts does not significantly impact the user experience. Additionally, browsers regularly audit certificates received from websites by attempting to fetch the certificate from a set of validated certificates [50, 63]; in this setting, clients can re-attempt failed audits at a later time.

We construct a distributional-PIR scheme whose expected running time can be a small fraction of the database size whenever the popularity distribution is far from uniform. For certain classes of popularity distributions (e.g. power-law [15]), our scheme can achieve sublinear expected running time. Moreover, our construction makes black-box use of any classical PIR scheme. Our technique is to copy the "popular" database entries into a separate, small database (just as a traditional database server stores "hot" items in a cache). When generating a query, with some probability clients query the popular database instead of the original one. The more skewed the popularity distribution $\mathcal{P}$, the smaller the popular database is, allowing the server to run in much less time. Since the client's choice of which database to query is *independent* of the record it wants to fetch, we guarantee exactly the same notion of cryptographic privacy as classical PIR.

To give an example: if, on a database with N-entries, 90% of queries 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 full database, giving a per-query server time of $0.9 \cdot 0.1N + 0.1 \cdot N = 0.19 \cdot N$—that is, more than 5× faster than standard PIR.

To demonstrate that distributional PIR provides practical reductions in server cost, we evaluate our construction using a real-world popularity distribution built from a 2014 crawl of the Twitter social graph that spans 505 million accounts. Our microbenchmarks show that distributional PIR, when parameterized to return 19 of 24 queried tweets on average, reduces server work by 5–77× and communication by 2–117× compared to existing batch-PIR schemes [6, 7, 43, 70, 74]. Additionally, we use data from a 2022 large-scale web study [76] to evaluate distributional PIR on the problem of auditing signed certificate timestamps in certificate transparency, and demonstrate that it reduces server work by 12× and communication by 3× compared to existing techniques [65].

We provide evidence that achieving a substantially lower runtime than our construction likely requires fundamentally different techniques. In particular, we prove a lower bound on the server-runtime of any distributional PIR scheme that answers queries by probing elements of the original database (i.e., schemes without database encoding); on the real-world distributions described above, the runtime of our construction is within 1.4× of this lower-bound.

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 reduce the running time of client encryption in SimplePIR [41]—a standard PIR scheme that we use as a subroutine—by 128–349× and server preprocessing by 116–351×. We achieve these gains by replacing SimplePIR's encryption scheme based on the learning-with-errors problem to one based on the ring version of the assumption. We then show how to transform these ring ciphertext back into SimplePIR-style ciphertexts with a new application of an old technique ("modulus switching" [12]). In addition, we utilize fast matrix-multiplication kernels on modern GPUs to speed up the per-query server work in SimplePIR [41] by upwards of 3× compared to a cluster of CPUs (maintaining the same dollar-cost per query).

Finally, we design and implement CrowdSurf, a system that enables a Twitter client to stream a feed of tweets from a server without revealing which accounts or tweets the client is interested in. Our CrowdSurf server runs a distributional-PIR scheme (parameterized by the 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 38 GB database with 500ms of latency, and 21 MB of traffic—costing the server 0.0057 cents in total, 8× less than existing techniques.

In summary, this paper contributes:

- the notion of distributional PIR, a new variant of PIR in which the server can run much faster than in classical PIR,
- a generic compiler that lifts a standard PIR scheme into a distributional-PIR scheme,
- a lower bound on the server-runtime of a certain class of distributional-PIR schemes, and
- an implementation and evaluation of our new scheme in CrowdSurf, a system that enables a Twitter client to receive tweets from other users without revealing which tweets they are receiving.

Limitations. While distributional-PIR can reduce server runtime, this new type of PIR comes with two main drawbacks. First, the PIR server must have a good approximation of the popularity distribution $\mathcal{P}$. While some applications naturally reveal such a popularity distribution to the server (e.g., Twitter publishes follower counts), collecting these statistics in a privacy-preserving manner may be challenging in some settings. Moreover, no matter how it is approximated, the resulting distribution reveals the aggregate behavior of the measured users. 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., social-media feeds), it may not be suitable for other deployment scenarios. Furthermore, this weakened correctness notion more notably affects users with "out-of-distribution" query patterns, potentially raising fairness concerns.

2 Defining distributional PIR

In this section, we introduce distributional PIR, a new type of private-information-retrieval scheme for applications in which (1) some database entries are queried more often than others and (2) a relaxed correctness guarantee is acceptable. In this case, distributional PIR schemes can, on average, probe a small fraction of the database entries when answering queries.

A distributional PIR scheme on a database of N items takes as input a probability distribution $\mathcal{P}$ over the indices $\{1, \ldots, N\}$. Each client can request a batch of B items from the server at once, as in batch-PIR [8, 43, 70].

2.1 Definition

We now define distributional PIR. We explain the intuition here and defer the full formal definitions to Appendix A.1.

Syntax. The syntax for a distributional-PIR scheme is a strict generalization of the standard syntax for batch-PIR schemes [43], 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. (See Appendix B.1 for a full description of batch-PIR.)

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 to be fairly small while the encoded database D_{code} can contain much more information about the distribution $\mathcal{P}$.

Security. Informally, the client's query should leak no information about their requested database indices, just as in a standard PIR scheme. In particular, for all database indices i and j the server's view of an interaction with a client querying for i and j should be indistinguishable. We formally define this notion in Appendix A.1.1.

As with standard PIR, security only holds if the client's query pattern is independent of its query indices or the success/failure of past queries. As a result, if failure is unacceptable and the client doesn't query with a regular frequency, the scheme must be parameterized to ensure that failure occurs with negligible probability. However, in many applications of interest (e.g. Sections 8 and 9), "best-effort" retrieval is acceptable and the client can simply ignore failures.

Correctness. We define three correctness notions for distributional PIR, capturing three types of correctness failure: (1) the client obtains corrupted output and fails to detect it, (2) the client fails to obtain its desired database record when fetching an arbitrary database record, and (3) the client fails to obtain its desired database record when their query is sampled from the popularity distribution $\mathcal{P}$. We give formal definitions of each of these correctness notions in Appendix A.1.2.

Explicit correctness. We say that a distributional-PIR scheme has explicit correctness κ_{exp} if, no matter which database records the client wants to fetch, they recover either their desired record or a failure symbol with probability at least κ_{exp}, where the probability is over the randomness of the PIR algorithms. That is, with probability κ_{exp}, the client knows if the server failed to answer their query correctly.

Worst-case correctness. We say that a distributional-PIR scheme has worst-case correctness κ_{worst} if, no matter which database records the client wants to fetch, the client recovers each of its desired records with probability at least κ_{worst}, where the probability is over the randomness of the PIR algorithms.

Average-case correctness. We say that a distributional-PIR scheme has average-case correctness κ_{avg} on a probability distribution $\mathcal{P}$ and batch size B if, when the client queries

for a list of B indices sampled i.i.d. from the distribution $\mathcal{P}$, the client recovers a κ_{avg} fraction of its desired records, in expectation over the random draws from $\mathcal{P}$ and the randomness of the PIR algorithms.

While modeling clients as sampling their indices i.i.d. from the distribution $\mathcal{P}$ is a simplification of real-world behavior (in practice, queries may be correlated), in Section 7.2 we demonstrate that it accurately approximates clients' behavior on a real-world dataset [30].

Relationship between correctness notions. We always have that $\kappa_{\mathsf{worst}} \leq \kappa_{\mathsf{avg}} \leq \kappa_{\mathsf{exp}}$ since an explicit correctness failure is also an average-case correctness failure, and an average-case correctness failure is also a worst-case correctness failure.

Efficiency. We consider two main cost metrics for distributional-PIR, which we define formally in Appendix A.1.3. Informally, a distributional-PIR scheme has:

- *expected server time T* if the Answer routine makes at most T probes to the database D_{code} in expectation, and
- *expected communication cost C* if the total size of the public parameters pp, a query q, and an answer a is less than C in expectation.

PIR schemes are distributional-PIR schemes. We can interpret any standard PIR scheme with correctness κ as a distributional-PIR scheme in which all three correctness parameters are κ. The power of distributional-PIR schemes over standard PIR schemes is that distributional schemes can—depending on the popularity distribution—have the same average-case correctness with significantly reduced server-side cost compared to a standard PIR scheme.

2.2 Robustness against distribution shift

In practice, the popularity distribution $\mathcal{P}$ used by the distributional-PIR server may be 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 the 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 correctness of a distributional-PIR scheme under the true query distribution $\hat{\mathcal{P}}$ is "close" to that under the shifted query distribution $\mathcal{P}$.

We formalize and prove this statement in the full version of the paper [53]. The idea is that if the distributions $\mathcal{P}$ and $\hat{\mathcal{P}}$ are statistically close, then the output of the distributional-PIR scheme on either distribution must also be close.

3 Constructing distributional PIR

In this section, we construct a conceptually simple distributional-PIR scheme: first, we construct a distributional-PIR scheme that is fast but fails often, then we combine it with a standard batch-PIR scheme to boost it to a scheme with both worst- and average-case correctness.

Background: Batch PIR. Our distributional-PIR construction makes black-box use of an arbitrary standard batch-PIR scheme [43], which we define formally in Appendix B.1. A batch-PIR scheme allows a client to privately fetch a batch of database records from a server, at roughly the same server-side cost as fetching a single database record. The standard construction of a batch-PIR scheme combines a batch code [44] with a standard single-query PIR scheme.

From a definitional point of view, a batch-PIR scheme is just a distributional-PIR scheme in which the Setup algorithm does not take a popularity distribution as input and that has no notion of average-case or explicit correctness. The efficiency of our constructions depend on the efficiency of the underlying batch-PIR scheme.

Notation. We denote $[m]$ as the set of integers $\{0, 1, \cdots, m-1\}$. We use $\widetilde{O}(\cdot)$ to hide poly-logarithmic factors in the input. 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 define a probability distribution $\mathcal{P}$ over $[N]$ as a list of N real numbers $(p_1, p_2, \ldots p_N)$. We slightly abuse notation and use $\mathsf{cdf}_{\mathcal{P}}(\cdot)$ to denote the cumulative distribution function of a distribution $\mathcal{P}$ *when sorted by popularity*, i.e., $\mathsf{cdf}_{\mathcal{P}}(1)$ returns $\max_{i \in [N]} p_i$. To be able to make asymptotic statements about the efficiency of our PIR schemes (e.g, to use big-O notation), formally we must work with *families* of probability distributions $\mathcal{P} = \{\mathcal{P}_N\}_{N=1}^{\infty}$, with one distribution defined for each database size N. We elide this formalism and just talk about one distribution $\mathcal{P}$.

3.1 A basic distributional-PIR scheme without worst-case correctness

We start by constructing a distributional PIR scheme that can achieve any value of average-case correctness (i.e., $\kappa_{\mathsf{avg}} \in [0, 1]$), but has no worst-case guarantees (i.e., $\kappa_{\mathsf{worst}} = 0$). The scheme first finds the smallest value $k \in \mathbb{N}$ such that the k most popular database records, under distribution $\mathcal{P}$, account for at least κ_{avg} of $\mathcal{P}$'s probability mass. Then, it runs standard batch-PIR over those k most-popular elements, ignoring all other elements of the database. We give the full details in Construction 3.1.

To allow clients to build queries, the public parameters of Construction 3.1 contain the database indices of the k most-popular entries. In practice these might be large, so in Section 4.2 we discuss several ways to minimize their size.

Lemma 3.2. *For all constants $\kappa_{\mathsf{avg}} \in [0, 1]$, message space $\mathcal{M}$, batch size B, and δ-secure, errorless batch-PIR scheme Π_{batch} such that:*

- *Π_{batch} is parameterized by a message space $\mathcal{M}$, 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 3.1 (A distributional-PIR scheme without worst-case correctness)**.** The construction is parameterized by a constant $\kappa_{\mathsf{avg}} \in [0,1]$ and a batch-PIR scheme (Appendix B.1) $(\mathsf{Setup}, \mathsf{Encode}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Recover})$ for message space $\mathcal{M}$ and batch size B.

$\underline{\mathsf{Dist.Setup}(\mathcal{P}) \to \mathsf{pp}.}$

- Compute $k \leftarrow \mathsf{cdf}_{\mathcal{P}}^{-1}(\kappa_{\mathsf{avg}})$
- Let $\mathcal{L} = (\ell_1, \ell_2, \ldots, \ell_k)$ be the k most-popular indices
- Run $\mathsf{pp}_{\mathsf{batch}} \leftarrow \mathsf{Setup}(1^k)$
- Output $(\mathsf{pp}_{\mathsf{batch}}, \mathcal{L})$

$\underline{\mathsf{Dist.Encode}(\mathsf{pp}, \mathcal{P}, D) \to D_{\mathsf{code}}.}$

- Parse $\mathsf{pp} \to (\_, \mathcal{L} = (\ell_1, \ell_2, \ldots, \ell_k))$, $D \to (d_1, d_2, \ldots, d_N)$
- Run $D_{\mathsf{code}} \leftarrow \mathsf{Encode}((d_{\ell_1}, d_{\ell_2}, \ldots, d_{\ell_k}))$
- Output D_{code}

$\underline{\mathsf{Dist.Query}(\mathsf{pp}, I) \to (\mathsf{st}, q).}$

- Parse $\mathsf{pp} \to (\mathsf{pp}_{\mathsf{batch}}, \mathcal{L} = (\ell_1, \ell_2, \ldots, \ell_k))$
- For all $j \in [B]$, compute $\mathcal{E} \leftarrow \{j : I_j \notin \mathcal{L}\}$
- Initialize a list $Q = (1)^B$
- For all $j \in [B]$ and $b \in [k]$, if $I_j = \ell_b$, set $Q_j = b$
- Compute $(\mathsf{st}, q) \leftarrow \mathsf{Query}(\mathsf{pp}_{\mathsf{batch}}, Q)$
- Output $((\mathsf{st}, \mathcal{E}), q)$

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

- Output $\mathsf{Answer}^{D_{\mathsf{code}}}(q)$

$\underline{\mathsf{Dist.Recover}(\mathsf{st}, a) \to (m_1, \ldots, m_B).}$

- 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 $m_j = \perp$
- Output $(m_1, \ldots, m_B)$

Construction 3.1 is a δ-secure distributional PIR scheme that, on any database $D \in M^*$ and probability distribution $\mathcal{P}$, has

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

We prove Lemma 3.2 in the full version of the paper [53].

One nice feature of Construction 3.1 is that, if $\mathcal{P}$ follows a power-law distribution—which is the case for many real-world datasets [15]—the per-query server runtime converges to constant as the database size N grows large. We formalize and prove this statement in the full version of the paper [53]. The basic idea is that, if the database follows a power-law distribution, then, for all $\kappa_{\mathsf{avg}} \in [0,1)$, $\mathsf{cdf}_{\mathcal{P}}^{-1}(\kappa_{\mathsf{avg}})$ is a constant, and therefore the runtime of Construction 3.1 is also constant.

3.2 Main construction

We now present our main distributional-PIR construction Π. The idea is to combine a standard linear-time errorless PIR scheme with worst-case correctness $\kappa_{\mathsf{worst}} = 1$ with the PIR scheme of Section 3.1, which has $\kappa_{\mathsf{worst}} = 0$. In particular,

- with probability κ_{worst}, the client and server run an errorless batch-PIR over all N database records, and
- with probability $1 - \kappa_{\mathsf{worst}}$, the client and server run the fast-but-errorful PIR scheme of Section 3.1 over the $\mathsf{cdf}_{\mathcal{P}}^{-1}\left(\frac{\kappa_{\mathsf{avg}} - \kappa_{\mathsf{worst}}}{1 - \kappa_{\mathsf{worst}}}\right)$ most popular database records.

This combination gives a PIR scheme whose parameters are a linear combination of the two underlying schemes: effectively trading off the performance of the distributional-PIR scheme for the worst-case correctness of the batch-PIR scheme. The security and efficiency properties follow from those of the underlying PIR schemes. We obtain the following theorem:

Theorem 3.3. *For all constants $\kappa_{\mathsf{avg}}, \kappa_{\mathsf{worst}} \in [0,1]$, database sizes N, batch sizes B, probability distributions $\mathcal{P}$, and δ-secure, errorless batch-PIR schemes Π_{batch} such that:*

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

Let $k = \mathsf{cdf}_{\mathcal{P}}^{-1}\left(\frac{\kappa_{\mathsf{avg}} - \kappa_{\mathsf{worst}}}{1 - \kappa_{\mathsf{worst}}}\right)$. There exists a 2δ-secure distributional-PIR scheme Π with:

- *explicit correctness 1,*
- *average-case correctness κ_{avg},*
- *worst-case correctness κ_{worst},*
- *expected server runtime*

$$\widetilde{O}\left(k \cdot (1 - \kappa_{\mathsf{worst}}) + N \cdot \kappa_{\mathsf{worst}}\right), \text{ and}$$

- *expected communication*

$$k \log N + C(k) \cdot (1 - \kappa_{\mathsf{worst}}) + C(N) \cdot \kappa_{\mathsf{worst}}.$$

In Construction B.1, we give a precise description of the PIR scheme that satisfies the theorem. We prove Theorem 3.3 in the full version of the paper [53].

Remark 3.4 (Instantiating our construction)**.** Our construction instantiates two PIR schemes: one over a database containing the most-popular entries and another over the full database. When clients make a query, one of these PIR instances is randomly selected–independently of the client's requested index–and run. Our presentation thus far has used the same PIR scheme to instantiate both instances of PIR, however, in some cases it is advantageous to instantiate our construction with two different PIR schemes exhibiting asymmetric properties. In the full version of the paper [53], we show that this composition remains secure provided that both underlying PIR schemes are secure. For example, a distributional-PIR server can trade-off computation with client-server communication

by instantiating the PIR scheme for the popular database with the naive PIR scheme, i.e., where the client simply downloads the entire database.

4 Deploying distributional PIR

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

4.1 Measuring the popularity distribution

A distributional 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 multiple ways for the server to measure the distribution, even when it does not see client queries.

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 [63], 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 externally obtains 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 [17, 27, 73] to learn the popularity distribution $\mathcal{P}$ over a set of measured clients. One drawback of this process is that it might leak information about the queries of measured clients depending on the skew of the distribution: a uniform distribution leaks nothing, while a skewed distribution leaks which entries clients queried more and less frequently. To mitigate this, some distributional PIR schemes, such as our construction from Section 3, only require partial information about the distribution $\mathcal{P}$. Moreover, many private aggregation schemes support techniques for reducing leakage (e.g. by adding structured noise) [17, 27, 73]. Additionally, the skew of the distribution could be measured using multi-party computation [20, 83] before deciding to release it or not.

4.2 How the client learns the distribution

A distributional-PIR scheme has public parameters that the server sends to the client before the client makes any queries. These public parameters must succinctly encode all the information about the popularity distribution $\mathcal{P}$ that the client needs to make a PIR query.

Here, we discuss some ways to craft the public parameters to minimize their size. Table 1 compares the total costs of our construction using these different setups.

Database is sorted by popularity. In the best case scenario, the database is sorted by popularity *before* preprocessing. Then, the public parameters only need to specify the cutoff point $k \in [N]$ for the popular portion of the database.

Recursive PIR. If the entries of the database D are $\ell \gg \log N$ bits long, then the client can use standard batch-PIR [44] to fetch the position of their desired indices for much cheaper than a standard batch-PIR query to the entire database. In other words, the server encodes the permuted database indices $\mathcal{L} = (\ell_1, \ell_2, \cdots, \ell_k)$ from Construction 3.1 as another database that a client queries using PIR before building their distributional-PIR query.

Moreover, in settings such as our Twitter application (Section 9), the popularity distribution $\mathcal{P}$ changes slowly and the client repeatedly fetches the ith entry of a constantly-updating database–the most recent tweet from user i. In this case, the client can make one PIR query to the permuted database indices and reuse this information over many future queries. By doing this, the server additionally learns that the client is repeatedly fetching the same database entry, and will know if the client ever decides to query a new database index, but the overhead of recursive PIR is reduced.

5 Lower bounds for distributional PIR

In this section, we prove a lower bound on the expected runtime of certain single-query distributional-PIR schemes. Since a many-query distributional-PIR scheme implies a single-query distributional-PIR scheme, our lower bound also gives a lower bound on the running time of distributional-PIR schemes with large batch sizes.

No database encoding. We say that a distributional-PIR scheme Π uses *no database encoding* if the encoded database that $\Pi.\mathsf{Encode}$ outputs is just the original database that it takes as input (possibly with replicated elements). Our lower bound only applies to PIR schemes using no database encoding.

Pre-processing PIR schemes that use sophisticated database-encoding schemes [9, 57] can subvert our lower bound, though these schemes are, as of now, very far from practical [71]. At the same time, when instantiated with an underlying PIR scheme that does not use a database encoding [6, 10, 14, 19], our distributional-PIR construction does not use a database encoding either. (As a reminder, our syntax also enforces that the public parameters can't depend on the database contents.)

Theorem 5.1. *Given a database size $N \in \mathbb{N}$, message space $\mathcal{M}$, and popularity distribution $\mathcal{P}$ over $[N]$, let Π be a δ-secure distributional-PIR scheme that uses no database encoding and has:*

- *explicit correctness κ_{exp},*

Setups	*Expected Communication* Params	Per-Query	*Expected Server Runtime* Per-Query
Construction 3.2	$\log N \cdot \mathsf{cdf}_{\mathcal{P}}^{-1}(\frac{\kappa_{\mathsf{avg}}-\kappa_{\mathsf{worst}}}{1-\kappa_{\mathsf{worst}}})$	Q	R
Download Top-k	$(\log N+\ell)\cdot \mathsf{cdf}_{\mathcal{P}}^{-1}(\frac{\kappa_{\mathsf{avg}}-\kappa_{\mathsf{worst}}}{1-\kappa_{\mathsf{worst}}})$	$Q-(1-\kappa_{\mathsf{worst}})\cdot C(\mathsf{cdf}_{\mathcal{P}}^{-1}(\frac{\kappa_{\mathsf{avg}}-\kappa_{\mathsf{worst}}}{1-\kappa_{\mathsf{worst}}}),\ell)$	$R-(1-\kappa_{\mathsf{worst}})\cdot\ell\cdot \mathsf{cdf}_{\mathcal{P}}^{-1}(\frac{\kappa_{\mathsf{avg}}-\kappa_{\mathsf{worst}}}{1-\kappa_{\mathsf{worst}}})$
Sorted DB	$\log N$	Q	R
Recursive PIR	$\log N$	$Q+C(N,\log N)$	$R+N\log N$

Table 1: Big-O asymptotic costs associated with our construction on a message space $\mathcal{M}$, database size N, popularity distribution $\mathcal{P}$, batch size $B=1$, and constants $\kappa_{\mathsf{avg}},\kappa_{\mathsf{worst}}\in[0,1]$ when using different setups. "Download Top-k" refers to our construction with the optimization described in Remark 3.4; "Sorted" and "Recursive PIR" refer to different ways of setting the public parameters as described in Section 4.2. 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 $Q=\kappa_{\mathsf{worst}}\cdot C(N,\ell)+(1-\kappa_{\mathsf{worst}})\cdot C(\mathsf{cdf}_{\mathcal{P}}^{-1}(\frac{\kappa_{\mathsf{avg}}-\kappa_{\mathsf{worst}}}{1-\kappa_{\mathsf{worst}}}),\ell)$ and $R=\kappa_{\mathsf{worst}}\cdot N\cdot\ell+(1-\kappa_{\mathsf{worst}})\cdot \mathsf{cdf}_{\mathcal{P}}^{-1}(\frac{\kappa_{\mathsf{avg}}-\kappa_{\mathsf{worst}}}{1-\kappa_{\mathsf{worst}}})\cdot\ell$.

– *worst-case correctness* κ_{worst},
– *average-case correctness* κ_{avg}, *and*
– *expected server time* T.

Let $W=\delta+\frac{1-\kappa_{\mathsf{exp}}}{|\mathcal{M}|-1}$. *Then it must hold that:*

$$\mathbb{E}[T]\geq\max\left\{N\cdot(\kappa_{\mathsf{worst}}-W),\ \mathsf{cdf}_{\mathcal{P}}^{-1}(\kappa_{\mathsf{avg}}-W)\right\}.$$

Our lower bound in Theorem 5.1 shows that, the more skewed the probability distribution $\mathcal{P}$, the lower the expected server time of a distributional-PIR scheme. At the same time, the expected runtime of a distributional-PIR scheme must always be at least $\approx N\cdot\kappa_{\mathsf{worst}}$ meaning that, the lower the worst-case correctness, the lower the scheme's runtime.

On the real-world distributions we use in our evaluation (Sections 7.2 and 8), the runtime of our distributional-PIR construction for a single query is within $\sim 1.4\times$ of the lower-bound. Thus, achieving a substantially lower runtime than our approach likely requires some form of database encoding.

We prove Theorem 5.1 in the full version of the paper [53]. Here, we give a sketch of the proof strategy.

Proof sketch. We prove lower-bounds for a distributional-PIR scheme Π with just worst-case correctness, and for Π with just average-case correctness. The theorem follows by taking a max of these two cases. Both lower-bounds proceed similarly:

– First, we compare two distributions: (1) a standard interaction using Π to fetch database record $i\in[N]$ (2) the same interaction except $\Pi.\mathsf{Answer}$ fails if it probes database index i. This gives an expression between Π's correctness and the probability that $\Pi.\mathsf{Answer}$ probes the i-th database record.
– Because of Π's security guarantee, queries leak no information about their target index. Thus, for any two indices $i,j\in[N]$ the probes that $\Pi.\mathsf{Answer}$ makes to the database must look the same (up to a factor of δ) whether the client is reading index i or index j.
– Finally, we upper-bound the probability that $\Pi.\mathsf{Answer}$ probes index $i\in[N]$ given that it runs in time T and receives a PIR query for some index $j\in[N]$ sampled independently of i.

Combining these three steps completes the proof. □

6 Reducing costs in SimplePIR

Our distributional PIR construction makes black-box use of a standard PIR scheme. In our system evaluation (Section 9), we use SimplePIR [41] because of its low server-side computational cost when answering PIR queries. In this section, we give several optimizations to SimplePIR: for a security parameter n, we reduce the word operations required for client encryption and server preprocessing by $O(n/\log n)\times$ while leaving the communication cost and the server's time to answer a query exactly the same.

In particular, we optimize SimplePIR's scheme for *linearly homomorphic encryption with preprocessing* [41], the core building block behind many of the fastest PIR schemes [21, 40, 41, 55]. SimplePIR is essentially a twist on Regev's lattice-based encryption scheme [74].

Using a linearly homomorphic encryption scheme, a client can encrypt a vector $\mathbf{x}$ and a server can multiply the encrypted vector by a matrix $\mathbf{D}$ under encryption. By preprocessing the matrix $\mathbf{D}$, the server can speed up the encrypted matrix-vector product. In the context of PIR, the client represents its query as a vector $\mathbf{x}$, that it encrypts and sends to the server. The server represents the database as a matrix $\mathbf{D}$, computes the matrix-vector product $\mathbf{D}\cdot\mathsf{Enc}(\mathbf{x})$ under encryption ("homomorphically"), and returns the result to the client.

Background: Computational costs of encryption schemes. Homomorphic operations in SimplePIR are incredibly cheap. For example, computing a homomorphic matrix-vector product with the SimplePIR encryption scheme on a square matrix of dimension $N\times N$, where each entry encrypts a 9-bit value, simply requires computing an N-by-N matrix-vector product modulo 2^{32}. In other words, the cost of homomorphic operations is the same as the underlying plaintext operations up to

a change in word size.

However, encryption and preprocessing with SimplePIR's encryption scheme are relatively expensive: encrypting a vector of 9-bit values of dimension N requires Nn word operations, where $n \approx 2^{11}$ is the size of the schemes secret key; similarly, preprocessing the database requires Nn^2 operations. These limitations appear in many schemes that, like SimplePIR, are based on the learning-with-errors (LWE) assumption.

In contrast, encryption schemes built on the ring learning-with-errors assumption (RingLWE) [11, 28, 60] have much faster encryption and preprocessing: requiring $N \log n$ word operations to encrypt a vector of size N, and $Nn \log n$ operations for preprocessing. In practice, these costs are $> 100\times$ less than those in SimplePIR. The downside of these schemes is that they require *all arithmetic to be carried out modulo a prime* (i.e., not modulo 2^{32} or 2^{64}) and consume more memory bandwidth, increasing the cost of homomorphic operations.

Our contribution. We construct a "best-of-both" encryption scheme that achieves fast encryption, preprocessing, and homomorphic evaluation. To accomplish this, our scheme performs preprocessing and encryption using a RingLWE-based encryption scheme, then converts the preprocessed state and ciphertext into forms that are compatible with SimplePIR; as a result, homomorphic operations are as cheap as in SimplePIR. Additionally, our scheme is compatible with techniques from prior work [40, 55, 65] that also mix LWE- and RingLWE-based encryption schemes to achieve fast decryption and a stateless client at the cost of slightly slower homomorphic evaluation. Table 2 compares our scheme to prior work.

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

1. The client encrypts its query vector $\mathbf{x}$ using a RingLWE-based encryption scheme with a prime modulus that is one-bit larger than 2^{32} or 2^{64}.
2. The client converts the RingLWE-type ciphertext to an LWE-type ciphertext (as used in Regev encryption and SimplePIR) and switches the modulus to 2^{32} or 2^{64}.

Step 2 of the above outline is the key new step. To implement it, we use modulus switching [11, 28], 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 reduce the error in a homomorphic computation [11, 22]. In contrast, we have the client modulus switch *before* homomorphic evaluation takes place. In more detail, the client encrypts their input under a RingLWE-based scheme using prime modulus q_1, reinterprets the RingLWE-type ciphertext as an LWE-type ciphertext (as in prior work [59, 65], see Appendix C for more details), and modulus switches the ciphertext to a new modulus q_2. By setting $q_2 \in \{2^{32}, 2^{64}\}$, the resulting scheme enjoys the fast encryption time of RingLWE-based schemes along with the fast homomorphic operations of the LWE-based schemes.

	Preproc.	Encrypt	Multiply	Decrypt	Stateless Client
LWE [41]	2973	0.7	0.4	0.7	✗
RingLWE [55]	3.3	0.008	2.2	0.247	✓
Hybrid [55, 65]	71	1	1.6*	0.004	✓
Section 6	16	0.004	0.4	0.7	✗
Section 6 [55, 65]	16	0.004	1.6*	0.004	✓

Table 2: Our use of modulus-switching produces the first linearly homomorphic encryption scheme that simultaneously achieves fast preprocessing, encryption, and evaluation. The table displays the performance of several encryption schemes when querying a 4 GB database. All numbers are in seconds. *The evaluation performance of this scheme approaches that of the LWE-based scheme as the database size grows.

(Preprocessing follows a similar template: computation is done using a RingLWE-based scheme over q_1 before modulus switching to q_2.)

The only downside of our scheme is noise growth during homomorphic operations. In particular, modulus switching before evaluation slightly *increases* the total noise in the ciphertext, restricting the size of the plaintext space. 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 computation when answering queries is so cheap that the bottleneck becomes the speed with which the processor can read the database from main memory.

Our idea to bypass this bottleneck is simple: offload the computation to a GPU, which has orders of magnitude more memory bandwidth than a CPU does. Our implementation of LWE homomorphic-evaluation makes black box use of existing, highly-optimized, matrix-multiplication libraries.

7 Evaluation

We evaluate the effectiveness of our techniques via several microbenchmarks. First, in Section 7.1 we demonstrate that our new encryption scheme (Section 6):

- reduces client and server costs in SimplePIR [41], and
- reduces the cost of deploying PIR by utilizing GPUs.

Then, we show that our distributional-PIR scheme (Section 3):

- improves PIR performance on real-world popularity distributions (Section 7.2),
- enables fast, private SCT auditing (Section 8), and
- reduces the server's cost in the context of serving Tweets to a pool of Twitter users without learning the users' interests (Section 9).

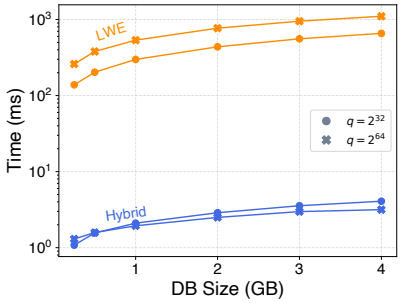

Figure 3: Client time to encrypt a vector for linear evaluation on a database of increasing size.

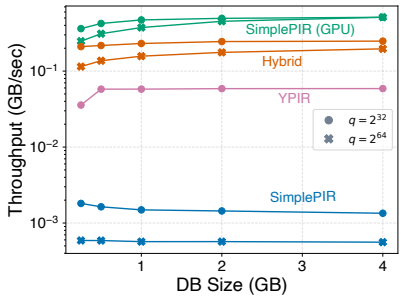

Figure 4: Server throughput when preprocessing a hint for a database of increasing size. The GPU instance is 8× more expensive to run than the other schemes. YPIR doesn't support 64-bit ciphertext moduli.

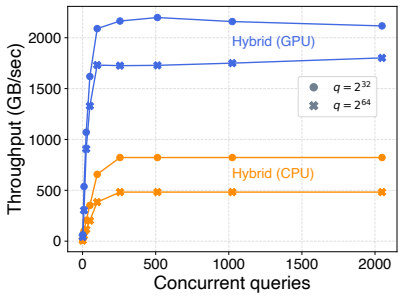

Figure 5: Server throughput when answering batches of client queries on a 4 GB database. The CPU throughput is a cluster of 8 machines each with eight cores. The deployment cost of the CPU-based cluster and singular GPU is the same.

Implementation. We implemented our distributional-PIR construction, PIR optimizations, and system for private Twitter feeds, CrowdSurf, in approximately 3000 lines of Go and 1000 lines of C++. CrowdSurf is open-source and available at `https://github.com/ryanleh/crowdsurf`.

7.1 Microbenchmarks: PIR optimizations

We evaluate the performance of our linearly homomorphic encryption with preprocessing scheme.

Throughout the microbenchmarks, we use ciphertext moduli $q = 2^{32}$ and $q = 2^{64}$ with respective lattice security parameters $n = 2048$ and $n = 4096$. These choices of moduli 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 [45, 68] or private web search [40], necessitate the larger plaintext space. We sample errors from a discrete Gaussian distribution with standard-deviation $\sigma = 3.2$.

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

Query Latency. In Figure 3 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 [67].

Preprocessing Throughput. In Figure 4, 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 [66]. 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 3.8–5.9× slowdown compared to our scheme.

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 [66].

The results of the experiment are shown in Figure 5. For a batch of 50 concurrent requests, one GPU can process roughly 3× more requests per second than the 64-core CPU cluster.

7.2 Microbenchmarks: Distributional PIR

Data set. We evaluate our distributional-PIR construction from Section 3 using real-world data from a 2014 crawl of the Twitter social graph [30] spanning 505 million accounts. We consider a database consisting of users' tweets and a distribution over the popularity of each user (Figure 6).

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

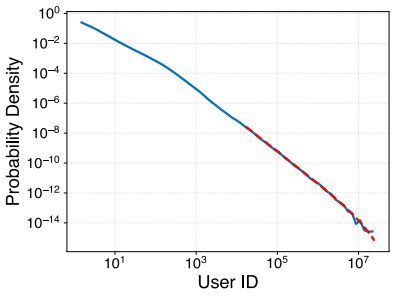

Figure 6: The distribution of Twitter followers per-user (shown in blue) follows a truncated power-law distribution [15] with $\alpha = 2.1$ (shown in red).

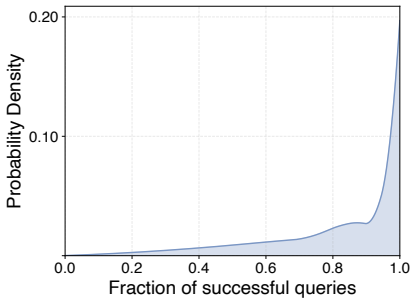

Figure 7: Our construction empirically provides the expected correctness for one million Twitter users making 24–32 queries. For $\kappa_{\text{avg}} = 0.8$ and $\kappa_{\text{worst}} = 0.01$, the resulting correctness distribution has a mean of 0.8 and a standard deviation of 0.22.

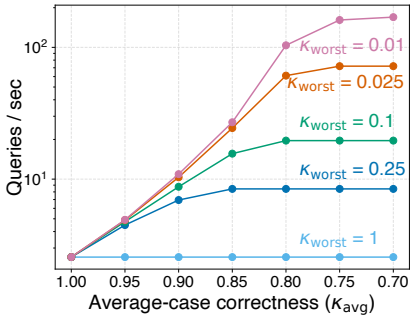

Figure 8: Decreasing correctness allows a distributional-PIR scheme to run much faster. The plot displays queries-per-second as average-case correctness varies when making 24 queries with our construction on a 4 GB database that follows the Twitter popularity distribution.

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. (Similar to prior work on batch-PIR [6, 43, 58, 70], we do not try to hide the number of accounts a user follows from the servers.) 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 8 we demonstrate how the correctness parameters effect performance. Throughout the evaluation we parameterize our construction with correctness values $\kappa_{\text{worst}} = 0.01, \kappa_{\text{avg}} = 0.8$. 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 every unique set of parameters we sampled a random subset of one million Twitter users and recorded the observed correctness for these users when using our construction. In all cases, we found that our scheme precisely achieved the desired accuracy. Figure 7 displays the result of one of these experiments.

7.2.1 Comparison to batch-codes

We compare the performance of distributional PIR against existing techniques for batch PIR from standard batch codes [6, 43]. 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. [43] that hashes database records into buckets ("Hash") and one from Angel et al. [6] 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.

We experiment with two different choices of PIR scheme for instantiating our distributional-PIR construction:

- SimplePIR [41]: this is the fastest single-server PIR scheme, but requires high communication and client state.
- Respire [13]: 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 200 MB (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 our construction with the best-performing batch-PIR scheme.

Since both our construction 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, 128 GB RAM). Such a large instance is necessary since the encoding outputted by the Cuckoo-based batch code [6] is upwards of 40 GB.

Performance results. In Figures 9 and 10 we plot the average queries-per-second vs. communication for the two experi-

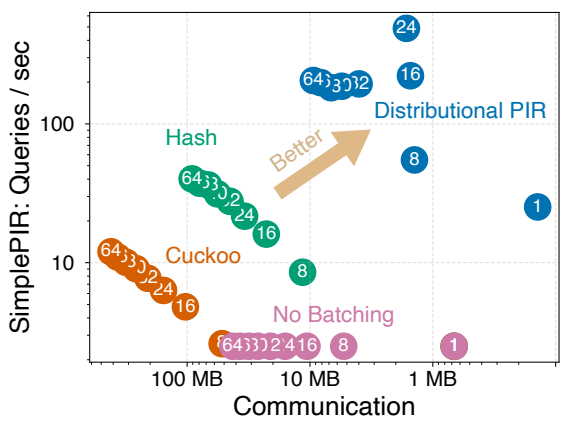

Figure 9: Average queries-per-second vs. total communication when answering a batch of client queries on a 4 GB database that follows the Twitter distribution. All schemes are instantiated with SimplePIR [41] and use 200 MB of client storage. The number in each bubble denotes the batch size.

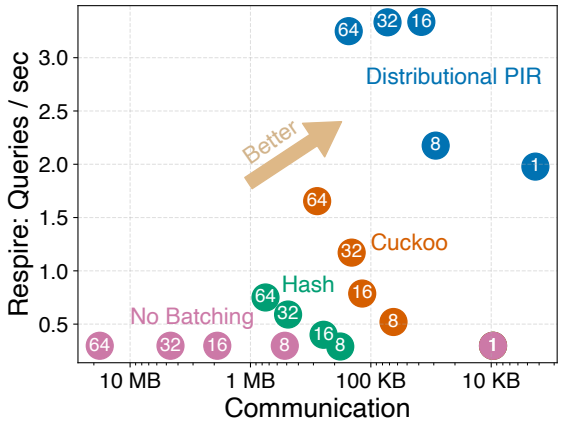

Figure 10: Average queries-per-second vs. total communication when answering a batch of client queries on a 1 GB database that follows the Twitter distribution. All schemes are instantiated with Respire [13]. The number in each bubble denotes the batch size.

ments. In summary, when using SimplePIR, our construction 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, construction increases the queries-per-second by 5.1–77× and reduces communication by 8.1–95×. When using Respire, our construction 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, our construction increases the queries-per-second by 2–8.5× and reduces communication by 1.8–9.73×. In both experiments, our construction improves performance over the baseline even for single queries, unlike batch codes.

Note that our construction'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 our construction 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 of distributional PIR heavily relies on the underlying distribution.

To better understand the performance of the batch-PIR schemes in Figure 9, 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× the size of the original database and b) uses many buckets ($1.5B$). All of that being said, the Cuckoo batch code correctness is very near zero, so in many settings the lower performance may be acceptable.

In Figure 10, 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 Application: Client auditing in Certificate Transparency

We demonstrate that distributional PIR improves on existing techniques for privately auditing signed certificate timestamps (SCTs) in Certificate Transparency (CT).

Background: SCT Auditing. Billions of devices around the globe use CT to detect certificate mis-issuance on the internet [24, 50, 77]. CT ensures that all certificates are stored in a set of publicly auditable logs. Towards this aim, certificate authorities submit issued certificate to log providers, who respond with an SCT; an SCT is essentially a promise that the log provider will include the certificate in their log by a specified deadline. Whenever a browser visits a website, the web server sends the browser its certificate and corresponding SCT. To ensure that log providers act honestly, browsers occasionally check that the CT logs include the certificate attested to by an SCT. This process is called *SCT auditing*.

Chrome–currently the only browser that supports SCT auditing [81]–leaks information about a user's web-browsing history to Google whenever it audits an SCT. In more detail, Chrome triggers an SCT audit for every one in 10,000 SCTs

	Cryptographic Privacy	Server CPU (core-ms)	Comm. (KB)	Storage (MB)
Chrome	✗	–	120	–
PIR	✓	1130	1534	–
Dist. PIR	✓	91	561	6

Table 11: Distributional PIR enables cryptographically-private SCT auditing at a lower cost than traditional PIR. The table displays the average cost of a single SCT auditing query using different schemes—all numbers are normalized to achieve the same auditing rate and correctness. We use YPIR [66] to instantiate PIR-based schemes.

that it receives. Since each SCT is associated with a unique website, auditing the log naively would leak the user's web-browsing history. Instead, Google runs an additional auditing service that stores a list of all SCTs whose corresponding certificates appear in the CT logs. To audit an SCT, Chrome checks whether its SCT appears in this list using a custom protocol that leaks 20 bits of its SCT (and therefore 20 bits of information about which website they visited) to Google.

To eliminate this leakage, several recent works use PIR to privately perform auditing [13, 41, 66]. However, this approach, when coupled with traditional PIR, results in prohibitive costs since the SCT database contains over 5 *billion* entries.

Reducing costs with distributional PIR. To reduce the overheads of private auditing, we use distributional PIR instead of traditional PIR, in particular, our construction from Section 3. Following prior work, clients audit with some constant frequency, e.g. once a week. Because audits occur consistently, clients can avoid failures in our distributional PIR scheme by scheduling their queries. Specifically, clients keep two queues corresponding to the popular and unpopular SCTs. Whenever Chrome flags an SCT for auditing, the client places it into the appropriate queue. When generating an audit, the client adaptively chooses which queue to pop an SCT from based on whether the distributional PIR's Query routine chooses to probe the popular or full database (if a selected queue is empty, the client makes a dummy query). By doing this, queries will never fail, but auditing an unpopular SCT might take longer to complete than when using traditional PIR.

Per-audit costs. We evaluate the per-audit cost of distributional PIR compared to Chrome's current approach and the best-existing PIR-based approach [65]. We instantiate our approach with the distributional PIR construction from Section 3. For the popularity distribution, we rely on the recent work of Ruth et al. [76] that showed that the top one-million websites account for over 95% of page visits for Chrome users. Both PIR-based schemes run PIR over a data structure for approximate private set membership from prior work [41].

When using our distributional PIR construction, clients need to know which websites are most popular. To do this, we have clients store a compressed list of popular domains, totaling 6 MB of client-side storage. This list can be updated relatively infrequently, e.g., once a month, as an out-of-date list only affects the rate of successful audits, not privacy or correctness. We parameterize our scheme with average-case correctness $\kappa_{\mathsf{avg}} = 0.95$ and worst-case correctness $\kappa_{\mathsf{worst}} = 0.05$. Thus, for a user that follows the popularity distribution, on average 5% of audits take two auditing cycles to complete and another 5% take twenty cycles to complete.

Table 11 displays the per-query costs of the various approaches. Compared to PIR-based schemes, we reduce computation by 12× and communication by 3×. Compared to Chrome's approach, we enable cryptographic privacy at a 4× communication overhead.

Increasing audit coverage and frequency. Making explicit use of websites' popularity enables new auditing approaches that improve the coverage and frequency of SCT auditing. See Appendix D for more details.

9 Private Twitter feeds with CrowdSurf

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

To accomplish this, CrowdSurf combines our distributional PIR construction (Section 3) with our optimized PIR scheme (Section 6). We demonstrate that CrowdSurf is more cost-effective than alternative approaches.

Note that CrowdSurf does not fully capture the functionality of Twitter (e.g., users won't see trending posts from users they don't follow), nor does it capture all of Twitter's use cases (e.g., when a user uses the platform primarily to interact with friends). We view CrowdSurf as a step towards more privacy-preserving social networks, not a full replacement for existing platforms like Twitter.

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

	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 12: 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.

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 (Section 7.2). 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 distributional PIR queries 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 (as described in Remark 3.4), and a distributional PIR answer over the second bucket. (Note that this requires a PIR scheme where queries can be re-answered on an updated database.)

PIR optimizations. For a PIR scheme, CrowdSurf uses our linearly homomorphic encryption scheme from Section 6 with hint-compression [40, 55, 65]. Since homomorphic operations on RingLWE-based ciphertexts are not efficient on GPUs, we use two different clusters: a cluster of CPUs for hint-compression, and a cluster of GPUs for everything else.

Learning the distribution. As we discuss in Section 4.2, 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 4.2) 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 this metadata database is more than 100× smaller than the tweet database, we ignore this cost in our evaluation.

9.1 CrowdSurf: End-to-end evaluation

We benchmarked a deployment of CrowdSurf serving users who follow 16–24 people according to the Twitter popularity distribution (Section 7.2) and compared it to the best-performing batch-PIR baseline from Section 7.2.1.

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 Section 7 and the Hash-based batch code for the baseline, so, on average, CrowdSurf recovers ~ 80% of queries and the baseline recovers ~ 90% of tweets.

We set the tweet size to a loose upper-bound of 560 bytes [16]. Constraining the Twitter popularity distribution to users who follow 16–24 users results in a 38 GB database over 73 million unique users. 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 combat this, we squish the database in each chunk to reduce the hint size; this additionally reduces the download size for the client but increases 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 12 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. 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 cost of hint-compression greatly diminishes the gains from our techniques. Thus, improvements to hint-compression performance will immediately increase the relative improvement of CrowdSurf compared to the baseline.

10 Related Work

Batch PIR. Ishai et al. [43] 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. [6, 7] explored more efficient variants of batch codes. While batch PIR can be achieved via black-box use of any PIR scheme and batch code, recent works [13, 58, 70] 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 [29] which, in multiserver 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. [49], 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 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, their scheme satisfies a much stronger notion of correctness than what we target, so 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 [31, 82]. 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 [9] 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 [9, 57] or interact with the user ahead of time [18, 19, 46, 52, 85]. These works are complementary to our own as our main construction makes black-box use of any PIR scheme.

Differentially-private PIR protocols. Several recent works [5, 79] improve the efficiency of traditional PIR schemes by relaxing from information-theoretic or cryptographic privacy guarantees to differential privacy [26]. Similar to our scheme, these protocols get speedups by allowing the server to probe a subset of database entries when answering a query. However, unlike our scheme, the choice of entries to probe *depends* on the client's query, leaking a bounded amount of information to the server. These techniques can be composed with our scheme to obtain even better performance, relaxing both the security and correctness guarantees of standard PIR.

Lattice-based homomorphic encryption schemes. There is a long line of working exploring how to improve the efficiency of lattice-based homomorphic encryption schemes [21, 37, 38, 40, 41, 42, 45, 55, 68, 75, 84]. Most relevant to our optimizations, Li et al. [55] and Henzinger et al. [40] 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. [65] designs a hybrid scheme that uses RingLWE-based assumptions to speed up preprocessing. Our approach outperforms their preprocessing techniques (Section 7.1) and additionally enables fast encryption, but slightly decreases the size of the usable plaintext space (see Section 6 for more details).

Frequency smoothing. In many settings, clients want to privately store data on a cloud provider. Encrypting data is not enough to ensure privacy: when a client's data follows a known popularity distribution, the server can determine the value of encrypted entries based on their access frequency. To mitigate this leakage, recent work uses *frequency smoothing* [32, 61] to artificially modify access patterns to appear uniform. (A similar line of work uses frequency smoothing to secure deterministic encryption schemes for skewed message distributions [34, 48, 62].) Like our work, these systems make explicit use of popularity distributions to improve performance, however, their goal is to *remove distributional skew* rather than take advantage of it. Consequently, they perform best when the distribution is uniform–the exact opposite of our work.

11 Conclusion

This work demonstrates how to speed up private-information-retrieval schemes by taking the "typical" distribution of inputs into account. Crucially, the privacy guarantees of our PIR schemes hold no matter whether a particular user's input is "typical." An exciting direction for future work would be to explore whether we can gain analogous speedups in other cryptographic protocols—secure multiparty computation, fully homomorphic encryption, etc.—through similar techniques.

Acknowledgements. We thank our anonymous shepherd and the USENIX Security reviewers for their detailed feedback, Vinod Vaikuntanathan for answering questions about linearly homomorphic encryption schemes, Nickolai Zeldovich for helpful discussions on the applications of distributional PIR, David J. Wu for his comments on a draft of this work, and Thomas Ristenpart for pointing out a mistake in a previous version of this work. This work was funded in part by NSF Award CNS-2054869 and gifts from Apple, Capital One, Facebook, Google, and Mozilla. Ryan Lehmkuhl was supported by the NSF Graduate Research Fellowship under Grant no. 2141064 and an MIT Sunlin and Priscilla Chou Fellowship. Alexandra Henzinger was supported by the NSF Graduate Research Fellowship under Grant No. 2141064 and an EECS Great Educators Fellowship.

12 Ethics Considerations

As technology becomes increasingly ubiquitous in our lives, it is critical to build systems that protect the privacy of personal information—this aligns with both the "Respect for Persons" and "Beneficence" ethical principles laid out in the Menlo Report [25]. In this work we presented new techniques that further this goal by improving the performance of private information retrieval, enabling users to fetch data from service providers without revealing what they are fetching.

However, there are several possible ethical concerns with applying our techniques in practice, particularly surrounding the "Justice" principle of the Menlo Report [25]. Our techniques weaken the correctness guarantees of traditional PIR, but do so more notably for less popular database entries. Consequently, deploying a system like CrowdSurf for private social-media feeds could lead to: (i) less-popular content-creators receiving less attention than they would have otherwise (ii) users with abnormal usage patterns receiving fewer of the posts they're interested in. Thus, before using our techniques in practice one must consider the balance between the negative impacts experienced by these subsets of users with the privacy benefits that cheaper PIR could provide for all users.

The Twitter dataset we use in our paper is from the work of Gabielkov et al. [30] and anonymizes user identifiers.

13 Compliance with the Open Science Policy

Our implementation of CrowdSurf is open-source and available at `https://github.com/ryanleh/crowdsurf` and `https://zenodo.org/records/14642111`.

The Twitter dataset we use in our paper was provided by the authors of Gabielkov et al. [30]. We do not have permission to publicly release anything beyond the statistics shown in the paper.

References

[1] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR: private information retrieval for everyone. *PoPETs*, 2016.

[2] Ishtiyaque Ahmad, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Pantheon: private retrieval from public key-value store. *Proceedings of the VLDB Endowment*, 16(4):643–656, 2022.

[3] Ishtiyaque Ahmad, Laboni Sarker, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Coeus: a system for oblivious document ranking and retrieval. In *SOSP*, pages 672–690, 2021.

[4] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: metadata-private voice communication over fully untrusted infrastructure. In *OSDI*, 2021.

[5] Kinan Dak Albab, Rawane Issa, Mayank Varia, and Kalman Graffi. Batched differentially private information retrieval. USENIX Security '22.

[6] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. PIR with Compressed Queries and Amortized Query Processing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 962–979. IEEE, 2018. DOI: 10.1109/SP.2018.00062.

[7] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI*, 2016.

[8] Sebastian Angel and Srinath Setty. Unobservable Communication over Fully Untrusted Infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 551–569, 2016.

[9] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers' computation in private information retrieval: PIR with preprocessing. *J. Cryptol.*, 2004.

[10] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: improvements and extensions. In *CCS*, 2016.

[11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully Homomorphic Encryption without Bootstrapping, 2011.

[12] Zvika Brakerski and Vinod Vaikuntanathan. Efficient Fully Homomorphic Encryption from (Standard) LWE. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 97–106. IEEE, 2011. DOI: 10.1109/FOCS.2011.12.

[13] Alexander Burton, Samir Jordan Menon, and David 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.

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

[15] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.

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

[17] Henry Corrigan-Gibbs and Dan Boneh. Prio: private, robust, and scalable computation of aggregate statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*.

[18] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-server private information retrieval with sublinear amortized time. In *EUROCRYPT*, 2022.

[19] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In *EUROCRYPT*, 2020.

[20] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015. ISBN: 9781107043053.

[21] Alex Davidson, Gonçalo Pestana, and Sofía Celi. Frodopir: simple, scalable, single-server private information retrieval. Cryptology ePrint Archive, Paper 2022/981, 2022.

[22] Leo de Castro, Chiraag Juvekar, and Vinod Vaikuntanathan. Fast Vector Oblivious Linear Evaluation from Ring Learning with Errors, 2020.

[23] Leo de Castro, Kevin Lewi, and Edward Suh. Whispir: stateless private information retrieval with low communication. Cryptology ePrint Archive, Paper 2024/266, 2024. URL: https://eprint.iacr.org/2024/266. https://eprint.iacr.org/2024/266.

[24] Joe DeBlasio. Opt-out SCT auditing in Chrome. https://docs.google.com/document/d/16G-Q7iN3kB46GSW5b-sfH5MO3nKSYyEb77YsM7TMZGE/edit.

[25] D Dittrich and E Kenneally. The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research. Technical report, U.S. Department of Homeland Security, 2012. DOI: https://catalog.caida.org/paper/2012_menlo_report_actual_formatted.

[26] Cynthia Dwork. Differential privacy. In ICALP '06.

[27] Tariq Ehsan Elahi, George Danezis, and Ian Goldberg. Privex: private collection of traffic statistics for anonymous communication networks. CCS '11.

[28] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012.

[29] Arman Fazeli, Alexander Vardy, and Eitan Yaakobi. Codes for distributed pir with low storage overhead. In *2015 IEEE International Symposium on Information Theory (ISIT)*.

[30] Maksym Gabielkov, Ashwin Rao, and Arnaud Legout. Studying Social Networks at Scale: Macroscopic Anatomy of the Twitter Social Graph. In *ACM Sigmetrics 2014*, 2014.

[31] Alejandro Gomez-Leos and Anoosheh Heidarzadeh. Single-server private information retrieval with side information under arbitrary popularity profiles. In *2022 IEEE Information Theory Workshop (ITW)*.

[32] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: frequency smoothing for encrypted data stores. In *USENIX Security '20*.

[33] Saikat Guha, Bin Cheng, and Paul Francis. Privad: practical privacy in online advertising. In NSDI'11. USENIX Association, 2011.

[34] Zichen Gui, Kenneth G. Paterson, Sikhar Patranabis, and Bogdan Warinschi. Swissse: system-wide security for searchable symmetric encryption. *Proceedings on Privacy Enhancing Technologies*, 2024.

[35] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with Popcorn. In *NSDI*, 2016.

[36] Hamed Haddadi, Pan Hui, and Ian Brown. Mobiad: private and scalable mobile advertising. In *International Workshop on Mobility in the Evolving Internet Architecture*, 2010.

[37] Shai Halevi and Victor Shoup. Bootstrapping for HElib, 2014.

[38] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. Iron: Private Inference on Transformers. In *Advances in Neural Information Processing Systems*, 2022.

[39] Michaela Hardt and Suman Nath. Privacy-aware personalization for mobile advertising. In CCS '12.

[40] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. Private web search with Tiptoe. In *SOSP*, Koblenz, Germany, October 2023.

[41] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: simple and fast single-server private information retrieval. In *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA. USENIX Association, August 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/henzinger.

[42] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. Cheetah: Lean and Fast Secure {Two-Party} Deep Neural Network Inference. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 809–826, 2022.

[43] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, pages 262–271. ACM, 2004. DOI: 10.1145/1007352.1007396.

[44] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In *STOC*, 2004.

[45] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A Low Latency Framework for Secure Neural Network Inference, 2018.

[46] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 875–892, 2021.

[47] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *FOCS*, 1997.

[48] Marie-Sarah Lacharité and Kenneth G. Paterson. Frequency-smoothing encryption: preventing snapshot attacks on deterministically encrypted data. *IACR Trans. Symmetric Cryptol.*, 2018.

[49] Maximilian Lam, Jeff Johnson, Wenjie Xiong, Kiwan Maeng, Udit Gupta, Yang Li, Liangzhen Lai, Ilias Leontiadis, Minsoo Rhu, Hsien-Hsin S. Lee, Vijay Janapa Reddi, Gu-Yeon Wei, David Brooks, and G. Edward Suh. GPU-based Private Information Retrieval for On-Device Machine Learning Inference, 2023. arXiv: 2301.10904 [cs].

[50] Ben Laurie. Certificate transparency. *Communications of the ACM*, 2014.

[51] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate transparency. RFC 6962, 2013.

[52] Arthur Lazzaretti and Charalampos Papamanthou. Treepir: sublinear-time and polylog-bandwidth private information retrieval from ddh. In *Annual International Cryptology Conference*, pages 284–314. Springer, 2023.

[53] Ryan Lehmkuhl, Alexandra Henzinger, and Henry Corrigan-Gibbs. Distributional private information retrieval. Cryptology ePrint Archive, Paper 2025/132.

[54] Let's encrypt statistics, 2024. URL: https://letsencrypt.org/stats/. Accessed: 2024-12-18.

[55] Baiyu Li, Daniele Micciancio, Mariana Raykova, and Mark Schultz-Wu. Hintless single-server private information retrieval. In *CRYPTO*, 2024.

[56] Suoheng Li, Jie Xu, Mihaela van der Schaar, and Weiping Li. Popularity-driven content caching. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*.

[57] Wei-Kai Lin, Ethan Mook, and Daniel 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*, pages 595–608, 2023.

[58] 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.

[59] Wen-jie Lu, Zhicong Huang, Qizhi Zhang, Yuchen Wang, and Cheng Hong. Squirrel: a scalable secure two-party computation framework for training gradient boosting decision tree. In USENIX Security '23.

[60] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM*, 2013.

[61] Sujaya Maiyya, Sharath Chandra Vemula, Divyakant Agrawal, Amr El Abbadi, and Florian Kerschbaum. Waffle: an online oblivious datastore for protecting data access patterns. *Proc. ACM Manag. Data*, 2023.

[62] Charalampos Mavroforakis, Nathan Chenette, Adam O'Neill, George Kollios, and Ran Canetti. Modular order-preserving encryption, revisited. In SIGMOD '15.

[63] Sarah Meiklejohn, Joe DeBlasio, Devon O'Brien, Chris Thompson, Kevin Yeo, and Emily Stark. SoK: SCT auditing in Certificate Transparency. In *PETS*, 2022.

[64] Samir Jordan Menon and David J. Wu. Spiral: fast, high-rate single-server PIR via FHE composition. In *S&P*, 2022.

[65] Samir Jordan Menon and David J. Wu. YPIR: high-throughput single-server PIR with silent preprocessing. In *USENIX Security Symposium*, 2024.

[66] Samir Jordan Menon and David J. Wu. YPIR: High-Throughput Single-Server PIR with Silent Preprocessing, 2024.

[67] Microsoft SEAL (release 4.1). Microsoft, 2024.

[68] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A Cryptographic Inference Service for Neural Networks, 2020.

[69] Muhammad Haris Mughees, Hao Chen, and Ling Ren. OnionPIR: Response Efficient Single-Server PIR, 2021.

[70] Muhammad Haris Mughees and Ling Ren. Vectorized Batch Private Information Retrieval. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 437–452. IEEE, 2023. DOI: 10.1109/SP46215.2023.10179329.

[71] Hiroki Okada, Rachel Player, Simon Pohmann, and Christian Weinert. Towards practical doubly-efficient private information retrieval. Cryptology ePrint Archive, Paper 2023/1510, 2023. URL: https://eprint.iacr.org/2023/1510.

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

[73] Raluca Ada Popa, Andrew J. Blumberg, Hari Balakrishnan, and Frank H. Li. Privacy and accountability for location-based aggregate statistics. In CCS '11.

[74] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):34:1–34:40, 2009. DOI: 10.1145/1568318.1568324.

[75] Hyeri Roh, Jinsu Yeo, Yeongil Ko, Gu-Yeon Wei, David Brooks, and Woo-Seok 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.

[76] Kimberly Ruth, Aurore Fass, Jonathan Azose, Mark Pearson, Emma Thomas, Caitlin Sadowski, and Zakir Durumeric. A world wide view of browsing the world wide web. In IMC '22. URL: https://doi.org/10.1145/3517745.3561418.

[77] Statista. Browser market share by region, 2024. URL: https://www.statista.com/chart/30734/browser-market-share-by-region/. Accessed: 2024-11-11.

[78] Statista. Distribution of tiktok influencers worldwide as of march 2021, by number of followers. https://www.statista.com/statistics/1250659/distribution-tiktok-influencers-by-number-of-followers-worldwide/, 2022.

[79] Raphael Toledo, George Danezis, and Ian Goldberg. Lower-cost epsilon-private information retrieval. *PoPETS 16*.

[80] Imdad Ullah, Babil Golam Sarwar, Roksana Boreli, Salil S. Kanhere, Stefan Katzenbeisser, and Matthias Hollick. Enabling privacy preserving mobile advertising via private information retrieval. In *2017 IEEE 42nd Conference on Local Computer Networks (LCN)*.

[81] User agents — certificate transparency. URL: https://certificate.transparency.dev/useragents/. Accessed: 2025-01-06.

[82] Sajani Vithana, Karim Banawan, and Sennur Ulukus. Semantic private information retrieval: effects of heterogeneous message sizes and popularities. In *GLOBECOM 2020*.

[83] Andrew C. Yao. Protocols for secure computations. In *SFCS 1982*.

[84] Jiawen Zhang, Jian Liu, Xinpeng Yang, Yinghao Wang, Kejia Chen, Xiaoyang Hou, Kui Ren, and Xiaohu Yang. Secure Transformer Inference Made Non-interactive, 2024.

Experiment A.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:

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

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

[85] Mingxun Zhou, Andrew Park, Elaine Shi, and Wenting Zheng. Piano: extremely simple, single-server pir with sublinear server computation, 2024.

A Additional material from Section 2

A.1 Distributional PIR Definitions

A.1.1 Security

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.

We define security using Experiment A.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:

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

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

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

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 A.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

Experiment A.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 $m_j \neq \perp$ and $m_j \neq D_{I_j}$, return the failure symbol $\perp$.
- Else, return $\{I_j \in [B] \mid m_j = D_{I_j}\}$.

queries are independent of each other, 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.

A.1.2 Correctness

We define all of our correctness notions using Experiment A.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 A.2. Similar to our security definition, we present single-query correctness definitions that imply multi-query ones.

Explicit correctness. 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* κ_{exp} if for all popularity distributions $\mathcal{P}$, databases $D \in \mathcal{M}^N$, and lists of indices $I \in [N]^B$:

$$\Pr\left[\mathsf{Correct}_\Pi(\mathcal{P}, D, I) \neq \perp\right] \geq \kappa_{\mathsf{exp}}.$$

Average-case correctness. 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* κ_{avg} on a probability distribution $\mathcal{P}$ over $[N]$ if, for all databases $D \in \mathcal{M}^N$:

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

Worst-case correctness. 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* κ_{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 $j \in [B]$:

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

Here we enforce a particular 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.1.3 Efficiency

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 A.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 $D \in \mathcal{M}^N$:

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

B Additional material from Section 3

B.1 Background: Batch PIR

A batch-PIR scheme [43] 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 $N \in \mathbb{N}$ 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 security and correctness.

Security. Batch-PIR security is identical to distributional-PIR security (Appendix A.1.1) up to syntactic differences. We give the formal details in the full version of the paper [53].

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* κ if the following holds for all database sizes $N \in \mathbb{N}$, databases $D \in \mathcal{M}^N$ and list of indices $I \in [N]^B$:

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

We say that a batch-PIR scheme is *errorless* if $\kappa = 1$.

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 *communication cost* C if for all databases $D \in \mathcal{M}^N$, list of indices $I \in [N]^B$, and randomness of the PIR algorithms:

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

C Additional material from Section 6

Our linearly homomorphic encryption scheme in Section 6 requires the client to view a RingLWE-type ciphertext as an LWE-type ciphertext. This is a standard technique used in prior work [59, 65] that we briefly describe here.

Define some lattice dimension $n \in \mathbb{N}$, ciphertext modulus $q \in \mathbb{N}$, plaintext modulus $p \in \mathbb{N}$, and error distribution χ over $\mathbb{Z}$. Then for a matrix $\mathbf{A} \xleftarrow{R} \mathbb{Z}_q^{n \times n}$, secret $s \xleftarrow{R} \chi^n$, and

Construction B.1 (Construction from Theorem 3.3)**.** The construction is parameterized by constants $\kappa_{\mathsf{avg}}, \kappa_{\mathsf{worst}} \in [0,1]$, a database size N, and a batch-PIR scheme Π_{batch} defined over a message space $\mathcal{M}$, and batch size B.

$\underline{\mathsf{Dist.Setup}(\mathcal{P}) \to \mathsf{pp}.}$
– Compute $k \leftarrow \mathsf{cdf}_{\mathcal{P}}^{-1}\left(\frac{\kappa_{\mathsf{avg}} - \kappa_{\mathsf{worst}}}{1 - \kappa_{\mathsf{worst}}}\right)$
– Let $\mathcal{L} = (\ell_1, \ell_2, \ldots, \ell_k)$ be the k most-popular indices
– 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 $(\mathsf{pp}_1, \mathsf{pp}_2, \mathcal{L})$

$\underline{\mathsf{Dist.Encode}(\mathsf{pp}, \mathcal{P}, D) \to D_{\mathsf{code}}.}$
– Parse $\mathsf{pp} \to (\_, \_, \mathcal{L} = (\ell_1, \ell_2, \ldots, \ell_k))$, $D \to (d_1, d_2, \ldots, d_N)$
– $D^1_{\mathsf{code}} \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Encode}((d_{\ell_1}, d_{\ell_2}, \cdots, d_{\ell_k}))$
– $D^2_{\mathsf{code}} \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Encode}(D)$
– Output $(D^1_{\mathsf{code}}, D^2_{\mathsf{code}})$

$\underline{\mathsf{Dist.Query}(\mathsf{pp}, I) \to (\mathsf{st}, q).}$
– Parse $\mathsf{pp} \to (\mathsf{pp}_1, \mathsf{pp}_2, \mathcal{L} = (\ell_1, \ell_2, \ldots, \ell_k))$
– Sample a bit $b \leftarrow \mathsf{Bernoulli}(\kappa_{\mathsf{worst}})$
– If $b = 0$:
  ∘ For all $j \in [B]$, compute $\mathcal{E} \leftarrow \{j \;:\; I_j \notin \mathcal{L}\}$
  ∘ Initialize a list $Q = (1)^B$
  ∘ For all $j \in [B]$ and $b \in [k]$, if $I_j = \ell_b$, set $Q_j = b$
  ∘ Compute $(\mathsf{st}, q) \leftarrow \mathsf{Query}(\mathsf{pp}_1, Q)$
– Else, $(\mathsf{st}, q) \leftarrow \Pi_{\mathsf{batch}}.\mathsf{Query}(\mathsf{pp}_2, I)$
– Output $((b, \mathsf{st}), (b, q))$

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

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

error $e \xleftarrow{R} \chi^n$, an LWE ciphertext [74] encrypting n messages $\boldsymbol{\mu} \in \mathbb{Z}_p^n$ has the form:

$$(\mathbf{A}, \mathbf{As} + \mathbf{e} + \lfloor q/p \rfloor \cdot \boldsymbol{\mu}) \in \mathbb{Z}_q^{n \times n} \times \mathbb{Z}_q^n.$$

A RingLWE-type ciphertext [28] has the same form except it is defined over the polynomial ring $R = \mathbb{Z}[x]/(x^n + 1)$ rather than the integer group $\mathbb{Z}_q$. In particular, the matrix $\mathbf{A}$ is replaced with a polynomial $\mathbf{a} \xleftarrow{R} R_q$, and $\mathbf{s}$, $\mathbf{e}$, and $\lfloor q/p \rfloor \cdot \boldsymbol{\mu}$ are each encoded as polynomials in R_q. Thanks to the structure of polynomial multiplication in R_q, the polynomial product $\mathbf{a} \cdot \mathbf{s}$ can be expressed as a matrix product $\mathbf{A}'\mathbf{s}$ using a negacylic matrix $\mathbf{A}'$ (see YPIR [65] or Squirrel [59, Section 4.4] for more details). As a result, every RingLWE ciphertext can be implicitly converted an LWE ciphertext with a particularly-structured $\mathbf{A}$ matrix.

D Increasing the coverage and frequency of SCT audits.

We describe two new approaches to SCT auditing that explicitly utilize websites' popularity. These approaches improve two aspects of SCT auditing: the coverage of audits, and the frequency of auditing.

1. *Better audit coverage*: Under Chrome's protocol, many websites may never be audited, even if thousands of people visit them regularly. In particular, since Chrome only rolls for an audit attempt once a website is visited by a client, less-popular websites have a much lower chance of being audited. To mitigate this, an auditing scheme could explicitly set a higher sampling rate for less-popular websites to ensure that all websites are audited. Note that doing this requires a generalization of the definition of distributional PIR as presented thus far, see the following section for more details.
2. *Protection from targeted attacks*: In order to detect targeted attacks–where an attacker spoofs a certificate to a select group of users–the frequency of SCT auditing would need to be increased by several orders of magnitude. Instead of doing this for all domains, this could be done for only popular domains, protecting against targeted attacks for 95% of websites that users' visit at a much smaller overhead.

D.1 Generalizing distributional PIR

The speedups in distributional PIR come from probing a fraction of the database (on average) to answers queries; using side-information (i.e. the popularity distribution) reduces the probability that this results in a failure. Our presentation thus far has focused on a setting where the goal is to successfully answer as many queries as possible, i.e., failures occur more frequently for less-popular database entries. However, some applications may better synergize with different failure profiles. Consider the following PIR applications:

- *SCT Auditing with better coverage* [50, 51, 63]. This application was discussed above. Clients use PIR to ensure websites possess valid certificates. Here the goal is to audit all websites. As a result, it is best for failures to occur for queries on the *most popular websites* since these will be over-audited by clients.
- *Private Ads* [33, 36, 39, 80]. Clients use PIR to fetch a batch of ads from an ad broker; ads are chosen based on the client's personal interests. Locally, clients use more fine-grained personal information to select a subset of the ads to display. Here the goal is to maximize the server's expected profit. As a result, it is best for failures to occur for queries on ads that produce low returns for the server.

To capture these examples, we introduce the notion of a *utility function* $\mathcal{U}$ that, given a list of database indices, returns a constant (termed the utility) that reflects the "value" of receiving those database entries. We then introduce a corresponding variant of average-case correctness.

Average-case utility: We say that a distributional-PIR scheme has average-case utility κ_{avg} for a utility function $\mathcal{U}$, probability distribution $\mathcal{P}$ and batch size B if, when the client queries for a list of B indices sampled i.i.d. from the distribution $\mathcal{P}$, the client recovers a κ_{avg} fraction of the total utility of its query indices, in expectation over the random draws from $\mathcal{P}$ and the randomness of the PIR algorithms.

If the utility function assigns the same utility to each database entry, then average-case utility is equivalent to average-case correctness. In the full version of the paper [53] we formalize this notion and show that average-case utility captures the applications above when instantiated with a class of utility functions called linear utility functions.

Finally, we prove that for any linear utility function and batch size 1, average-case utility reduces to average-case correctness. In other words, for any distribution $\mathcal{P}$ and linear utility function $\mathcal{U}$, we can construct a distribution $\mathcal{P}'$ such that, if a distributional-PIR scheme has average-case correctness κ_{avg} on $\mathcal{P}'$, then it also achieves average-case utility κ_{avg} on $\mathcal{P}$. See the full version of the paper for more details [53].