One Server for the Price of Two: Simple and Fast Single-Server Private Information Retrieval

Alexandra Henzinger MIT
Matthew M. Hong MIT
Henry Corrigan-Gibbs MIT
Sarah Meiklejohn Google
Vinod Vaikuntanathan MIT

Abstract. We present SimplePIR, the fastest private information retrieval (PIR) scheme known to date. SimplePIR is a single-server PIR scheme, whose security holds under the learning-with-errors assumption. To answer a client's PIR query, the SimplePIR server performs one 32-bit multiplication and one 32-bit addition per database byte. SimplePIR achieves 6.5 GB/s/core server throughput, which is 7% faster than the fastest *two-server* PIR schemes (which require non-colluding servers). SimplePIR has relatively large communication costs: to make queries to a 1 GB database, the client must download a 124 MB "hint" about the database contents; thereafter, the client may make an unbounded number of queries, each requiring 242 KB of communication. We present a second single-server scheme, DoublePIR, that shrinks the hint to 16 MB at the cost of slightly higher per-query communication (345 KB) and slightly lower throughput (5.2 GB/s/core). Finally, we apply our PIR schemes, together with a new data structure for approximate set membership, to the problem of private auditing in Certificate Transparency. We achieve a strictly stronger notion of privacy than Google Chrome's current approach with a modest, 13× larger communication overhead.

1 Introduction

In a Private Information Retrieval (PIR) protocol [26, 53], a database server holds an array of N records. A client wants to fetch record $i \in \{1, \cdots, N\}$ from the server, without revealing the index i that it desires to the server. PIR has applications to systems for private database search [75, 81], metadata-hiding messaging [9, 10], private media consumption [48], credential breach reporting [4, 57, 80, 82], and private blocklist lookups [52], among others.

Modern PIR schemes require surprisingly little communication: with a single database server, and under modest cryptographic assumptions [20, 44, 70], the total communication required to fetch a database record grows only polylogarithmically with the number of records, N. Unfortunately, PIR schemes are computationally expensive: the server must touch every bit of the database to answer even a single client query [12], since otherwise the PIR scheme leaks information about which database records the client is *not* interested in. (Recently, there have been a number of PIR schemes that preprocess the database to achieve a runtime sublinear in the database size, but all known approaches either require client-specific preprocessing [29, 30, 52, 77, 85] or impractically large server storage [12, 16, 21].) Thus, a hard limit on the performance of PIR schemes is the speed with which the PIR server can read the database from memory—roughly 12.8 GB/s/core on our machine.

In the standard setting, in which the client communicates with only a single database server, the performance of existing PIR schemes is far from this theoretical limit: we measure that the fastest single-server PIR schemes achieve a throughput of 259 MB/s/core, or 2% of the maximum possible [66], on a database of hundred-byte records. It is possible to push the performance up to 1.3 GB/s/core when the database records are hundreds of kilobytes long, though that parameter setting is not relevant for many PIR applications, including our application to Certificate Transparency.

When the client can communicate with multiple *non-colluding* database servers [26], faster PIR schemes exist. In the two-server setting, schemes based on distributed point functions [15, 45] can achieve 5.4 GB/s/core throughput, or 42% of the maximum possible. Here, two servers must answer each query, which halves the throughput per core. Unfortunately, these two-server schemes are cumbersome to deploy, since they rely on multiple coordinating yet independent infrastructure providers. In addition, their security is brittle, as it stems from a non-collusion assumption rather than from cryptographic hardness. Thus, existing PIR schemes either have poor performance—in the single-server setting—or undesirable trust assumptions—in the multi-server setting.

In this paper, we present two new single-server PIR schemes; our faster scheme achieves the highest server throughput of *any* existing PIR protocol, including multi-server ones. Our schemes have modest per-query communication of a few hundred KB on a database of one billion short rows, and their security is based on Regev's learning-with-errors assumption [76]. In addition, our schemes are relatively simple to explain and easy to implement: our complete implementation of both schemes together requires roughly 1,200 lines of Go code, plus 80 lines of C, and uses no external libraries.

More specifically, our first scheme, SimplePIR, achieves a server throughput of 6.5 GB/s/core, though it requires the client to download a relatively large "hint" about the database contents before making its queries. On a database of size N, the hint has size roughly $1024\sqrt{N}$ bytes . The hint is not client-specific, and a client can reuse the hint over many queries,

so the amortized communication cost per query can be small. Our second scheme, DoublePIR, achieves slightly lower server throughput of 5.2 GB/s/core, but shrinks the hint to roughly 16 MB for a database of one-byte records—independent of the number of records in the database.

Our techniques. We now summarize the technical ideas behind our results.

Recap: Single-server PIR. Our starting point is the single-server PIR construction of Kushilevitz and Ostrovsky [53]. In their scheme, the PIR server represents an N-record database as a matrix $\mathbf{D}$ of dimension $\sqrt{N}$ by $\sqrt{N}$. To fetch the database record in row i and column j, the client sends the server the encryption $E(\mathbf{q})$ of a dimension-$\sqrt{N}$ vector that is zero everywhere except that it has a "1" in index j. If the encryption scheme is linearly homomorphic, the server can compute the matrix-vector product $\mathbf{D} \cdot E(\mathbf{q}) = E(\mathbf{D} \cdot \mathbf{q})$ under encryption and return the result to the client. The client decrypts to recover $\mathbf{D} \cdot \mathbf{q}$ which, by construction, is the j-th column of the database, as desired. The total communication grows as $\sqrt{N}$.

SimplePIR from linearly homomorphic encryption with preprocessing. The PIR server's throughput here is limited by the speed with which it can compute the product of the plaintext matrix $\mathbf{D}$ with the encrypted vector $E(\mathbf{q})$. Our observation in SimplePIR (Section 4) is that, using Regev's learning-with-errors-based encryption scheme [76], the server can perform the vast majority of the work of computing the matrix-vector product $\mathbf{D} \cdot E(\mathbf{q})$ *in advance*—before the client even makes its query. The server's preprocessing depends only on the database $\mathbf{D}$ and the public parameters of the Regev encryption scheme, so the server can reuse this preprocessing work across many queries from many independent clients. After this preprocessing step, to answer a client's query, the server needs to compute only roughly N 32-bit integer multiplications and additions on a database of N bytes. The only catch is that the client must download a "hint" about the database contents after this preprocessing step—the hint accounts for the bulk of the communication cost in SimplePIR.

DoublePIR from one recursive step. The idea behind DoublePIR (Section 5) comes from the original Kushilevitz and Ostrovsky paper [53]: in SimplePIR, the client downloads the hint from the server, along with a dimension-$\sqrt{N}$ encrypted vector. However, to recover its record of interest, the client only needs one small part of the hint and one component of this vector. We show how the client can use SimplePIR recursively on the hint and this vector to fetch its desired database record at a reduced communication cost. To minimize the concrete costs, we make non-black-box use of SimplePIR in this recursive construction, which saves a factor of the lattice dimension, which is roughly 1024 for our parameters, over a naïve design.

Application to Certificate Transparency. Finally, we evaluate our PIR schemes in the context of the application of *signed certificate timestamp (SCT) auditing* in Certificate Transparency. In this auditing application, a server holds a set S of strings and a client (web browser) wants to test whether a particular string σ, representing an SCT, appears in the set S, while hiding σ from the server. (The string σ reveals information about which websites a client has visited.) Google Chrome currently proposes implementing this auditing step using a solution that provides k-anonymity for $k = 1000$ [33].

Along the way, we construct a new data structure (Section 6) for more efficiently solving this type of private set-membership problem using PIR, when a constant rate of false positives is acceptable (as in our application). In this setting, standard Bloom filters [14] and approaches based on PIR by keywords [25] require the client to perform PIR over a database of λN bits (if the set S has size N and $\lambda \approx 128$ is a security parameter). In contrast, our data structure requires performing PIR over only $8N$ bits—giving a roughly 16× speedup in our application.

Google's current solution to SCT auditing, which provides k-anonymity rather than full cryptographic privacy, requires the client to communicate 240 B on average per TLS connection. Our solution, which provides cryptographic privacy, requires 1.5 KB and 0.004 core-seconds of server compute on average per TLS connection, along with 16 MB of client download and 400 KB of client storage every week to maintain the hint.

Limitations. Our new PIR schemes come with two main downsides. First, our client must download a "hint:" on databases gigabytes in size, the hint is tens of megabytes. If a client makes only one query, this hint download dominates the overall communication. Second, our schemes' online communication is on the order of hundreds of kilobytes on databases with short entries—which is 10× larger than in some prior work. On databases with with larger entries, this gap increases further. Nevertheless, we believe that SimplePIR and DoublePIR represent an exciting new point in the PIR design space: large computation savings, along with a conceptually simple design and small, stand-alone codebase, at the cost of modest communication and storage overheads.

Our contributions. In summary, our contributions are:

- two new high-throughput single-server private information retrieval protocols (Sections 4 and 5),
- a new data structure for private set membership using PIR (Section 6) and its application to private auditing in Certificate Transparency (Section 7), and
- the evaluation of these schemes, using a new open-source implementation (Section 8).

2 Related work and comparison

Chor, Goldreich, Kushilevitz and Sudan [26] introduced PIR in the multi-server setting and Kushilevitz and Ostrovsky [53] gave the first construction of single-server PIR. Their scheme uses a linearly homomorphic encryption scheme that expands ℓ-bit plaintexts to $\ell \cdot F$-bit ciphertexts. We call F the *expansion factor* of the encryption scheme. Then, on a database of N

Scheme	Servers	Communication	No per-client storage (on the server)	polylog(n) overhead (on the server)	Max. achievable throughput/core
DPF PIR [15, 51]	2	$\log N$	✓	✓	5,381 MB/s
XOR	2	–	–	–	6,067 MB/s
SealPIR [9] ($d = 2$)	1	$\sqrt{N}$	✗	✓	97 MB/s
MulPIR [8] ($d = 2$)	1	$\sqrt{N}$	✗	✓	69 MB/s*
FastPIR [6]	1	N	✗	✓	215 MB/s
OnionPIR [67]	1	$\log N$	✗	✓	104 MB/s
Spiral family [66]	1	$\log N$	✗	✓	1,314 MB/s
KO [53]+Paillier [71]	1	N^ϵ	✓	✗	0.131 MB/s
XPIR [5] ($d = 2$)	1	$\sqrt{N}$	✓	✓	142 MB/s$^\heartsuit$
SimplePIR (§4)	1	$\sqrt{N}$	✓	✓	6,496 MB/s
DoublePIR (§5)	1	$\sqrt{N}$	✓	✓	5,217 MB/s

Table 1: A comparison of PIR schemes on database size N and security parameter n. The memory bandwidth of our machine is 12.8 GB/s/core; as each of these schemes must read the whole database from memory, their throughput cannot be higher than this upper bound. XOR refers to a benchmark that performs a linear scan of XORs over the database (see Section 8.1). The overhead column indicates whether the server computation per database bit is at most polylogarithmic in n. The throughput column gives the maximum throughput we measured for any database size and record size. The database sizes we used for these numbers are in Appendix A. The throughput is normalized by the number of cores, i.e., divided by two for 2-server PIR schemes (DPF PIR and XOR). *No open-source implementation available. This is the throughput reported in the MulPIR paper [8]. $^\heartsuit$This is the XPIR throughput reported by SealPIR [9].

bits and any dimension parameter $d \in \{1, 2, 3, \dots\}$, their PIR construction has communication roughly $N^{1/d} F^{d-1}$. The server must perform roughly NF^{d-1} homomorphic operations in the process of answering the client's query.

The Damgård-Jurik [32] cryptosystem has expansion factor $F \approx 1$, which yields very communication-efficient PIR schemes [60]. It is possible to construct PIR with similar communication efficiency from an array of cryptographic assumptions [20, 23, 36]. However, these schemes are all costly in computation: for each *bit* of the database, the server must perform work polynomial in the security parameter.

Lattice-based PIR. To drive down this computational cost, recent PIR schemes instantiate the Kushilevitz-Ostrovsky construction using encryption schemes based on the ring learning-with-errors problem ("Ring LWE") [63]. In these schemes, for each bit of the database, the server performs work *polylogarithmic* in the security parameter—rather than polynomial. However, these savings in computation come at the cost of a larger expansion factor ($F \approx 10$), which increases the communication as the dimension parameter d cannot be too large. For example, XPIR [5] takes $d = 2$. In addition, the client in the Kushilevitz-Ostrovsky scheme must upload $N^{1/d}$ ciphertexts, and each ring-LWE ciphertext is at least thousands of kilobytes in size. This imposes large absolute communication costs (e.g., tens of MB per query, on a database of hundreds of MB).

SealPIR [9] shows that the client can *compress* the ciphertexts in an XPIR-style scheme before uploading them. The server can then expand these ciphertexts using homomorphic operations. (FastPIR [6] uses a similar idea to compress responses.) This optimization reduces the communication costs by orders of magnitude, though it requires the server to store some per-client information ("key-switching hints")—essentially, encryptions of the client's secret decryption key—that is megabytes in size and that the client must upload to the server before it makes any queries.

MulPIR [8], OnionPIR [67], and Spiral [66] additionally use *fully* homomorphic encryption [42] to further reduce the communication cost. In Spiral [66], for example, the cost grows roughly as $N^{1/d} F$, where the exponent on the F term is now 1 instead of $d - 1$. Building on ideas of Gentry and Halevi [43], Spiral shows how to achieve this decrease in communication cost while keeping the throughput high: up to 259 MB/s on a database of short records. (When the database records are long, Spiral does not use the SealPIR query compression technique and gets throughput as large as 1,314 MB/s at the cost of increased communication.)

Plain learning with errors. We base our PIR schemes on the standard learning-with-errors (LWE) problem—not the ring variant. The expansion factor of the standard LWE-based encryption scheme, Regev encryption [76], is roughly $F = n \approx 1024$, where n is the lattice security parameter. This large expansion factor means that a direct application of Regev encryption to the Kushilevitz-Ostrovsky PIR scheme would be disastrous in terms of communication and computation. Our innovation is to show that the server can do the bulk of its work *in advance*, and reuse it over multiple clients.

Aside from the fact that our scheme is based on a *weaker cryptographic assumption*, namely plain LWE as opposed to ring LWE, this strategy yields multiple benefits:

1. Our LWE-based schemes are *simple* to implement: they require no polynomial arithmetic or fast Fourier transforms.
2. Our schemes do not require the server to store any extra per-client state. In contrast, many schemes based on Ring LWE [8, 9, 66, 67] rely on optimizations that require the server to store one "key-switching hint" for each client.
3. Our schemes are *faster*. We avoid the costs associated with ciphertext compression and expansion. In addition, since we only need our encryption scheme to be linearly (*not* fully) homomorphic, we can use smaller and more efficient lattice parameters.

The drawback of our schemes is that they have larger communication cost, especially when the client makes only a single query (so the client cannot amortize the offline download cost over multiple queries) or when the database records are long.

Preprocessing and PIR. The server in our PIR schemes

performs some client-independent preprocessing. Prior work uses server-side preprocessing—either one-time [12, 16, 21] or per-client [29, 30, 52, 77, 85]—to build PIR where the server online work is *sublinear* in the database size. Prior work also proves strong lower bounds on the performance of any such PIR with preprocessing schemes [12, 29, 30, 74]. In contrast, in this work, we use preprocessing to build PIR where the amortized per-query server work is still *linear*, but it is concretely efficient.

Hardware acceleration for PIR. Recent work improves the throughput of both single-server [58] and multi-server [49] PIR using hardware acceleration. This approach is complementary to ours, as it may further speed up our new PIR protocols.

Privacy and certificate transparency. Lueks and Goldberg [62] and Kales, Omolola, and Ramacher [51] propose using *multi*-server PIR for auditing in certificate transparency. We work in the *single*-server setting, where the client communicates with a separate audit server (e.g., Google, in the application to Chrome). Further, we introduce a new set-membership data structure to reduce the cost of auditing (Section 6). We discuss existing approaches to auditing in Section 7.2.

3 Background and definitions

Notation. For a probability distribution χ, we use $x \xleftarrow{R} \chi$ to indicate that x is a uniformly random sample from χ. For a finite set S, we use $x \xleftarrow{R} S$ to denote sampling x uniformly at random from S. We use $\mathbb{N}$ to represent the natural numbers and $\mathbb{Z}_p$ to represent integers modulo p. All logarithms are to the base two. For $n \in \mathbb{N}$, we let $[n]$ denote the set $\{1, \ldots, n\}$. Throughout, we assume that values like $\sqrt{N}$ are integral, wherever doing so is essentially without loss of generality. Algorithms are modeled as RAM programs and their runtime is measured in terms of the number of RAM instructions executed. We use the symbols MB and MiB, as well as GB and GiB, synonymously.

3.1 Learning with errors (LWE)

The security of our PIR schemes relies on the decision version of the learning-with-errors assumption [76]. The assumption is parameterized by the dimension of the LWE secret $n \in \mathbb{N}$, the number of samples $m \in \mathbb{N}$, the integer modulus $q \geq 2$, and an error distribution χ over $\mathbb{Z}$. The LWE assumption then asserts that for a matrix $\mathbf{A} \xleftarrow{R} \mathbb{Z}_q^{m \times n}$, a secret $\mathbf{s} \xleftarrow{R} \mathbb{Z}_q^n$, an error vector $\mathbf{e} \xleftarrow{R} \chi^m$, and a random vector $\mathbf{r} \xleftarrow{R} \mathbb{Z}_q^m$, the following distributions are computationally indistinguishable:

$$\{(\mathbf{A}, \mathbf{As} + \mathbf{e})\} \overset{c}{\approx} \{(\mathbf{A}, \mathbf{r})\}.$$

More specifically, the (n, q, χ)-LWE problem with m samples is (T, ϵ)-hard if all adversaries running in time T have advantage at most ϵ in distinguishing the two distributions. In Section 4.2, we give concrete values for the LWE parameters.

Secret-key Regev encryption. Regev [76] gives a secret-key encryption scheme that is secure under the LWE assumption. With LWE parameters (n, q, χ) and a plaintext modulus p, the Regev secret key is a vector $\mathbf{s} \xleftarrow{R} \mathbb{Z}_q^n$. The Regev encryption of a message $\mu \in \mathbb{Z}_p$ is

$$(\mathbf{a}, c) = (\mathbf{a}, \mathbf{a}^\intercal \mathbf{s} + e + \lfloor q/p \rfloor \cdot \mu) \in \mathbb{Z}_q^n \times \mathbb{Z}_q,$$

for $e \xleftarrow{R} \chi$. To decrypt the ciphertext, anyone who knows the secret $\mathbf{s}$ can compute $c - \mathbf{a}^\intercal \mathbf{s} \bmod q$ and round the result to the nearest multiple of $\lfloor q/p \rfloor$. Decryption succeeds as long as the absolute value of the error sampled from the error distribution χ is smaller than $\frac{1}{2} \cdot \lfloor q/p \rfloor$. We say that a setting of the Regev parameters supports *correctness error* δ if the probability of a decryption error is at most δ (over the encryption algorithm's randomness). Regev encryption is additively homomorphic, since given two ciphertexts $(\mathbf{a}_1, c_1)$ and $(\mathbf{a}_2, c_2)$, their sum $(\mathbf{a}_1 + \mathbf{a}_2, c_1 + c_2)$ decrypts to the sum of the plaintexts, provided again that the error remains sufficiently small.

3.2 Private information retrieval with hints

We now give the syntax and security definitions for the type of PIR schemes we construct. Our form of PIR is very similar to the standard single-server PIR schemes [26, 53]. The primary distinction is that we allow the PIR server to preprocess the database ahead of time and to output two "hints": one that the server stores locally, and another that the server sends to each client. This preprocessing allows the PIR server to push much of its computational work into an offline phase that takes place before the client makes its query. In our constructions, both hints are small—they have size sublinear in the database size. In addition, all clients use the same hint and a client can reuse the same hint for an unbounded number of PIR queries.

Remark 3.1 (Handling database updates). As PIR schemes with preprocessing perform some precomputation over the database, the server inherently needs to repeat some of this work if the database contents change. Related work investigates how to minimize the amount of computation and communication that such database updates incur, in both a black-box [52] and a protocol-specific [64] manner. We address how to handle updates in our schemes in Appendices C.3 and E.3.

A PIR-with-preprocessing scheme [12], over plaintext space $\mathcal{D}$ and database size $N \in \mathbb{Z}$, consists of four routines, which all take the security parameter as an implicit input:

- $\mathsf{Setup}(\mathsf{db}) \to (\mathsf{hint}_s, \mathsf{hint}_c)$. Given a database $\mathsf{db} \in \mathcal{D}^N$, output preprocessed hints for the server and the client.
- $\mathsf{Query}(i) \to (\mathsf{st}, \mathsf{qu})$. Given an index $i \in [N]$, output a secret client state st and a database query qu.
- $\mathsf{Answer}(\mathsf{db}, \mathsf{hint}_s, \mathsf{qu}) \to \mathsf{ans}$. Given the database db, a server hint hint_s, and a client query qu, output an answer ans.
- $\mathsf{Recover}(\mathsf{st}, \mathsf{hint}_c, \mathsf{ans}) \to d$. Given a secret client state st, a client hint hint_c, and a query answer ans, output a database record $d \in \mathcal{D}$.

Correctness. When the client and the server execute the PIR protocol faithfully, the client should recover its desired

database record with all but negligible probability in the implicit correctness parameter. Formally, we say that a PIR scheme has *correctness error* δ if, on database size $N \in \mathbb{N}$, for all databases $\mathsf{db} = (d_1, \ldots, d_N) \in \mathcal{D}^N$ and for all indices $i \in [N]$, the following probability is at least $1 - \delta$:

$$\Pr\left[d_i = \hat{d}_i : \begin{array}{r} (\mathsf{hint}_s, \mathsf{hint}_c) \leftarrow \mathsf{Setup}(\mathsf{db}) \\ (\mathsf{st}, \mathsf{qu}) \leftarrow \mathsf{Query}(i) \\ \mathsf{ans} \leftarrow \mathsf{Answer}(\mathsf{db}, \mathsf{hint}_s, \mathsf{qu}) \\ \hat{d}_i \leftarrow \mathsf{Recover}(\mathsf{st}, \mathsf{hint}_c, \mathsf{ans}) \end{array} \right].$$

For the PIR scheme to be non-trivial, the total client-to-server communication should be smaller than the bitlength of the database. That is, it must hold that $|\mathsf{hint}_c| + |\mathsf{qu}| + |\mathsf{ans}| \ll |\mathsf{db}|$.

Security. The client's query should reveal no information about its desired database record. That is, we say that a PIR scheme is (T, ϵ)-*secure* if, for all adversaries $\mathcal{A}$ running in time at most T, on database size $N \in \mathbb{N}$, and for all $i, j \in [N]$,

$$\big|\Pr[\mathcal{A}(1^N, \mathsf{qu}) = 1 : (\mathsf{st}, \mathsf{qu}) \leftarrow \mathsf{Query}(i)] - \Pr[\mathcal{A}(1^N, \mathsf{qu}) = 1 : (\mathsf{st}, \mathsf{qu}) \leftarrow \mathsf{Query}(j)]\big| \leq \epsilon.$$

Remark 3.2 (Stateless client). The client in our PIR schemes does not hold any secret state across queries. In contrast, in SealPIR [9] and related schemes, the client builds its queries using persistent, long-term cryptographic secrets. We show in Appendix B that, in certain settings, a malicious PIR server can perform a state-recovery attack against these schemes and thus break client privacy for both past and future queries. Our stateless schemes are not vulnerable to such attacks.

4 SimplePIR

In this section, we present our first PIR scheme, SimplePIR. SimplePIR is the fastest PIR scheme known to date in terms of throughput per second per core—even when compared against two-server PIR schemes (Table 1). In particular, we prove the following theorem:

Informal Theorem 4.1. *On database size N, let $p \in \mathbb{N}$ be a suitable plaintext modulus for secret-key Regev encryption with LWE parameters (n, q, χ), achieving (T, ϵ)-security for $\sqrt{N}$ LWE samples and supporting $\sqrt{N}$ homomorphic additions with correctness error δ (cf. Section 4.2). Then, for a random LWE matrix $\mathbf{A} \in \mathbb{Z}_q^{\sqrt{N} \times n}$, SimplePIR is a $(T - O(\sqrt{N}), 2\epsilon)$-secure PIR scheme on database size N, over plaintext space $\mathbb{Z}_p$, with correctness error δ.*

We give a formal description of SimplePIR in Figure 2 and we prove its security and correctness in Appendix C.

Remark 4.1 (Concrete costs of SimplePIR). Using the parameters of Informal Theorem 4.1, we give SimplePIR's concrete costs, with no hidden constants, in terms of operations (i.e., integer additions and multiplications) over $\mathbb{Z}_q$. In a one-time public preprocessing phase, SimplePIR requires:

Construction: SimplePIR. The parameters of the construction are a database size N, LWE parameters (n, q, χ), a plaintext modulus $p \ll q$, and a LWE matrix $\mathbf{A} \in \mathbb{Z}_q^{\sqrt{N} \times n}$ (sampled in practice using a hash function). The database consists of N values in $\mathbb{Z}_p$, which we represent as a matrix in $\mathbb{Z}_p^{\sqrt{N} \times \sqrt{N}}$. Define the scalar $\Delta := \lfloor q/p \rfloor \in \mathbb{Z}$.

$\mathsf{Setup}(\mathsf{db} \in \mathbb{Z}_p^{\sqrt{N} \times \sqrt{N}}) \rightarrow (\mathsf{hint}_s, \mathsf{hint}_c)$.

- Return $(\mathsf{hint}_s, \mathsf{hint}_c) \leftarrow (\perp, \mathsf{db} \cdot \mathbf{A} \in \mathbb{Z}_q^{\sqrt{N} \times n})$.

$\mathsf{Query}(i \in [N]) \rightarrow (\mathsf{st}, \mathsf{qu})$.

- Write i as a pair $(i_{\text{row}}, i_{\text{col}}) \in [\sqrt{N}]^2$.
- Sample $\mathbf{s} \xleftarrow{R} \mathbb{Z}_q^n$ and $\mathbf{e} \xleftarrow{R} \chi^{\sqrt{N}}$.
- Compute $\mathsf{qu} \leftarrow (\mathbf{A}\mathbf{s} + \mathbf{e} + \Delta \cdot \mathbf{u}_{i_{\text{col}}}) \in \mathbb{Z}_q^{\sqrt{N}}$, where $\mathbf{u}_{i_{\text{col}}}$ is the vector of all zeros with a single '1' at index i_{col}.
- Return $(\mathsf{st}, \mathsf{qu}) \leftarrow ((i_{\text{row}}, \mathbf{s}), \mathsf{qu})$.

$\mathsf{Answer}(\mathsf{db} \in \mathbb{Z}_p^{\sqrt{N} \times \sqrt{N}}, \mathsf{hint}_s, \mathsf{qu} \in \mathbb{Z}_q^{\sqrt{N}}) \rightarrow \mathsf{ans}$.

- Return $\mathsf{ans} \leftarrow \mathsf{db} \cdot \mathsf{qu} \in \mathbb{Z}_q^{\sqrt{N}}$.

$\mathsf{Recover}(\mathsf{st}, \mathsf{hint}_c \in \mathbb{Z}_q^{\sqrt{N} \times n}, \mathsf{ans} \in \mathbb{Z}_q^{\sqrt{N}}) \rightarrow d$.

- Parse $(i_{\text{row}} \in [\sqrt{N}], \mathbf{s} \in \mathbb{Z}_q^n) \leftarrow \mathsf{st}$.
- Compute $\hat{d} \leftarrow (\mathsf{ans}[i_{\text{row}}] - \mathsf{hint}_c[i_{\text{row}}, :] \cdot \mathbf{s}) \in \mathbb{Z}_q$, where $\mathsf{ans}[i_{\text{row}}]$ denotes component i_{row} of ans and $\mathsf{hint}_c[i_{\text{row}}, :]$ denotes row i_{row} of hint_c.
- Return $d \leftarrow \mathsf{Round}_\Delta(\hat{d})/\Delta \in \mathbb{Z}_p$, which is $\hat{d}$ rounded to the nearest multiple of Δ and then divided by Δ.

Figure 2: The SimplePIR protocol.

- the server to perform $2nN$ operations in $\mathbb{Z}_q$, and
- the client to download $n\sqrt{N}$ elements in $\mathbb{Z}_q$,

where our implementation takes $n = 2^{10}$ and $q = 2^{32}$ to achieve 128-bit security against the best known attacks [7].

On each query, SimplePIR requires

- the client to upload $\sqrt{N}$ elements in $\mathbb{Z}_q$,
- the server to perform $2N$ operations in $\mathbb{Z}_q$, and
- the client to download $\sqrt{N}$ elements in $\mathbb{Z}_q$.

4.1 Technical ideas

We now discuss the SimplePIR construction in more detail.

The simplest non-trivial single-server PIR schemes [22, 53, 60] take the following "square-root" approach: given an N-element database, the server stores this database as a $\sqrt{N}$-by-$\sqrt{N}$ square matrix. Meanwhile, a client who wishes to query for database entry $i \in [N]$ decomposes index i into the pair of coordinates $(i_{\text{row}}, i_{\text{col}}) \in [\sqrt{N}]^2$. Then, the client builds a unit vector $\mathbf{u}_{i_{\text{col}}}$ in $\mathbb{Z}_2^{\sqrt{N}}$ (i.e., the vector of all zeros with a single '1' at index i_{col}), element-wise encrypts it with a linearly

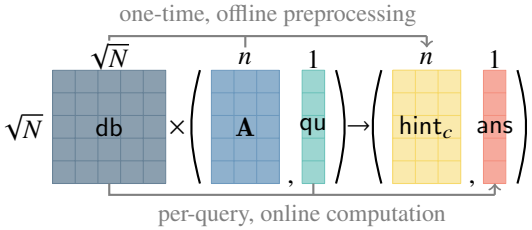


Figure 3: The server computation in SimplePIR. Each cell represents a $\mathbb{Z}_q$ element, and × denotes matrix multiplication. The server performs the bulk of its work—namely, multiplying the database db and the matrix $\mathbf{A}$—in a one-time preprocessing phase. Thereafter, the server can answer each client's query with a lightweight online phase.

homomorphic encryption scheme, and sends this encrypted vector to the server. The server computes the matrix-vector product between the database and the query vector and returns it to the client. Finally, the client decrypts element i_{row} of the server's answer vector—which corresponds exactly to the inner product of database row i_{row} and encrypted unit vector $\mathbf{u}_{i_{\text{col}}}$, or, equivalently, the encrypted database entry at $(i_{\text{row}}, i_{\text{col}})$. In this scheme, the server and the client exchange $2\sqrt{N}$ ciphertext elements, while the server performs N ciphertext multiplications and additions to answer each PIR query.

Our starting point is to instantiate this "square-root" approach with the secret-key version of Regev's LWE-based encryption scheme [76]. Let (n, q, χ) be LWE parameters. Then, the Regev encryption of a vector $\boldsymbol{\mu} \in \mathbb{Z}_p^L$ consists of a pair of a matrix and a vector:

$$\mathsf{Enc}(\boldsymbol{\mu}) = (\mathbf{A}, \mathbf{c}) = (\mathbf{A}, \mathbf{A}\mathbf{s} + \mathbf{e} + \lfloor q/p \rfloor \cdot \boldsymbol{\mu}),$$

for some LWE matrix $\mathbf{A} \xleftarrow{R} \mathbb{Z}_q^{L \times n}$, secret $\mathbf{s} \xleftarrow{R} \mathbb{Z}_q^n$, and error vector $\mathbf{e} \xleftarrow{R} \chi^L$.

We make three crucial observations about Regev encryption:

1. First, a large part of the ciphertext—namely, the matrix $\mathbf{A}$—is independent of the encrypted message. It is thus possible to generate the matrix $\mathbf{A}$ ahead of time.
2. Second, Regev encryption remains secure even when many different parties use the same matrix $\mathbf{A}$ to encrypt many messages [73], provided that each ciphertext uses an independent secret vector $\mathbf{s}$ and error vector $\mathbf{e}$.
3. Finally, we can take $\mathbf{A}$ to be pseudorandom (rather than random) at a negligible loss in security, allowing us to succinctly represent $\mathbf{A}$ by a short random seed.

In SimplePIR, we leverage these three observations as follows. Consider a client who wishes to retrieve the database entry at $(i_{\text{row}}, i_{\text{col}})$. At a conceptual level, the client's query to the server consists of $\mathsf{Enc}(\mathbf{u}_{i_{\text{col}}}) = (\mathbf{A}, \mathbf{c})$—the Regev encryption of the vector in $\mathbb{Z}_p^{\sqrt{N}}$ that is zero everywhere but with a "1" at index i_{col}. The server then represents the database as a matrix $\mathbf{D} \in \mathbb{Z}_p^{\sqrt{N} \times \sqrt{N}}$ and computes and returns the matrix-vector product of the database with the client's encrypted query, i.e., $(\mathbf{D} \cdot \mathbf{A}, \mathbf{D} \cdot \mathbf{c})$. From the server's reply, the client can use standard Regev decryption to recover $\mathbf{D} \cdot \mathbf{u}_{i_{\text{col}}} \in \mathbb{Z}_p^{\sqrt{N}}$, which is exactly the i_{col}-th column of the database, as desired.

Now, we make the following modifications:

1. We have the server compute the value $\mathbf{D} \cdot \mathbf{A}$ ahead of time in a preprocessing phase. This preprocessing step requires $2nN$ operations in $\mathbb{Z}_q$, where $n \approx 2^{10}$ is the dimension of the LWE secret and N is the database size. Then, to answer the client's query, the server needs to compute the value $\mathbf{D} \cdot \mathbf{c}$, which requires only $2N$ operations in $\mathbb{Z}_q$. So, an $n/(n+1)$ fraction (i.e., 99.9%) of the server's work can happen ahead of time—before the client even decides which database record it wants to fetch.
2. We have all clients use the same matrix $\mathbf{A}$ to build their queries. Then, the server must only precompute $\mathbf{D} \cdot \mathbf{A}$ once. The server sends this one-time "hint" to all clients. Thus, the server amortizes the cost of computing and communicating $\mathbf{D} \cdot \mathbf{A}$ over many clients and over many queries.
3. As an optimization, we compress $\mathbf{A}$ using pseudorandomness. In an initialization step before the protocol starts, the server and the clients agree on a public random seed used to generate $\mathbf{A}$. This saves bandwidth and storage, as the server and the clients communicate and store only a small seed.

The security of the SimplePIR construction follows almost immediately from the security of Regev encryption [76] with a reused matrix $\mathbf{A}$ [73], which in turn follows from the hardness of LWE. SimplePIR's correctness follows from the correctness of Regev's linearly homomorphic encryption scheme and of Kushilevitz and Ostrovsky's "square-root" PIR template.

4.2 Parameter selection

Picking the LWE parameters (n, q, χ) and the plaintext modulus p requires a standard (though tedious) analysis. We choose our parameters to have 128-bit security, according to modern lattice-attack-cost estimates [7]. In particular, we set the secret dimension $n = 2^{10}$, use modulus $q = 2^{32}$ (as modern hardware natively supports operations with this modulus), set the error distribution χ to be the discrete Gaussian distribution with standard deviation $\sigma = 6.4$, and allow correctness error $\delta = 2^{-40}$. We obtain the following trade-off between database size N and plaintext modulus p:

Database size N:	2^{26}	2^{28}	2^{30}	2^{34}	2^{38}	2^{42}
Plaintext modulus p:	991	833	701	495	350	247

We discuss parameter selection further in Appendix C.1.

4.3 Extensions

Finally, we extend our SimplePIR construction to meet the requirements of realistic deployment scenarios:

Supporting databases with larger record sizes. The basic SimplePIR scheme (Figure 2) supports a database in which each record is a single $\mathbb{Z}_p$ element—or, roughly 8-10 bits with our parameter settings. Our main application (Section 7) uses a database with one-bit records, though other applications of PIR [6, 9, 10, 48, 66] use much longer records.

To handle large records, we observe that the client in SimplePIR can retrieve an entire column of the database at once. Concretely, after executing a single online phase with the server to query for database element $(i_{\mathsf{row}}, i_{\mathsf{col}})$, the client can run the Recover procedure $\sqrt{N}$ times—once for every row in $[\sqrt{N}]$—to reconstruct the entire database column i_{col}. Thus, to support large records, we encode each record as multiple elements in the plaintext space, $\mathbb{Z}_p$, and store these elements stacked vertically in the same database column. By making a single online query and reconstructing the corresponding column of elements, the client recovers any record of its choosing.

On a database of N records, each in $\mathbb{Z}_p^d$ (where $d \leq N$), with LWE secret dimension n and modulus q, SimplePIR has:

- one-time (hint) download $n \cdot \sqrt{dN}$ elements in $\mathbb{Z}_q$,
- per-query upload and download $\sqrt{dN}$ elements in $\mathbb{Z}_q$, and
- per-query server computation $2dN$ operations in $\mathbb{Z}_q$.

Fetching many database records at once ("Batch PIR"). In many applications [9, 10], a client wants to fetch k records from the PIR server at once. If the client runs our PIR protocol k times on a database of N records, the total server time would be roughly kN. We can apply the "batch PIR" techniques of Ishai et al. [50] to allow a client to fetch k records at server-side cost $\ll kN$, without increasing the hint size.

The idea is to randomly partition the database of N records into k chunks, each represented as a matrix of dimension $(\sqrt{N}/k)$-by-$\sqrt{N}$. To fetch a set of k database records that fall into distinct chunks, the client and server then run SimplePIR once on each database chunk. The hint size (i.e., offline client download) remains $n\sqrt{N}$—as in one-query SimplePIR. The communication cost for the client is $k \cdot \sqrt{N}$—k times larger than in one-query SimplePIR (and identical to the communication if the client fetched all k records sequentially). The server performs N operations in $\mathbb{Z}_q$, as in one-query SimplePIR.

However, more than one of the client's desired records may fall into the same chunk. There are two ways to handle this:

- If the client must recover all k records with overwhelming probability, the client can run the entire batch-PIR protocol λ times to achieve failure probability $2^{-\Omega(\lambda)}$ [9, 50].
- If the client needs only to recover a constant fraction of the k database records, then the client and server run this protocol only once. The server-side computation cost is as in one-query SimplePIR.

Additional improvements. We discuss how to further improve the asymptotic efficiency of SimplePIR in Appendix C.3.

4.4 Fast linearly homomorphic encryption

In Appendix D, we introduce the notion of *linearly homomorphic encryption with preprocessing*. This new primitive abstracts out the key properties of Regev encryption that we use in SimplePIR. We expect this new form of linearly homomorphic encryption to have further practical applications.

5 DoublePIR

While SimplePIR has high server-side throughput, it requires the client to download and store a relatively large preprocessed hint, of size roughly $n\sqrt{N}$ on lattice dimension $n \approx 2^{10}$ and database size N. In this section, we present DoublePIR, a new PIR scheme that recursively applies SimplePIR to reduce the hint size to roughly n^2 on lattice dimension n—independent of the database size—while maintaining a server-side throughput upwards of 5.2 GB/s. (In practice, this hint size is 16 MB for one-byte records.) For databases of very many records ($N \gg n^2 \approx 2^{20}$), DoublePIR has a much smaller hint size than SimplePIR. As in SimplePIR, the per-query communication cost for DoublePIR is $O(\sqrt{N})$ on database size N.

5.1 Construction

We present a formal description of DoublePIR in Figure 13 of Appendix E, along with a full correctness and security analysis. In this section, we describe the key design ideas.

We first give the concrete costs of DoublePIR on database size N, lattice dimension n, LWE modulus q, plaintext modulus p, and $\kappa = \lceil \log(q)/\log(p) \rceil \approx 4$ (chosen as in Appendix E.1). In a one-time public preprocessing phase, DoublePIR requires

1. the server to perform $2nN + 2\kappa n^2\sqrt{N}$ operations in $\mathbb{Z}_q$, and
2. the client to download κn^2 elements in $\mathbb{Z}_q$.

On each query, DoublePIR requires

1. the client to upload $2\sqrt{N}$ elements in $\mathbb{Z}_q$,
2. the server to do $2N + 2(2n+1) \cdot \sqrt{N} \cdot \kappa$ $\mathbb{Z}_q$ operations, and
3. the client to download $(2n+1) \cdot \kappa$ elements in $\mathbb{Z}_q$.

At a high level, DoublePIR first executes exactly as SimplePIR: from the database, the server computes a hint matrix and, in response to each client's query, produces an answer vector. At this point, we observe that a client querying for element $(i_{\mathsf{row}}, i_{\mathsf{col}})$ in SimplePIR needs two pieces of information to recover its desired database element:

- row i_{row} of the hint matrix $\mathbf{D} \cdot \mathbf{A} \in \mathbb{Z}_q^{\sqrt{N} \times n}$, and
- element i_{row} of the answer vector $\mathbf{a} \in \mathbb{Z}_q^{\sqrt{N}}$.

Thus, in DoublePIR, we have the client execute a second level of SimplePIR over the hint matrix and the answer vector to retrieve these $(n+1)$ values. As such, the client in DoublePIR recovers the database entry at $(i_{\mathsf{row}}, i_{\mathsf{col}})$ without downloading the large first-level hint.

Kushilevitz and Ostrovsky [53] first proposed using recursion to reduce communication costs in PIR in this way. However, applied naïvely, this strategy requires $(n+1) \approx 2^{10}$ instances of PIR to recover the $(n+1)$ desired values. We avoid this bottleneck with the insight that SimplePIR lets the client retrieve a *column* of the database at a time (as discussed in Section 4.3). Therefore, in DoublePIR, we run the second level of PIR over the database corresponding to the *transpose* of the hint matrix concatenated with the answer vector (i.e.,

$\left[\mathbf{D} \cdot \mathbf{A} \;||\; \mathbf{a}\right]^T$). Using a single invocation of SimplePIR, the client in DoublePIR can retrieve column i_{row} of this database—which holds exactly row i_{row} of the hint matrix and element i_{row} of the answer vector—and finally recover the database entry at $(i_{\mathsf{row}}, i_{\mathsf{col}})$. As SimplePIR executes over a database of elements in $\mathbb{Z}_p$, while the hint matrix and the answer vector consist of elements in $\mathbb{Z}_q$, the server in DoublePIR computes the base-p decomposition of the entries in the hint matrix and the answer vector before performing the second level of PIR.

Since this second level of PIR operates on a much smaller database, its cost is dwarfed by that of the first level of PIR: in DoublePIR, both the online communication and the server throughput remain roughly the same as in SimplePIR. Moreover, as the client in DoublePIR forgoes downloading the large first-level hint, it now only downloads a much smaller hint, whose size is independent of the database length, produced by the second level of PIR. Concretely, our PIR client downloads a 16 MB hint in the offline phase.

Remark 5.1 (Why not recurse more?). DoublePIR performs two levels of PIR queries to reduce the total communication cost. A natural question is whether additional levels of recursion can help, as in standard single-server PIR schemes [53]. After r levels of recursion, the communication cost of the recursive PIR scheme, on lattice dimension n and database size N, would be (hiding constants):

- one-time download n^r in the preprocessing step,
- per-query upload $r \cdot N^{1/r}$, and
- per-query download n^{r-1}.

For $r > 2$, the communication costs here are likely too large for database sizes of interest. An intriguing open question is whether it is possible to construct recursive LWE-based PIR schemes with total communication cost $n \cdot N^{1/r}$.

5.2 Extensions

Finally, we extend DoublePIR to handle diverse deployment scenarios.

Handling large database records. To handle databases with large records, we represent each record as a series of elements in $\mathbb{Z}_p$, where p is the plaintext modulus, using base-p decomposition. Let d denote the number of $\mathbb{Z}_p$ elements that each record maps to. Then, on each execution of DoublePIR, we run the PIR scheme d times in parallel, over d databases, where the i-th database holds the i-th $\mathbb{Z}_p$ element of each record. With this approach, DoublePIR's throughput is identical on databases with long records and with short records. On a database of N records, each in $\mathbb{Z}_p^d$, with LWE secret dimension n, modulus size q, and $\kappa = \lceil \log(q)/\log(p) \rceil \approx 4$, DoublePIR has:

- hint size $d\kappa n^2$ elements in $\mathbb{Z}_q$,
- online upload $2\sqrt{N}$ elements in $\mathbb{Z}_q$,
- online server work $2d\left(N + \kappa(2n+1)\sqrt{N}\right)$ ops. in $\mathbb{Z}_q$, and
- online download $d\kappa \cdot (2n+1)$ elements in $\mathbb{Z}_q$.

Batching client queries. To implement query batching in DoublePIR, we batch queries exactly as in SimplePIR (cf. Section 4.3) when performing the first level of PIR. As DoublePIR makes non-black-box use of SimplePIR in performing the second level of PIR, we are not able to derive any computation savings from batching in this second, recursive step. (In particular, in the second level of PIR, the client must read an entire column consisting of $(n+1)$ elements at once for each query; this breaks SimplePIR's batching trick as many elements fall into the same chunk.) However, as the first level of PIR dominates the computation in DoublePIR, batching many queries nevertheless greatly improves DoublePIR's throughput.

Concretely, to fetch a set of k records from a database of N elements in $\mathbb{Z}_p$, on lattice dimension n, LWE modulus q, and $\kappa = \lceil \log(q)/\log(p) \rceil \approx 4$, DoublePIR has:

- hint size κn^2 elements in $\mathbb{Z}_q$,
- online upload $\sqrt{N}(k + \sqrt{k})$ elements in $\mathbb{Z}_q$,
- online server work $2N + 2k(2n+1)\kappa\sqrt{N}$ ops. in $\mathbb{Z}_q$, and
- online download $k\kappa(2n+1)$ elements in $\mathbb{Z}_q$.

6 Data structure for private approximate set membership

In this section, we introduce a new data structure for the *approximate private set membership* problem. In this problem, a client holds a private string σ, a server holds a set of strings S, and the client wants to test whether σ is included in S without revealing σ to the server. To rule out the trivial solution where the server sends S to the client, we insist on communication sublinear in $|S|$. This problem differs from that of private set intersection [41] in two key ways: the server's set S does not need to be kept private, and the client's output bit b (indicating whether $\sigma \in S$) may be approximate, i.e., there is some chance that it will be incorrect. Looking ahead, our data structure will be at the core of our new scheme for auditing in Certificate Transparency (Section 7).

At a very high level, we have the server preprocess its set of strings S into a data structure. Then, the client, holding a private string σ, can test whether $\sigma \in S$ by privately reading only a few bits of the server's data structure. The client fetches these bits from the server using PIR. Thus, the relevant cost metrics in this setting (Table 4) are:

- *Number of probes.* How many bits/symbols of the server's data structure must the client read?
- *PIR database size.* What is the size of the data structure from which the client performs its private PIR read?
- *Soundness error (i.e., false-positive rate).* For an adversarially chosen set S and string σ, what is the probability, *only over the choice of the client's randomness*, that the client outputs "$\sigma \in S$" when $\sigma \notin S$.

Remark 6.1 (False positives versus false negatives). Some, but not all, applications can tolerate false positives. For example,

	Probes	*PIR size*	*Soundness error*
PIR by keys [25]	2	$3\lambda N$	$\lvert\mathcal{U}\rvert \cdot 2^{-\lambda}$
Bloom filter [14]	λ	$2\lambda N$	$\lvert\mathcal{U}\rvert \cdot 2^{-\lambda}$
Bloom filter [14]	1	$2N$	1 (i.e., not sound)
This work	1	$8N$	1/2

Table 4: Private set membership data structures, for sets of N elements from universe $\mathcal{U}$, on security parameter λ. The soundness error for a one-query Bloom filter is 1—i.e., is insecure.

password-breach database lookups [57,72,80] could potentially tolerate false-positive rates as large as 2^{-30}; in contrast, Safe Browsing blocklist lookups [47, 52] demand a cryptographically negligible false-positive rate, as false positives would cause a legitimate website to be flagged as malicious. For the latter case, other data structures may be more appropriate.

6.1 Related approaches

Standard Bloom filters [14] are an excellent solution to this problem, except for a major wrinkle: the soundness guarantee they provide is too weak for our purposes. In particular, the soundness guarantee of standard Bloom filters only holds when the string σ being tested is chosen *independently* of the randomness used to construct the filter (i.e., the hash function). In our adversarial setting (Section 7), the data structure (including its hash function(s)) is public, and an adversary could choose the string σ in a way that depends on the hash function(s) used to construct the Bloom filter. Standard Bloom filters do not provide good soundness in this setting, unless the number of probes is relatively large (Table 4).

To address this issue, prior work has constructed adversarial Bloom filters [27, 68]. While these data structures provide adversarial soundness as we require, they do not provide privacy. In particular, they require the client to send its string σ to the server; the server then performs a private computation on σ to determine the probes.

In contrast, we show how the server can build a data structure such that the client can perform an approximate set-membership test by reading a single bit from a roughly λ-fold smaller database than would be required by Bloom filters or other standard techniques (on security parameter $\lambda \approx 128$). If the client reads this bit using PIR, this approach is private. The downside is that our data structure's soundness-error rate is constant (i.e., 1/2) instead of negligible in λ. However, since our application (Section 7) can naturally tolerate constant soundness error, this trade-off is profitable.

6.2 Syntax and properties

We define the syntax and security properties for our approximate private set membership data structure. The data structure is parameterized by a universe $\mathcal{U} \subseteq \{0,1\}^*$ of possible strings and consists of two routines:

$\mathsf{Setup}(S) \to D$. Take as input a set of strings $S \subseteq \mathcal{U}$ and output a data structure $D \in \{0,1\}^{a \times b}$, where $a, b \in \mathbb{N}$ can depend on $|S|$. This algorithm is deterministic.

$\mathsf{Query}(\sigma) \to (i, j)$. Given a candidate string $\sigma \in \mathcal{U}$, output an index $(i, j) \in [a] \times [b]$ in the data structure D to probe. If $\sigma \in S$, it holds that $D_{i,j} = 1$. This algorithm is randomized.

Properties. For a set $S \subseteq \mathcal{U}$ and string $\sigma \in \mathcal{U}$, define

$$\mathsf{Accept}(S, \sigma) := \Pr\left[D_{i,j} = 1 : \begin{array}{r} D \leftarrow \mathsf{Setup}(S) \\ (i,j) \leftarrow \mathsf{Query}(\sigma) \end{array}\right]$$

where the probability is taken *only* over the randomness of the Query algorithm.

Completeness. An approximate membership test is *complete* if, for all $S \subseteq \mathcal{U}$ and $\sigma \in S$, $\mathsf{Accept}(S, \sigma) = 1$.

Soundness. An approximate membership test has *soundness error* ϵ if, for all $S \subseteq \mathcal{U}$ and $\sigma \notin S$, $\mathsf{Accept}(S, \sigma) \leq \epsilon$.

Privacy. An approximate membership test is *private* if the row of the data structure that the Query algorithm probes reveals no information about the query string σ. In particular, we say that privacy holds if, for all strings $\sigma_0, \sigma_1 \in \mathcal{U}$, the following distributions of the row indices are identical:

$$\{i : (i, \_) \leftarrow \mathsf{Query}(\sigma_0)\} \equiv \{i : (i, \_) \leftarrow \mathsf{Query}(\sigma_1)\}.$$

In a deployment, to probe the element at row i and column j in the data structure, the client sends i in the clear, along with a PIR query for j, to the server. The server executes the PIR scheme over the i-th row, and sends the PIR answer (and, if needed, the client hint for that row) back to the client. Thus, by the above privacy property, the server learns nothing about σ. Performing PIR over only a row of the data structure (rather than the whole data structure) will be the source of our λ-fold performance improvements compared to related approaches.

6.3 Our approximate membership test

Our construction is parameterized by integers $a, k \in \mathbb{N}$ and a set size N. Our data structure then maps the set of strings S into a matrix of a-by-(kN) bits. The construction uses cryptographic hash functions $H_1, \ldots, H_a : \mathcal{U} \to [kN]$, which we model as independent random oracles [13].

$\mathsf{Setup}(S) \to D \in \{0,1\}^{a \times kN}$:

- Let $D \in \{0,1\}^{a \times kN}$ be the all-zeros matrix.
- For each $i \in [a]$ and $\sigma \in S$, set $D_{i, H_i(\sigma)} = 1$.
- Return D.

$\mathsf{Query}(\sigma) \to (i, j) \in [a] \times [kN]$:

- Choose $i \xleftarrow{R} [a]$, and output $(i, H_i(\sigma))$.

We then prove the following proposition in Appendix F:

Proposition 6.2. *The approximate membership-test data structure* (Setup, Query)*, on parameters* $a \geq 2\,(\log(|\mathcal{U}|) + \lambda)$ *and*

$k \geq 8$, is complete, private, and has soundness error $1/2$. The construction fails with probability $2^{-\lambda}$, over the choice of the hash functions modeled as independent random oracles. Concretely, on $|\mathcal{U}| = 2^{256}$, taking $k = 8$ and $a = 768$ gives soundness error $1/2$ and failure probability 2^{-128}.

7 Application: Auditing in Certificate Transparency

We now explain how to use our new PIR schemes to solve the problem of privately auditing signed certificate timestamps in deployments of Certificate Transparency [54, 55, 65].

7.1 Problem statement

Background: Certificate Transparency. The goal of Certificate Transparency is to store every public-key certificate that every certificate authority issues in a set of publicly accessible logs. To this end, certificate authorities submit the certificates they issue to log operators, who respond with a *signed certificate timestamp* (SCT). The SCT is a promise to include the new certificate in the log maintained by this operator within some bounded period of time.

Later on, when a TLS server sends a public-key certificate to a client, the server attaches a number of SCTs according to the client's policy (e.g., Google and Apple both require SCTs from three distinct log operators in Chrome and Safari, respectively). By verifying the SCTs, the client can be sure that each of the log operators has seen the new certificate and—if the operator is honest—will eventually log it. Domain operators can then use the logs to detect whether or not a certificate authority has mistakenly or maliciously issued a certificate for their domain.

SCT auditing. To keep the logs honest, some party in the system must also verify that log operators are fulfilling the promise implicit in the SCTs that they issue. In particular, if a client receives an SCT for some certificate C signed by a log operator, this party would like to verify that C actually appears in that operator's public log. This process is *SCT auditing*.

Clients must be involved in SCT auditing, as they are the only participants who see SCTs "in the wild." However, since the set of SCTs that a client sees reveals information about the client's browsing history, the client should not reveal which SCTs it has seen to the log operators or any other entity.

Google's recent proposals for SCT auditing [33, 78] involve an SCT *auditor* (run by Google) that is separate from the client. In their model, the auditor maintains the entire set of SCTs for non-expired certificates from all Certificate Transparency logs. Every SCT that a client sees for a live website should appear in the auditor's set. To determine whether an SCT is valid, a client can check whether it (or really, its SHA256 hash) appears in the set of valid SCTs maintained by the auditor:

- If the client's SCT appears in the auditor's set, then the log server that issued the SCT correctly fulfilled its promise.
- If not, the client can report the problematic SCT to the auditor (i.e., Google) to investigate further.

To protect the client's privacy, the client should not reveal its SCT to the auditor directly. (Again, the fact that a client has seen an SCT for `example.com` reveals that the client has visited `example.com`.) A privacy-protecting solution for SCT auditing must thus allow the client to test whether or not its SCT appears in the auditor's set of valid SCTs, without revealing its SCT to the auditor. This is a private set-membership problem [80].

To summarize, we need the following properties to hold, which we state only informally:

- **Correctness with false negatives.** When an honest client holding string σ, chosen independently of the client's secret randomness, interacts with an honest audit server holding strings $S = \{\sigma_1, \sigma_2, \dots\}$:
  - if $\sigma \in S$, then the client always outputs "valid", and
  - if $\sigma \notin S$, then the client outputs "invalid" with probability at least $1/2$, over the choice of the client's randomness.
- **Privacy for the client.** When an honest client interacts with a malicious audit server, the server learns nothing about the client's private input string σ.

We do not require correctness to hold against a malicious audit server. (If the audit server wants to break correctness, it can trivially do so by lying about its set of SCTs.)

System parameters. There are roughly five billion active SCTs in the web today [33]. Roughly six million of these are added or removed each day as certificate authorities issue certificates and as certificates expire [2]. Google's current proposal for SCT auditing requires the Chrome client to randomly sample 0.1% of the SCTs associated with its TLS connections, and to audit only this small fraction of all SCTs [33]. Random sampling in this way reduces the amortized cost of auditing by $1000\times$, but also reduces the chance that a single auditing client catches a cheating log. Still, across many auditing clients this randomized SCT auditing will—with high probability—catch invalid SCTs that many clients see. (After 1000 clients observe an invalid SCT, one will audit it in expectation.)

7.2 Existing approaches to SCT auditing

There are two existing proposals we consider for SCT auditing, which could be used separately or combined.

Opt-out SCT auditing. Google Chrome's current proposal [33] has the client reveal the first 20 bits of the hash of its SCT to the audit server. The audit server replies with all ≈ 1000 SCTs in its set that begin with this 20-bit prefix. This method is simple to implement, but achieves only k-anonymity for $k = 1000$; i.e., Google's audit server learns that the client visited one in a set of 1000 websites.

Anonymizing proxy. The client could use one or multiple proxy servers, such as in Tor [34], to send its SCT to the auditor anonymously [31], and the auditor could reply with the bit

indicating whether or not the SCT appears in its set. This mechanism is susceptible to timing attacks and it reveals the entire distribution of clients' SCTs to the audit server, which could allow the auditor to deanonymize particular clients. As a practical matter, browser vendors may not want to outsource this security-critical task to the Tor network.

7.3 Our approach

We propose a new scheme for SCT auditing that uses our PIR schemes (Sections 4 and 5) along with our new approximate set-membership data structure (Section 6). As in Chrome's proposal for SCT auditing [33], we would expect to run this protocol on a small fraction of the SCTs sampled from the client's TLS connections. The deployment is as follows:

1. Audit server: Data-set construction. The SCT audit server prepares an approximate set-membership data structure for the set of all SHA256 hashes of all N active SCTs. This data structure consists of $a \approx 768$ arrays, each $8N$ bits in length. The server runs our PIR scheme's Setup algorithm on each of these a arrays, to produce a PIR hints.

2. Client: Hint download. The client chooses a secret random value $i^* \xleftarrow{R} [a]$ and downloads the i^*-th hint from the audit server (revealing i^* to the server in the process). To test whether a particular SCT hash appears in the server's set, the client now only needs to privately read a single bit from the i^*-th array that the server holds. A malicious TLS server can trick the client into accepting an invalid SCT with probability at most $\epsilon \approx 1/2$. (The value ϵ is a parameter of the underlying approximate set-membership data structure.) If the client audits an f-fraction of all TLS connections, the false-negative rate is $(1 - f) + f \cdot \epsilon$. In our deployment, $f = 1/500$ and $\epsilon = 1/2$, which yields an overall false-negative rate of $1/1000$, matching that of Chrome's current proposal [33].

3.Client and audit server: SCT lookup via PIR. Once the client receives an SCT whose validity it wants to verify, the client hashes the SCT, computes the bit of the server's i^*-th array that the client needs to read to verify the validity of the SCT, and uses our PIR protocol, along with its offline hint, to read this bit. The audit server answers the client's PIR query with respect to the i^*-th array, and the client reconstructs the bit.

7.4 How we handle database updates

The client holds a PIR "hint" that depends on the contents of the audit server's database. Whenever the set of active SCTs changes—which happens continuously as certificate authorities issue new certificates and certificates expire—the audit server must update its database state. Without extra engineering, the client would have to download a fresh hint from the audit server. Since the hints are relatively large (tens of megabytes for our parameter settings), we would like the client to have to download a hint only rarely.

Our approach is to have the client only download a hint periodically—once per week, for example. When the audit server sends the hint to the client, the server specifies the range of *certificate issue dates* that the hint covers. When the client wants to audit an SCT, the client checks whether its current hint covers the issue date of the SCT in question. If so, the client makes a PIR query to test the validity of the SCT. If not, the client caches the hash of the SCT so that it can test the validity of this SCT the next time it downloads a hint from the audit server. In this way, the client eventually audits its full random sample of SCTs, but the client needs only to download a hint from the audit server periodically.

The server must now store a few copies of the database (e.g., one per day) but server storage is relatively inexpensive. In a large-scale deployment, it would be readily possible to keep all of these database copies in memory by having one or more physical servers responsible for each version of the database.

8 Evaluation

Implementation. We implement SimplePIR in 1,000 lines of Go code, along with 80 lines of C (for the matrix-multiplication routines). Implementing DoublePIR requires 170 additional lines of Go code. Our code does not rely on any external libraries and is published under the MIT open-source license [1]. A code listing for the core of SimplePIR appears in Appendix G.

We store the database in memory in packed form and decompress it into $\mathbb{Z}_p$ elements on-the-fly, as otherwise the Answer routine is memory-bandwidth-bound. In DoublePIR, we represent the database as a rectangular (rather than square) matrix, so that the first level of PIR dominates the computation.

We run all experiments using a single thread of execution, on an AWS `c5n.metal` instance running Ubuntu 22.04. To collect the throughput numbers for tables, we run each scheme five times and report the average. All standard deviations in throughput are smaller than 5% of the throughput measured.

8.1 Microbenchmarks

Throughput. We first measure the maximal throughput of each PIR scheme, on the database dimensions that suit it best. In Table 1, we report the throughput measured for each PIR scheme, on a database consisting of roughly 2^{33} bits (or, 1 GB), where we take the entry size to be that for which the highest throughput was reported in the corresponding paper (or in a related paper, if it is not made explicit). These entry sizes appear in Table 9. We confirm that these throughputs are indeed the best achievable for each scheme by measuring each scheme's throughput on each entry size in Figure 6.

SimplePIR and DoublePIR achieve throughputs of 6.5 GB/s and 5.2 GB/s respectively, which is roughly 5× better than the fastest prior single-server PIR scheme designed for the streaming setting (SpiralStreamPack) and 18× better than the fastest prior single-server PIR scheme designed for databases

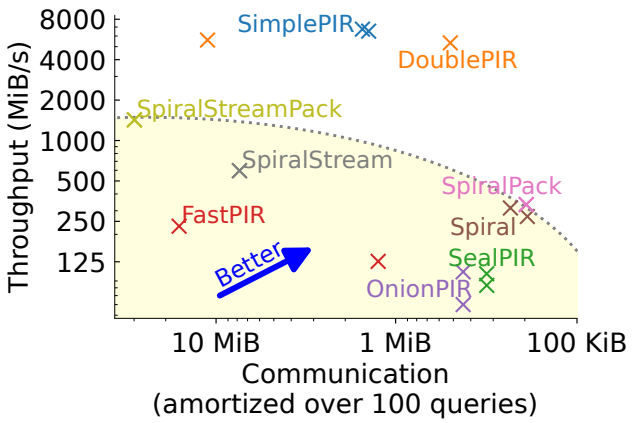


Figure 5: Throughput vs. per-query communication, on a database of total size 2^{33} bits (or, 1 GiB). For each PIR scheme, we display the communication and the corresponding throughput for two choices of entry size: one that maximizes the throughput, and another that minimizes the communication. (For schemes with only one point displayed, both entry sizes are the same.) The per-query communication cost given is the total (i.e., offline and online) communication, amortized over 100 queries.

with short entries (Spiral). SimplePIR and DoublePIR also achieve higher *per-server* throughput[1] than prior two-server PIR schemes: two-server PIR from DPFs achieves a per-server throughput of 5.4 GB/s. Finally, we benchmark the throughput of performing XORs in a linear scan over the database to provide a hard upper bound on the performance of linear-work, two-server PIR. This benchmark achieves a per-server throughput of 6.1 GB/s, also slower than SimplePIR.

Communication. In Figure 6, we give each scheme's total communication, amortized over 100 queries, for increasing entry sizes. On databases with short entries, DoublePIR's amortized communication is comparable to that of the most communication-efficient schemes (Spiral, SpiralPack, SealPIR, and OnionPIR). With larger entries, DoublePIR's ammortized communication costs increase, as the client must download many hints. The two schemes with the closest throughput to ours (SpiralStream and SpiralStreamPack), as well as FastPIR, have much larger amortized communication than both DoublePIR and SimplePIR on entry sizes less than a kilobit.

Throughput vs. communication trade-off. We summarize these findings in Figure 5, which displays the throughput/communication trade-off achieved by each PIR scheme. Concretely, we run each scheme on a database of 2^{33} bits with increasing entry sizes (as also done in Figure 6). Then, for each scheme, we display the per-query communication (amortized over 100 queries) and the corresponding throughput for two choices of the entry size: one that maximizes the throughput, and another that minimizes the communication. Figure 5 demonstrates that our new PIR schemes achieve a novel point in the design space: SimplePIR and DoublePIR have substantially higher throughput than all prior single-server PIR schemes; DoublePIR further has a per-query communication cost that is competitive with the most communication-efficient schemes.

[1]In computing the *per-server* throughput of two-server PIR (from DPFs and from XOR), we divide the measured throughput by two.

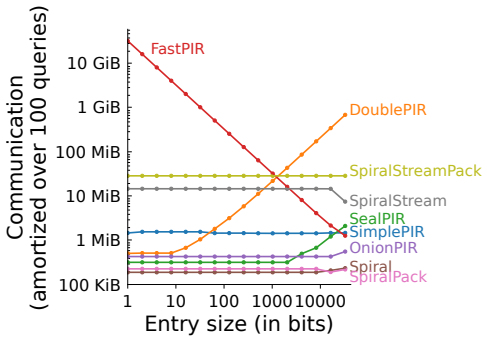
	Communication				
	Offline (MiB)		Online (KiB)		**Throughput**
	Up.†	Down.*	Up.	Down.	(MiB/s)
SealPIR	5	0	91	181	97
FastPIR	0.06	0	33 000	64	217
OnionPIR	5	0	256	128	60
Spiral	15	0	14	20	259
SpiralPack	19	0	14	20	260
SpiralStream	0.34	0	15 000	20	485
SpiralStreamPack	15	0	29 000	99	1 370
SimplePIR (§4)	0	124	121	121	6 496
DoublePIR (§5)	0	16	313	32	5 217

Table 7: PIR scheme performance a on 1 GiB database, consisting of $2^{33}\times$ 1-bit entries. We highlight in green cells that are the best, or within 5× of the best, and in red cells that are the worst, or within 5× of the worst, in their respective columns. (We leave uncolored cells that are within 5× of the best and worst.) For schemes that do not have an automatic parameter selection tool (SealPIR, FastPIR, and OnionPIR), we automatically "re-balance" the scheme by executing it on a database of $2^{33}/d$ entries, each of size d, where d is the closest valid power-of-2 to the scheme's "optimal" entry size (displayed in Table 9). †The offline upload is equal to the server-side, per-client storage. *The offline download is equal to the client-side storage.

Comparison on a database of $2^{33}\times 1$-bit entries. In Table 7, we give a fine-grained comparison of the performance of each scheme on a database relevant to our application, consisting of 2^{33} 1-bit entries. On this database, SimplePIR and DoublePIR again achieve much higher throughput than all other schemes (6 GB/s and 5 GB/s respectively). SimplePIR has high offline download and thus also client-side storage costs. However, DoublePIR's offline download is comparable to the offline communication of other PIR schemes, and its online communication is on the order of kilobytes.

For each scheme, we additionally compute its cost per query, when the client makes 100 database queries, using the AWS costs for compute ($\$1.5\cdot 10^{-5}$ per core second) and data transfer out of Amazon EC2 ($0.09 per GB). SimplePIR achieves a per-query cost of $\$1\cdot 10^{-4}$, while DoublePIR and the cheapest scheme from related work (SpiralStreamPack) each achieve a per-query cost of $\$2\cdot 10^{-5}$. We note, however, that SpiralStreamPack has a very large online upload (on the order of megabytes), which is not reflected in its per-query cost, as AWS only charges for outgoing communication.

Batching queries. Finally, we evaluate how SimplePIR and DoublePIR's effective throughput scales when the client makes a batch of queries for k records at once, assuming the client only needs to recover a constant fraction of the k records. For increasing values of k, we compute the expected number of "successful" queries (i.e., the expected number of queries that



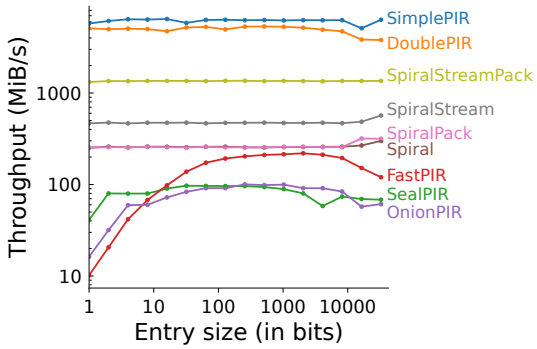


Figure 6: Throughput and per-query communication for each PIR scheme, on a database of total size 2^{33} bits (or, 1 GB), with entries of increasing size. The per-query communication cost given is the total (i.e., offline and online) per-query communication, amortized over 100 queries.

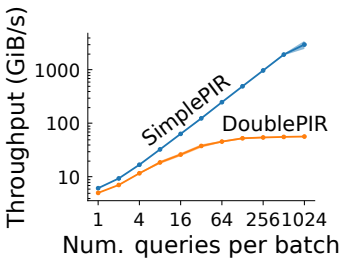


Figure 8: Effective PIR throughput (database size × queries per second), with increasing batch sizes and a fixed-size hint, on a database consisting of $2^{33} \times 1$-bit entries. The shading displays the standard deviation.

fall into a distinct database chunk, as discussed in Section 4.3) and we derive the expected "successful" throughput, i.e. the throughput measured when the server answers that number of queries at once, with a single pass over the database.

Figure 8 shows that both SimplePIR and DoublePIR's throughput increases when the client makes a batch of queries at once. SimplePIR's throughput scales linearly, achieving a value of over 100 GB/s on batch size $k \geq 32$ and roughly 1000 GB/s on batch size $k \geq 256$. DoublePIR achieves a throughput of roughly 50 GB/s for $k \geq 64$; at this point, the throughput plateaus, as the second level of PIR becomes a bottleneck.

8.2 Certificate Transparency benchmark

We propose using DoublePIR for the SCT auditing application. With our new data structure for private set membership, the task of SCT auditing requires a single round of PIR over a database with 1-bit entries. For such a database, our evaluation in Section 8.1 shows that DoublePIR achieves both high server throughput and small client storage and communication. SCT auditing occurs in the background, and is not on the critical path to web browsing. Thus, while using PIR may increase the latency of auditing, we believe this is a desirable trade-off in exchange for cryptographic privacy, as long as the computation remains modest and the communication remains comparable.

To benchmark DoublePIR in this application context, we evaluate the scheme on a database consisting of $2^{36} \times 1$-bit entries, which is the size of a row in our approximate set membership data structure (Proposition 6.2) when we instantiate it with all 5 billion active SCTs. (In our evaluation, each entry is a random bit.) On this database size, we measure that DoublePIR has a "hint" of size 16 MB, an online upload of 724 KB, and an online download of 32 KB. The server can answer each query in fewer than 1.9 core-seconds (and this work is fully parallelizable). As our client must audit one in every 500 TLS connections, our proposal for SCT auditing then requires: (1) 16 MB of client storage and download every week (to keep the client hint), and (2) per TLS connection, an amortized overhead of 0.004 core-seconds of server compute and 1.5 KB of communication. Using the AWS costs for compute ($\$1.5 \cdot 10^{-5}$ per core second) and data transfer ($0.09 per outgoing GB), for each client, this amounts to a fixed cost of $0.001 per week, along with $\$6 \cdot 10^{-8}$ per TLS connection. For a typical client making 10^4 TLS connections per week [3], we expect this cost to be roughly $0.1 per year. Since (after sampling its TLS connections) a typical client makes only roughly 20 queries to the audit server using each week's hint, we can reduce the client's storage to less than 400 KB using the optimization from Appendix E.3.

By comparison, Google Chrome's current SCT auditing scheme has the client audit one in every 1000 TLS connections and provides only k-anonymity (for $k = 1000$) [33]. That is, the server learns that a client visited one of a set of 1000 domains. Auditing incurs an amortized overhead of 240 B of communication per connection [33], negligible server-side computation, and no client-side storage (unless the client caches popular SCTs). Again assuming a client making 10^4 TLS connections per week [3], we expect this scheme to cost roughly $0.01/client/year; our approach using DoublePIR incurs only 13× more communication and achieves the goal of cryptographic privacy.

9 Conclusion

We show that the server-side, per-core throughput of single-server PIR can approach the memory bandwidth of the machine. One of our new schemes, SimplePIR, is faster than two-server PIR without two-server PIR's undesirable trust assumptions. Two exciting directions remain open: one is to reduce our schemes' communication; and another is to explore combining our ideas with those of *sublinear*-time PIR [29, 30] to reduce the computation cost beyond the linear-server-time barrier.

Acknowledgements. We thank Martin Albrecht for answering questions about LWE hardness estimates, Vadim Lyubashevsky for advice on discrete gaussian sampling, and Adam Belay and Zhenyuan Ruan for discussions about AVX performance. We are grateful to Anish Athalye and Derek Leung for reviewing a draft of this work, and to Dima Kogan, David Wu, Jean-Philippe Bossuat, and Samir Menon for helpful conversations and feedback. We thank Sebastian Angel for constructive comments on the discussion of malicious security in an earlier version of this work, and for suggestions on how to improve the presentation. This work was supported in part by the National Science Foundation (Award CNS-2054869), a gift from Google, a Facebook Research Award, and MIT's Fintech@CSAIL Initiative. Alexandra Henzinger was supported by the National Science Foundation Graduate Research Fellowship under Grant No. 2141064 and an EECS Great Educators Fellowship. Matthew M. Hong was funded by NIH R01 HG010959. Vinod Vaikuntanathan was supported by DARPA under Agreement No. HR00112020023, NSF CNS-2154149, MIT-IBM Watson AI, Analog Devices, a Microsoft Trustworthy AI grant and a Thornton Family Faculty Research Innovation Fellowship. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA.

References

[1] Code for SimplePIR and DoublePIR. `https://github.com/ahenzinger/simplepir`.

[2] Merkle town. `https://merkle.town/`.

[3] Mozilla Telemetry Portal, Measurement Dashboard. `https://telemetry.mozilla.org/new-pipeline/dist.html#!cumulative=0&end_date=2022-07-17&include_spill=0&keys=__none__!__none__!__none__&max_channel_version=nightly%252F104&measure=HTTP_TRANSACTION_IS_SSL&min_channel_version=nightly%252F104&processType=*&product=Firefox&sanitize=1&sort_by_value=0&sort_keys=submissions&start_date=2022-06-27&table=1&trim=1&use_submission_date=0`. Accessed 19 July 2022.

[4] Might I Get Pwned: A second generation compromised credential checking service. In *USENIX Security Symposium*, 2022.

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

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

[7] Martin Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. In *Journal of Mathematical Cryptology*, 2015.

[8] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication–Computation trade-offs in PIR. In *USENIX Security Symposium*, 2021.

[9] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. PIR with compressed queries and amortized query processing. In *Symposium on Security and Privacy*, 2018.

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

[11] W. Banaszczyk. Inequalities for convex bodies and polar reciprocal lattices in $\mathbf{R}_n$. *Discrete & computational geometry*, 1995.

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

[13] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS*, 1993.

[14] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970.

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

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

[17] Elette Boyle, Lisa Kohl, and Peter Scholl. Homomorphic secret sharing from lattices without fhe. In *EUROCRYPT*, 2019.

[18] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *CRYPTO*, 2012.

[19] Zvika Brakerski, Nico Döttling, Sanjam Garg, and Giulio Malavolta. Leveraging linear decryption: Rate-1 fully-homomorphic encryption and time-lock puzzles. In *TCC*, 2019.

[20] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *EUROCRYPT*, 1999.

[21] Ran Canetti, Justin Holmgren, and Silas Richelson. Towards doubly efficient private information retrieval. In *TCC*, 2017.

[22] Yan-Cheng Chang. Single database private information retrieval with logarithmic communication. In *ACISP*, 2004.

[23] Melissa Chase, Sanjam Garg, Mohammad Hajiabadi, Jialin Li, and Peihan Miao. Amortizing rate-1 OT and applications to PIR and PSI. In *TCC*, 2021.

[24] Massimo Chenal and Qiang Tang. On key recovery attacks against existing somewhat homomorphic encryption schemes. Cryptology ePrint Archive, Report 2014/535, 2014. `https://ia.cr/2014/535`.

[25] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. Cryptology ePrint Archive, Report 1998/003, 1998. `https://ia.cr/1998/003`.

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

[27] David Clayton, Christopher Patton, and Thomas Shrimpton. Probabilistic data structures in adversarial environments. In *CCS*, 2019.

[28] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *STOC*, 1987.

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

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

[31] Rasmus Dahlberg, Tobias Pulls, Tom Ritter, and Paul Syverson. Privacy-preserving and incrementally-deployable support for Certificate Transparency in Tor. *PoPETS*, 2021.

[32] Ivan Damgård and Mads Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In *PKC*, 2001.

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

[34] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation onion router. In *USENIX Security Symposium*, 2004.

[35] Yevgeniy Dodis, Shai Halevi, Ron D. Rothblum, and Daniel Wichs. Spooky encryption and its applications. In *CRYPTO*, 2016.

[36] Nico Döttling, Sanjam Garg, Yuval Ishai, Giulio Malavolta, Tamer Mour, and Rafail Ostrovsky. Trapdoor hash functions and their applications. In *CRYPTO*, 2019.

[37] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 1985.

[38] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2012.

[39] Prastudy Fauzi, Martha Norberg Hovd, and Håvard Raddum. A practical adaptive key recovery attack on the lgm (gsw-like) cryptosystem. In *PQCrypto*, 2021.

[40] Prastudy Fauzi, Martha Norberg Hovd, and Håvard Raddum. On the IND-CCA1 security of FHE schemes. *Cryptography*, 2022.

[41] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *EUROCRYPT*, 2004.

[42] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, 2009.

[43] Craig Gentry and Shai Halevi. Compressible FHE with applications to PIR. In *TCC*, 2019.

[44] Craig Gentry and Zulfikar Ramzan. Single-database private information retrieval with constant communication rate. In *ICALP*, 2005.

[45] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *EUROCRYPT*, 2014.

[46] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 1984.

[47] Google. Safe Browsing APIs (v4). `https://developers.google.com/safe-browsing/v4`.

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

[49] Daniel Günther, Maurice Heymann, Benny Pinkas, and Thomas Schneider. GPU-accelerated PIR with Client-Independent preprocessing for Large-Scale applications. In *Usenix Security*, 2022.

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

[51] Daniel Kales, Olamide Omolola, and Sebastian Ramacher. Revisiting user privacy for certificate transparency. In *EuroS&P*, 2019.

[52] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with Checklist. In *USENIX Security 21*, 2021.

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

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

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

[56] François Le Gall. Faster algorithms for rectangular matrix multiplication. In *FOCS*, 2012.

[57] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. In *CCS*, 2019.

[58] Jilan Lin, Ling Liang, Zheng Qu, Ishtiyaque Ahmad, Liu Liu, Fengbin Tu, Trinabh Gupta, Yufei Ding, and Yuan Xie. INSPIRE: In-storage private information retrieval via protocol and architecture co-design. In *ISCA*, 2022.

[59] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In *Topics in Cryptology – CT-RSA 2011*, 2011.

[60] Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In *International Conference on Information Security*, 2005.

[61] Jake Loftus, Alexander May, Nigel P. Smart, and Frederik Vercauteren. On CCA-secure somewhat homomorphic encryption. In Ali Miri and Serge Vaudenay, editors, *Selected Areas in Cryptography*, 2012.

[62] Wouter Lueks and Ian Goldberg. Sublinear scaling for multi-client private information retrieval. In *International Conference on Financial Cryptography and Data Security*, 2015.

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

[64] Yiping Ma, Ke Zhong, Tal Rabin, and Sebastian Angel. Incremental offline/online PIR. In *USENIX Security*, 2022.

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

[66] Samir Jordan Menon and David J. Wu. SPIRAL: Fast, high-rate single-server PIR via FHE composition. In *IEEE S&P*, 2022.

[67] Muhammad Haris Mughees, Hao Chen, and Ling Ren. OnionPIR: Response efficient single-server PIR. In *CCS*, 2021.

[68] Moni Naor and Eylon Yogev. Bloom filters in adversarial environments. In *CRYPTO*, 2015.

[69] Tatsuaki Okamoto and Shigenori Uchiyama. A new public-key cryptosystem as secure as factoring. In *EUROCRYPT*, 1998.

[70] Rafail Ostrovsky and William E Skeith. A survey of single-database private information retrieval: Techniques and applications. In *PKC*, 2007.

[71] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.

[72] Bijeeta Pal, Mazharul Islam, Thomas Ristenpart, and Rahul Chatterjee. Might I Get Pwned: A second generation password breach alerting service. In *USENIX Security*, 2022.

[73] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In *CRYPTO*, 2008.

[74] Giuseppe Persiano and Kevin Yeo. Limits of preprocessing for single-server PIR. In *SODA*, 2022.

[75] Joel Reardon, Jeffrey Pound, and Ian Goldberg. Relational-complete private information retrieval. Technical report, University of Waterloo, CACR, 2007.

[76] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 2009.

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

[78] Emily Stark and Chris Thompson. Opt-in SCT auditing, 2020. `https://docs.google.com/document/d/1G1Jy8LJgSqJ-B673GnTYIG4b7XRw2ZLtvvSlrqFcl4A/edit`.

[79] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 1969.

[80] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. Protecting accounts from credential stuffing with password breach alerting. In *USENIX Security*, 2019.

	Database size $N \times d$		Max. achievable throughput/core
Prior two-server PIR			
DPF PIR [51]	2^{25}	32 B	5,381 MiB/s*
XOR	2^{33}	1 bit	6,067 MiB/s*
Prior single-server PIR			
SealPIR [9]	2^{22}	288 B	97 MiB/s
MulPIR [8]	10^5	40 KiB	69 MiB/s†
FastPIR [6]	2^{20}	1024 B	215 MiB/s
OnionPIR [67]	2^{15}	30 KiB	104 MiB/s
Spiral [66]	2^{14}	100 KiB	353 MiB/s
SpiralPack [66]	2^{15}	30 KiB	303 MiB/s
SpiralStream [66]	2^{15}	30 KiB	518 MiB/s
SpiralStreamPack [66]	2^{15}	30 KiB	1,314 MiB/s★
This work (single-server PIR)			
SimplePIR	2^{33}	1 bit	6,496 MiB/s
DoublePIR	2^{33}	1 bit	5,217 MiB/s

Table 9: Maximal throughput measured for each PIR scheme, on databases of size roughly 1 GiB, consisting of N entries each of size d. The entry sizes, d, are those for which the highest throughput was reported in the corresponding paper. *The throughput is normalized by the number of servers, i.e., divided by two for 2-server PIR schemes. †We estimate MulPIR's throughput from the measurements given in the paper, as no implementation is publicly available at this date. ★SpiralStreamPack's throughput reported here is slightly lower than that reported in Table 7, due to variance in the measurements.

[81] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical private queries on public data. In *NSDI*, 2017.

[82] Ke Coby Wang and Michael K Reiter. Detecting stuffing of a user's credentials at her own accounts. In *USENIX Security 20*, 2020.

[83] Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *STOC*, 2012.

[84] Zhenfei Zhang, Thomas Plantard, and Willy Susilo. On the CCA-1 security of somewhat homomorphic encryption over the integers. In Mark D. Ryan, Ben Smyth, and Guilin Wang, editors, *Information Security Practice and Experience*, 2012.

[85] Mingxun Zhou, Wei-Kai Lin, Yiannis Tselekounis, and Elaine Shi. Optimal single-server private information retrieval. Cryptology ePrint Archive, Paper 2022/609, 2022. https://eprint.iacr.org/2022/609.

A Additional details on related work

For each PIR scheme from related work, we take its "optimal" entry size to be that for which the highest throughput was reported in the corresponding paper (or, if omitted, in a related paper). For each of our new PIR schemes (SimplePIR and DoublePIR), we compute its "optimal" entry size by executing the scheme on entries of increasing size, and selecting the entry size that yields the highest throughput. In Table 9, we display these entry sizes, along with each PIR scheme's measured throughput on a database roughly 1 GB in size, with entries of the optimal size.

B Client-state-recovery attacks on some existing PIR schemes

We demonstrate that several recent PIR schemes, including SealPIR and its descendants [6,8,9,66,67], are insecure against a certain type of active attack that enables the server to recover a client's long-term, secret state. After such an attack, the server can learn the contents of all of the client's PIR queries—even the ones made *before* the attacker compromised the server. In contrast, our PIR schemes are not vulnerable to this type of attack (see Remark 3.2) and thus provide a form of forward secrecy (Definition B.1).

Fully reasoning about the security of single-server PIR schemes under active attack is nuanced and beyond the scope of this paper. At the same time, we point out this active attack to highlight the fact that PIR schemes with secret long-term client state can carry additional security risks that our schemes do not.

The active attack applies to schemes in which the client holds and uses a persistent state across many queries. The state is a secret key for a homomorphic-encryption scheme. In particular, such schemes have the following syntax: first, the client runs a setup algorithm, which we call Setup_c. In contrast, in our new PIR schemes, it is the server who runs the setup algorithm. The Setup_c algorithm generates two pieces of information:

1. a public hint, σ_s, that the client transmits to the server, and
2. a private hint, σ_c, that the client keeps to itself.

Both σ_s and σ_c are re-used (by the server and the client respectively) across many queries: each time the client wants to query for some index $j \in [N]$, the client runs $\mathsf{Query}(\sigma_c, j) \to (\mathsf{st}, \mathsf{qu})$ and sends qu to the server. The server runs $\mathsf{Answer}(\mathsf{db}, \sigma_s, \mathsf{qu}) \to \mathsf{ans}$ and sends ans to the client. Finally, the client runs $\mathsf{Recover}(\mathsf{st}, \sigma_c, \mathsf{ans}) \to d$ to recover the database record of interest. As we will show in the remainder of this section, this syntax can be problematic as it lends itself to active attacks in which a malicious server recovers the client's private, persistent state σ_c.

We note that the reason why SealPIR and related schemes reuse a persistent, secret client state over many queries is to save on communication. In these schemes, σ_c holds the client's secret key for a homomorphic encryption scheme, while σ_s holds some "key-switching hints" associated with this key. The server uses these "key-switching hints" to perform query compression [9] or response packing [6].

Definition B.1 (Forward secrecy against active attacks)**.** Given a database size $N \in \mathbb{N}$, a number of queries $Q \in \mathbb{N}$, and a PIR scheme $(\mathsf{Setup}_c, \mathsf{Query}, \mathsf{Answer}, \mathsf{Recover})$, consider the following experiment between a challenger and an adversary,

1. The challenger computes $(\sigma_s, \sigma_c) \leftarrow \mathsf{Setup}_c(1^\lambda)$ and sends σ_s to the adversary.
2. The adversary sends $i_0, i_1 \in [N]$ to the challenger.
3. The challenger runs $\mathsf{Query}(\sigma_c, i_b) \rightarrow (\_, \mathsf{qu})$, and sends the query qu to the adversary.
4. For $q \in \{1, \cdots, Q\}$:
   - The adversary chooses index $j_q \in [N]$ and sends it to the challenger.
   - The challenger runs $\mathsf{Query}(\sigma_c, j_q) \rightarrow (\mathsf{st}_q, \mathsf{qu}_q)$ and sends the query qu_q to the adversary.
   - The adversary computes some (potentially malformed) ans'_q and sends it to the challenger.
   - The challenger runs $\mathsf{Recover}(\mathsf{st}_q, \sigma_c, \mathsf{ans}'_q) \rightarrow d$ and sends it to the adversary. (Note that d could be the failure symbol, $\perp$, if the Recover algorithm fails on ans'_q.)
5. The adversary outputs a guess $\tilde{b} \in \{0, 1\}$.

The PIR scheme satisfies (T, ϵ)-*forward secrecy against active attacks* for Q queries if, for all $N \in \mathbb{N}$ and all adversaries $\mathcal{A}$ running in time at most T and making up to Q queries in the above experiment, it holds that

$$|\Pr[W_0] - \Pr[W_1]| \leq \epsilon,$$

where W_b denotes the event that $\mathcal{A}$ outputs "1" in Experiment b, for $b \in \{0, 1\}$.

Active attacks on some stateful PIR schemes. Existing PIR schemes, which let the client reuse a secret key for homomorphic encryption indefinitely, do not provide forward secrecy against active attacks because their underlying homomorphic encryption schemes are not CCA1-secure [24, 39, 40, 61, 84]. In the remainder of this section, we show that there exist attacks that, with polynomially many PIR queries, recover the entire client state. For instance, consider SealPIR with $d = 1$ (i.e., no recursion) and recall that the scheme uses BFV encryption [18, 38], where the secret keys are ring elements $s = s_0 + s_1 x + s_2 x^2 + \cdots + s_{n-1} x^{n-1}$ in the quotient ring $R_q := \mathbb{Z}_q[x]/(x^n + 1)$ and p denotes the plaintext modulus. We modify the existing CCA attacks on BFV encryption [24] to obtain Algorithm 1, which is an active attack on SealPIR. Conceptually, this attack recovers the client's secret key by performing a binary search for each $s_i \in \mathbb{Z}_q$, for $i \in [n]$.

Analysis of Algorithm 1. In SealPIR, when $d = 1$, the client's private hint σ_c is the BFV secret key, i.e., a secret ring element $s \in R_q$, that the client uses to generate its queries. The server answers each client query with a single BFV ciphertext

Algorithm 1 Active attack for recovering $s_i \in \mathbb{Z}_q$ in SealPIR

Setup: Let $\mathcal{O}(\cdot) \leftarrow \mathsf{Recover}(\mathsf{st}, \sigma_c, \cdot)$. (To implement $\mathcal{O}(\cdot)$ in step 4 of the experiment in Definition B.1, first send $\mathcal{C}$ any index in $[N]$, and then send $\mathcal{C}$ the desired input to $\mathcal{O}(\cdot)$.)
Define $\Delta \leftarrow \lfloor q/p \rfloor$.
Algorithm: Let $g \leftarrow \mathcal{O}(-1, 0)$, where $g \in \mathbb{Z}_q[x]/(x^n + 1)$.
Let $t \leftarrow g[i]$ (where $g[i]$ denotes the coefficient of x^i in g).
if $t \neq 0$ **then**
  Define $M \leftarrow \Delta$.
else
  Define $M \leftarrow (q \bmod p) + \Delta$.
end if
Perform binary search to find the smallest $j \in \{0, 1, \cdots, M\}$ such that $\mathcal{O}(-1, j \cdot x^i)[i] = (t + 1) \bmod p$.
Output $((t \cdot \Delta + \lfloor \frac{\Delta}{2} \rfloor + 1) - j) \bmod q$.

$(a, b) \in R_q^2$. Finally, it holds that

$$\begin{aligned} \mathcal{O}(-1, j \cdot x^i) &= \mathsf{Dec}_{BFV}(s, (-1, j \cdot x^i)) \\ &= \mathsf{Round}_\Delta(j \cdot x^i + s)/\Delta, \end{aligned} \tag{1}$$

where $\mathsf{Round}_\Delta(g)$ rounds each coefficient of the polynomial g to the closest multiple of $\Delta := \lfloor q/p \rfloor$.

By Equation (1), we see that t (defined as in Algorithm 1) is equal to $\mathcal{O}(-1, 0)[i]$, or, equivalently, $\mathsf{Round}_\Delta(s_i)/\Delta \in \mathbb{Z}_p$. Therefore, s_i must fall into the following ranges:

- If $t = 0$, then

$$s_i \in \left\{ p \cdot \Delta - \left\lfloor \frac{\Delta}{2} \right\rfloor, \cdots, q - 1 \right\} \cup \left\{ 0, \cdots, \left\lfloor \frac{\Delta}{2} \right\rfloor \right\}.$$

- If $t \neq 0$, then $s_i \in \left\{ t \cdot \Delta - \lfloor \frac{\Delta}{2} \rfloor, \cdots, t \cdot \Delta + \lfloor \frac{\Delta}{2} \rfloor \right\}$.

Let j be the smallest value in $\{0, \ldots, M\}$ such that $\mathsf{Round}_\Delta(j + s_i)/\Delta = \mathsf{Round}_\Delta(s_i)/\Delta + 1 \in \mathbb{Z}_p$. The algorithm performs a binary search to find j, requiring roughly $\log q$ queries to $\mathcal{O}(\cdot)$. We observe that, after this binary search, j will be exactly $T - s_i$, where $T = t \cdot \Delta + \lfloor \frac{\Delta}{2} \rfloor + 1$ is the "next rounding boundary" of $\mathsf{Round}_\Delta(\cdot)$. Finally, the algorithm outputs $T - j$, which is equal to s_i. Thus, it successfully recovers $s_i \in \mathbb{Z}_q$, with roughly $\log q$ queries to $\mathcal{O}(\cdot)$.

To recover the entire secret key s, an attacker can perform the above binary search for all coefficients in $\{s_i\}_{i \in [n]}$ in parallel. In other words, instead of asking the oracle $\mathcal{O}$ for $(-1, j \cdot x^i)$, the attacker may ask for $(-1, \sum_{i=1}^n j_i \cdot x^i)$, where the $\{j_i\}_{i \in [n]}$ are the query points checked by each of the n parallel binary searches. So, the total number of queries needed to recover $s \in R_q$ remains roughly $\log q$. After this attack, the adversary knows the client's secret key, s, and can thus learn all of the client's past and future PIR queries.

C Additional material on SimplePIR

C.1 Parameter selection

We instantiate SimplePIR with the parameter choices that maximize its throughput, while meeting the desired correctness and security requirements. Given a database size N, a correctness failure probability δ, an adversary's runtime T, and a security failure probability ϵ, we proceed as follows:

1. We first fix the ciphertext modulus q to be one of $\{2^{16}, 2^{32}, 2^{64}\}$, as modern hardware natively supports operations over $\mathbb{Z}_q$ with these moduli.
2. We use LWE hardness estimates [7] to pick the LWE secret dimension n along with the LWE error distribution χ, such that a collection of $\sqrt{N}$ such LWE samples is (T, ϵ)-secure against known attacks.
3. We compute the largest plaintext modulus, p, such that the chosen LWE parameters support at least $\sqrt{N}$ homomorphic additions, with correctness error probability δ.

In this work, we take χ to be a discrete Gaussian distribution. We give concrete values for our parameters in Section 4.2.

C.2 Correctness and security of SimplePIR

We now give the formal SimplePIR theorem statement:

Theorem C.1 (SimplePIR)**.** *On database size $N \in \mathbb{N}$, correctness failure probability δ, adversary runtime T, and security failure probability ϵ, let*

- *χ be the discrete Gaussian distribution with variance σ^2,*
- *(n, q, χ) be LWE parameters achieving (T, ϵ)-security for $\sqrt{N}$ LWE samples, and*
- *$p \in \mathbb{N}$ be a plaintext modulus chosen to satisfy*

$$\lfloor q/p \rfloor \geq \sqrt{2} \cdot \sigma \cdot p \cdot N^{1/4} \cdot \sqrt{\ln(2/\delta)}. \tag{2}$$

Then, for a random LWE matrix $\mathbf{A} \xleftarrow{R} \mathbb{Z}_q^{\sqrt{N} \times n}$, SimplePIR is a $(T - O(\sqrt{N}), 2\epsilon)$-secure PIR scheme on database size N, over plaintext space $\mathbb{Z}_p$, with correctness error δ.

Proof. We prove correctness and security separately.

Correctness. Consider a client that interacts with the server to query for the database value at $(i_{\text{row}}, i_{\text{col}}) \in [\sqrt{N}]^2$. Let $\mathbf{u}_{i_{\text{row}}}$ denote the unit vector i_{row} in $\mathbb{Z}_q^{\sqrt{N}}$ (i.e., the vector of all zeros, with a single '1' at index i_{row}). The Recover routine in SimplePIR computes:

$$\begin{aligned}
\hat{d} &= \mathsf{ans}[i_{\text{row}}] - \mathsf{hint}_c[i_{\text{row}}, :] \cdot \mathbf{s} \\
&= \mathbf{u}_{i_{\text{row}}}^T \cdot (\mathsf{ans} - \mathsf{hint}_c \cdot \mathbf{s}) \\
&= \mathbf{u}_{i_{\text{row}}}^T \cdot (\mathsf{db} \cdot (\mathbf{A}\mathbf{s} + \mathbf{e} + \Delta \cdot \mathbf{u}_{i_{\text{col}}}) - (\mathsf{db} \cdot \mathbf{A}) \cdot \mathbf{s}) \\
&= \mathbf{u}_{i_{\text{row}}}^T \cdot (\mathsf{db} \cdot \mathbf{e} + \Delta \cdot \mathsf{db} \cdot \mathbf{u}_{i_{\text{col}}}) \\
&= \mathsf{db}[i_{\text{row}}, :] \cdot \mathbf{e} + \Delta \cdot \mathsf{db}[i_{\text{row}}, i_{\text{col}}],
\end{aligned}$$

where $\mathsf{db}[i_{\text{row}}, :]$ denotes row i_{row} of the database matrix db, and $\mathsf{db}[i_{\text{row}}, i_{\text{col}}]$ denotes the element at $(i_{\text{row}}, i_{\text{col}})$ in db. Recovery succeeds when $\mathsf{Round}_\Delta(\hat{d})/\Delta$ is equal to $\mathsf{db}[i_{\text{row}}, i_{\text{col}}]$. This happens if and only if $|\mathsf{db}[i_{\text{row}}, :] \cdot \mathbf{e}| < \Delta/2$.

We can guarantee successful decryption by ensuring that $\lfloor p/2 \rfloor \cdot \sqrt{N} \cdot |\mathbf{e}|_\infty < \Delta/2$ (because we can store the database entries—which are elements in $\mathbb{Z}_p$—as values in the range $\{-\lfloor p/2 \rfloor, -\lfloor p/2 \rfloor + 1, \ldots, \lceil p/2 \rceil - 1\}$, so that they have maximal norm $\lfloor p/2 \rfloor$). However, we instead use a tighter bound on $|\mathsf{db}[i_{\text{row}}, :] \cdot \mathbf{e}|$ by relying on properties of the error distribution, χ. As a result, we will obtain a scheme in which decryption succeeds with probability $1 - \delta$ (rather than 1) with a larger plaintext modulus, p.

Indeed, as χ is the discrete Gaussian distribution with variance $\sigma^2 = \frac{s^2}{2\pi}$ for some $s > 0$, by [59, Lemma 2.2][11, Lemma 2.4], for any $T > 0$, it holds that,

$$\Pr\left[|\mathsf{db}[i_{\text{row}}, :] \cdot \mathbf{e}| \geq T \cdot s \, \|\mathsf{db}[i_{\text{row}}, :]\|\right] < 2\exp(-\pi \cdot T^2),$$

where $\|\cdot\|$ denotes the Euclidean norm.
Taking $T = \Delta/(2s \cdot \|\mathsf{db}[i_{\text{row}}, :]\|)$, we see that

$$\Pr\left[|\mathsf{db}[i_{\text{row}}, :] \cdot \mathbf{e}| \geq \Delta/2\right] < \delta,$$

as long as

$$2\exp\left(-\pi \cdot \left(\frac{\Delta}{2s \cdot \|\mathsf{db}[i_{\text{row}}, :]\|}\right)^2\right) \leq \delta.$$

Equivalently, recovery fails with probability at most δ, if

$$\Delta \geq 2s \cdot \|\mathsf{db}[i_{\text{row}}, :]\| \cdot \sqrt{\frac{\ln(2/\delta)}{\pi}}.$$

Again, as we can store the elements in db in the range $[-\lfloor p/2 \rfloor, \lfloor p/2 \rfloor]$, we have that

$$\left\|\mathsf{db}_{[i_{\text{row}}, :]}\right\| \leq \sqrt{\sqrt{N} \cdot \lfloor p/2 \rfloor^2} = N^{1/4} \cdot \lfloor p/2 \rfloor.$$

In addition, we know that $\Delta = \lfloor q/p \rfloor$ and $s = \sigma \cdot \sqrt{2\pi}$. Thus, recovery fails with probability at most δ, as long as:

$$\lfloor q/p \rfloor \geq \sigma \cdot \sqrt{2\pi} \cdot p \cdot N^{1/4} \cdot \sqrt{\frac{\ln(2/\delta)}{\pi}},$$

or, equivalently,

$$\lfloor q/p \rfloor \geq \sqrt{2} \cdot \sigma \cdot p \cdot N^{1/4} \cdot \sqrt{\ln(2/\delta)},$$

By Equation (2), this condition holds. Thus, SimplePIR is correct, except with error probability at most δ.

Security. Security follows from the fact that the LWE problem is hard, even when the LWE matrix $\mathbf{A}$ is reused across many independent trials, each using an independently generated LWE secret $\mathbf{s} \xleftarrow{R} \mathbb{Z}_q^n$, as shown in prior work [73, Lemma 7.3]. We connect the security of SimplePIR to the hardness of LWE with the following lemma:

Lemma C.2. *Let $N \in \mathbb{N}$ be the database size, (n, q, χ) be the LWE parameters, and $\mathbf{A} \in \mathbb{Z}_q^{\sqrt{N} \times n}$ be the random LWE matrix used in SimplePIR. For any $i \in [N]$, we define the distribution*

$$\mathcal{Q}_i = \{(\mathbf{A}, \mathsf{qu}_i) : \_, \mathsf{qu}_i \leftarrow \mathsf{Query}(i)\}.$$

If the (n, q, χ)-LWE problem with $\sqrt{N}$ samples is (T, ϵ)-hard, then any algorithm running in time $T - O(\sqrt{N})$ can have success probability at most ϵ in distinguishing $\mathcal{Q}_i$ from the distribution $\{(\mathbf{A}, \mathbf{r}) : \mathbf{r} \xleftarrow{R} \mathbb{Z}_q^{\sqrt{N}}\}$.

Proof. Consider any index $i \in [N]$. We decompose i into the pair of coordinates $(i_{\text{row}}, i_{\text{col}}) \in [\sqrt{N}]^2$, as done in the Query routine in SimplePIR, and let $\mathbf{u}_{i_{\text{col}}}$ denote unit vector i_{col} in $\mathbb{Z}_q^{\sqrt{N}}$. Additionally, we define the following distributions:

- $\mathcal{D}_1 = \{(\mathbf{A}, \mathbf{As} + \mathbf{e}) : \mathbf{s} \xleftarrow{R} \mathbb{Z}_q^n, \mathbf{e} \xleftarrow{R} \chi^{\sqrt{N}}\}$, and
- $\mathcal{D}_2 = \{(\mathbf{A}, \mathbf{r}) : \mathbf{r} \xleftarrow{R} \mathbb{Z}_q^{\sqrt{N}}\}$.

Now, consider the simulator $\mathcal{S}$ that, given as input $(\mathbf{A}, \mathbf{v}) \in \mathbb{Z}_q^{\sqrt{N} \times n} \times \mathbb{Z}_q^{\sqrt{N}}$, computes and outputs $(\mathbf{A}, \mathbf{v} + \lfloor q/p \rfloor \cdot \mathbf{u}_{i_{\text{col}}})$. We observe that:

- when $(\mathbf{A}, \mathbf{v})$ is sampled from $\mathcal{D}_1$, the simulator's output is distributed identically to $\mathcal{Q}_i$.
- when $(\mathbf{A}, \mathbf{v})$ is sampled from $\mathcal{D}_2$, the simulator's output is distributed identically to $\mathcal{D}_2$.

As the (n, q, χ)-LWE problem with $\sqrt{N}$ samples is (T, ϵ)-hard, we know that any algorithm running in time T has advantage at most ϵ in distinguishing between $\mathcal{D}_1$ and $\mathcal{D}_2$. Our simulator $\mathcal{S}$ runs in time $O(\sqrt{N})$. Thus, it must hold that any algorithm distinguishing between $\mathcal{Q}_i$ and $\mathcal{D}_2$ in time at most $T - O(\sqrt{N})$ can have success probability at most ϵ. □

Lemma C.2 shows that any algorithm running in time $T - O(\sqrt{N})$ can distinguish the queries made by a client in SimplePIR from random vectors in $\mathbb{Z}_q^{\sqrt{N}}$ with success probability at most ϵ. Therefore, any algorithm running in time $T - O(\sqrt{N})$ can distinguish the queries made by a client in SimplePIR to any pair of indices $i \in [N]$ and $j \in [N]$ with success probability at most 2ϵ. In other words, SimplePIR is $(T - O(\sqrt{N}), 2\epsilon)$-secure. □

We can extend the security definition in Section 3.2 to handle Q queries, by stating that any two sequences of Q queries produce indistinguishable query distributions. A hybrid argument proves the following corollary:

Corollary C.3 (Q-query security of SimplePIR)**.** *Consider any two sequences of Q indices, $I, J \in [N]^Q$, and the query distributions, $\mathcal{D}_i$ and $\mathcal{D}_j$, that they induce, i.e.,*

$$\begin{cases} \mathcal{D}_i = \{(\mathbf{A}, \mathsf{qu}_k) : \_, \mathsf{qu}_k \leftarrow \mathsf{Query}(I_k)\}_{k \in [Q]} \\ \mathcal{D}_j = \{(\mathbf{A}, \mathsf{qu}_k) : \_, \mathsf{qu}_k \leftarrow \mathsf{Query}(J_k)\}_{k \in [Q]} \end{cases}$$

If the (n, q, χ)-LWE problem with $\sqrt{N}$ samples is (T, ϵ)-hard, then $\mathcal{D}_i$ is $(T - O(Q \cdot \sqrt{N}), 2Q\epsilon)$-indistinguishable from $\mathcal{D}_j$.

C.3 Additional extensions and optimizations

Answering many client queries at once. To answer a client query, the SimplePIR server multiplies the $\sqrt{N}$-by-$\sqrt{N}$ database matrix by the client's dimension-$\sqrt{N}$ query vector, in $2N$ operations. If the server wants to answer $\sqrt{N}$ client queries at once, it can use fast matrix-multiplication algorithms [28, 83] to compute this more efficiently than answering each query independently—in $o(N^{3/2})$ time. This observation comes directly from prior work [12, 62]. We have not yet experimented with this optimization, since it is not clear to us that the asymptotic efficiency gain will translate to concrete cost improvements.

Faster preprocessing. In its preprocessing step, the server computes the matrix-matrix product of the database with the LWE matrix, $\mathbf{A}$. The asymptotic performance of this step can be improved using fast matrix-multiplication algorithms [28, 56, 79, 83]. We suspect that such a refinement will not improve the concrete performance for our parameter sizes.

Handling database updates. As discussed in Remark 3.1, if the database changes, the server inherently needs perform some of its preprocessing work again. In SimplePIR, the amount of preprocessing work that needs to be repeated is proportional to the number of rows in the database matrix whose contents have changed. More specifically, the preprocessing phase in SimplePIR computes hint_c to be the product of the database matrix, db, and the LWE matrix, $\mathbf{A}$. Thus, when some row $\mathbf{d}$ of the database changes, only the corresponding row in hint_c changes and must be updated. In other words, it is sufficient for the server to compute $\mathbf{d} \cdot \mathbf{A} \in \mathbb{Z}_q^{1 \times n}$ (rather than the much more expensive $\mathsf{db} \cdot \mathbf{A} \in \mathbb{Z}_q^{\sqrt{N} \times n}$) and send it back to the client. Additions and deletions of rows in db can be handled similarly.

Decreasing the online download with local rounding. In SimplePIR's online phase, the client downloads a vector of $\sqrt{N}$ elements in the ciphertext space, $\mathbb{Z}_q$, performs a linear computation on them, and finally rounds the result to map it into the message space, $\mathbb{Z}_p$. Prior work [35, Lemma 3.2][36, Lemma 4.1][17, Lemma 1] observes that, for any value $v \in \mathbb{Z}_q$ that is "close" to a multiple of Δ, and for a random $r \in \mathbb{Z}_q$, it holds with high probability that

$$\mathsf{Round}_\Delta(v) = \mathsf{Round}_\Delta(v + r) - \mathsf{Round}_\Delta(r).$$

Using this observation (with $v \leftarrow \mathsf{ans} - \mathsf{hint}_c \cdot s$ and $r \leftarrow \mathsf{hint}_c \cdot s$), the server and the client could each locally round the values they contribute to the client's final computation (ans and $\mathsf{hint}_c \cdot \mathbf{s}$, respectively), decreasing the online download to only $\sqrt{N}$ elements in $\mathbb{Z}_p$. If the LWE parameters are chosen appropriately (namely, $\lfloor q/p \rfloor \geq \sqrt{2} \cdot \frac{4}{\delta} \cdot \sigma \cdot p \cdot N^{1/4} \cdot \sqrt{\ln(8/\delta)}$), then this local rounding does not affect the scheme's correctness (i.e., recovery still succeeds, except with probability δ).

In practice, this optimization decreases the online download by a factor of $\frac{\log q}{\log p} \approx 3\times$, at the expense of requiring a much smaller plaintext modulus, p, to maintain correctness and thus

markedly reducing SimplePIR's throughput. Exploring other approaches to local rounding [19] or to correcting the errors that it introduces are interesting directions for future work.

D Linearly homomorphic encryption with preprocessing

In this section, we abstract out the key property of Regev encryption that enables SimplePIR's high throughput. We expect this primitive to have applications beyond PIR.

Specifically, we introduce the notion of *linearly homomorphic encryption with preprocessing*. This notion is similar to standard linearly homomorhpic encryption: given an encryption of vector $\mathbf{v} \in \mathbb{Z}_p^m$ (where the ring $\mathbb{Z}_p$ and dimension m are parameters of the scheme), it is possible to homomorphically evaluate any linear function $f : \mathbb{Z}_p^m \to \mathbb{Z}_p$ on the encrypted values; after evaluation, the resulting ciphertext decrypts to the function value $f(v)$. What distinguishes our use of Regev encryption from standard linearly homomorphic encryption is that we can preprocess the function f to speed up the homomorphic-evaluation operation. In particular, given a linear function f, we compute a preprocessed hint, hint_f. The hint enables homomorphic evaluation of f on ciphertexts encrypting any $\mathbf{v} \in \mathbb{Z}_p^m$ at very low computational cost—nearly the cost of evaluating f on a plaintext vector.

D.1 Definition

Formally, a *linearly homomorphic encryption scheme with preprocessing* consists of the following efficient algorithms, which are implicitly parameterized by a security parameter $n \in \mathbb{N}$, a correctness parameter $\delta \in \mathbb{N}$, a plaintext dimension $m \in \mathbb{N}$, a key space $\mathcal{K}$, and a plaintext space $\mathbb{Z}_p$:

$\mathsf{Setup}() \to \mathsf{pp}$. Sample and output the public parameters.

$\mathsf{Enc}(\mathsf{pp}, \mathsf{sk}, \mathbf{v}) \to \mathsf{ct}$. Given the public parameters pp, a secret key $\mathsf{sk} \in \mathcal{K}$, and a vector $\mathbf{v} \in \mathbb{Z}_p^m$, output a ciphertext ct.

$\mathsf{Preproc}(\mathsf{pp}, f) \to \mathsf{hint}_f$. Given public parameters pp and a linear function $f : \mathbb{Z}_p^m \to \mathbb{Z}_p$, output a function hint hint_f.

$\mathsf{Apply}(\mathsf{pp}, f, \mathsf{ct}) \to \mathsf{ct}_f$. Given the public parameters pp, a linear function $f : \mathbb{Z}_p^m \to \mathbb{Z}_p$, and a ciphertext ct, produce a ciphertext ct_f that encrypts the value of the function f applied to the vector encrypted by ct.

$\mathsf{Dec}(\mathsf{sk}, \mathsf{hint}_f, \mathsf{ct}_f) \to d$. Given a secret key sk, the function hint hint_f, and a ciphertext ct_f, output a decrypted value d.

Correctness. After encryption, preprocessing and application of the linear function, and decryption are carried out sequentially, the output should be the correct function value, except with negligible probability in the implicit correctness parameter. Formally, we say that the encryption scheme has *correctness error* δ if, for all $\mathsf{sk} \in \mathcal{K}$, for all $\mathbf{v} \in \mathbb{Z}_p^m$, and linear functions $f \in \mathbb{Z}_p^m \to \mathbb{Z}_p$, the following probability is at least $1 - \delta$:

$$\Pr\left[\, d = f(\mathbf{v}) : \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}() \\ \mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pp}, \mathsf{sk}, \mathbf{v}) \\ \mathsf{hint}_f \leftarrow \mathsf{Preproc}(\mathsf{pp}, f) \\ \mathsf{ct}_f \leftarrow \mathsf{Apply}(\mathsf{pp}, f, \mathsf{ct}) \\ d \leftarrow \mathsf{Dec}(\mathsf{sk}, \mathsf{hint}_f, \mathsf{ct}_f) \end{array} \right].$$

Security. We require semantic security [46]. In other words, ct should reveal no information about the encrypted vector $\mathbf{v} = (v_1, \ldots, v_m)$. Formally, given vectors $\mathbf{v}_0, \mathbf{v}_1 \in \mathbb{Z}_p^m$, for $b \in \{0, 1\}$, we define the value W_b as:

$$W_b := \Pr\left[\mathcal{A}(\mathsf{pp}, \mathsf{ct}) = 1 : \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}() \\ \mathsf{sk} \leftarrow \mathcal{K} \\ \mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pp}, \mathsf{sk}, \mathbf{v}_b) \end{array} \right].$$

Then, we say that the encryption scheme is *secure* if, for all efficient adversaries $\mathcal{A}$, for all $\mathbf{v}_0, \mathbf{v}_1 \in \mathbb{Z}_p^m$, it holds that $|W_0 - W_1|$ is negligible in the implicit security parameter.

D.2 Construction

In Figure 11, we construct a linear homomorphic encryption scheme with preprocessing from Regev encryption. The construction yields the following theorem:

Theorem D.1. *Assume the (n, q, χ)-LWE problem with m samples is (T, ϵ)-hard, and let $p \in \mathbb{N}$ be a suitable plaintext modulus for Regev encryption with these parameters (cf. Section 4.2) supporting m homomorphic additions. Then, the construction of Figure 11 is a $(T - O(m), 2\epsilon)$-secure linearly homomorphic encryption scheme with preprocessing for plaintext dimension m and plaintext modulus p where:*

- *the hint consists of n elements of $\mathbb{Z}_q$,*
- *the ciphertext encrypting a vector in $\mathbb{Z}_p^m$ consists of m elements of $\mathbb{Z}_q$,*
- *the evaluation routine* Apply *requires m additions and m multiplications in $\mathbb{Z}_q$, and*
- *the ciphertext that* Apply *outputs consists of one $\mathbb{Z}_q$ element.*

It is worth emphasizing the efficiency of the construction implied by Theorem D.1. By our choice of LWE parameters

	Size			***Running time (per bit)***			
	Hint	Key	Ct/bit	Preproc	Enc	Apply	Dec
Factoring [69, 71]	n/a	λ^5	λ^2	n/a	λ^2	λ^2	λ^2
EC ElGamal [37]	n/a	λ	λ	n/a	λ^2	λ	λ
Regev [73, 76]	n/a	λ	1	n/a	λ	λ	λ
Ring LWE [63]	n/a	λ	1	n/a	1	1	1
This work (LWE)	λ	λ	1	λ	λ	1	λ

Table 10: Comparison of linearly homomorphic encryption schemes on plaintext dimension m. We suppress $\log \lambda$ factors and we assume that it is possible to perform modular multiplication in linear time. Since our scheme uses plain LWE instead of ring LWE, it is simpler to implement—there is no need for polynomial arithmetic, fast Fourier transforms, or a modulus of special form.

in Equation (2), the number of bits encoding each ciphertext element (i.e., $\log q$) grows logarithmically with the plaintext dimension m and is independent of the security parameter n. Thus, our construction has:

- a hint size that grows only logarithmically with the size of the linear function f,
- ciphertext size that is only a $\log m$ factor larger than the corresponding plaintext (and that is independent of the security parameter), and
- a homomorphic evaluation routine that is essentially as fast as plaintext evaluation of f, as homomorphic evaluation just requires computing f over $\mathbb{Z}_q$ instead of over $\mathbb{Z}_p$. This evaluation time is independent of the security parameter.

In contrast, when using Paillier [71] or ElGamal-based [37] linearly homomorphic encryption, the ciphertext size and evaluation time grow quadratically or cubically with the security parameter (Table 10).

D.3 Abstract view of SimplePIR

In Figure 12, we re-express the SimplePIR construction of Section 4 in the language of linear homomorphic encryption with preprocessing. The PIR scheme of Figure 12 demonstrates that, given a linearly homomorphic encryption scheme with preprocessing, there exists a PIR scheme with $\sqrt{N}$ communication cost and for which the server's online computation time consists of running the encryption's Apply algorithm $\sqrt{N}$ times, each on a vector of dimension $\sqrt{N}$. The client's hint—which it must fetch in an offline phase—consists of $\sqrt{N}$ hints for the encryption scheme, each computed for a linear function with $\sqrt{N}$ inputs.

An interesting task for future work would be to construct a linear homomorphic encryption scheme with preprocessing with smaller hints. This would reduce the offline communication in the resulting PIR schemes.

E Additional material on DoublePIR

We give a formal description of DoublePIR in Figure 13.

E.1 Parameter selection

As for SimplePIR, we take the ciphertext modulus, q, to be 2^{32} and the LWE distribution, χ, to be a discrete Gaussian in DoublePIR. We use hardness estimates [7] to select appropriate choices of χ and of the LWE dimension, n. Our selection of the plaintext modulus, p, in DoublePIR is slightly more conservative than that for SimplePIR, as for correctness to hold, the recovery routine must succeed for:

- the single element read from the first-level database, and
- the $(n+1)$ elements read from the second-level database.

We detail our calculation of p in Theorem E.1. Here, we give sample choices of the plaintext modulus, p, for different database sizes:

Parameters: LWE parameters (n, q, χ), a plaintext modulus $p \ll q$. Define $\Delta := \lfloor q/p \rfloor$. The keyspace is $\mathcal{K} = \mathbb{Z}_q^n$.

$\mathsf{Setup}() \to \mathsf{pp} \in \mathbb{Z}_q^{m \times n}$

- Sample $\mathbf{A} \xleftarrow{R} \mathbb{Z}_q^{m \times n}$.
- Return $\mathsf{pp} \leftarrow \mathbf{A}$.

$\mathsf{Enc}(\mathsf{pp}, \mathsf{sk}, \mathbf{v}) \to \mathsf{ct} \in \mathbb{Z}_q^m$

- Parse $(\mathbf{A} \in \mathbb{Z}_q^{m \times n}) \leftarrow \mathsf{pp}$.
- Sample $\mathbf{e} \xleftarrow{R} \chi^m$.
- Return $\mathsf{ct} \leftarrow \mathbf{A} \cdot \mathsf{sk} + \mathbf{e} + \Delta \cdot \mathbf{v} \in \mathbb{Z}_q^m$.

$\mathsf{Preproc}(\mathsf{pp}, f) \to \mathsf{hint}_f \in \mathbb{Z}_q^{1 \times n}$

- Parse $(\mathbf{A} \in \mathbb{Z}_q^{m \times n}) \leftarrow \mathsf{pp}$.
- Parse $(\mathbf{f} \in \mathbb{Z}_q^{1 \times m}) \leftarrow f$ as a row vector.
- Return $\mathsf{hint}_f \leftarrow \mathbf{f} \cdot \mathbf{A}$.

$\mathsf{Apply}(\mathsf{pp}, f, \mathsf{ct}) \to \mathsf{ct}_f \in \mathbb{Z}_q$

- Parse:
  - $(\mathbf{f} \in \mathbb{Z}_q^{1 \times m}) \leftarrow f$ as a row vector,
  - $(\mathbf{c} \in \mathbb{Z}_q^m) \leftarrow \mathsf{ct}$.
- Return $\mathsf{ct}_f \leftarrow \mathbf{f} \cdot \mathbf{c} \in \mathbb{Z}_q$.

$\mathsf{Dec}(\mathsf{pp}, \mathsf{sk}, \mathsf{hint}_f, \mathsf{ct}_f) \to d \in \mathbb{Z}_p$

- Compute $\hat{d} \leftarrow \mathsf{ct}_f - \mathsf{hint}_f \cdot \mathsf{sk} \in \mathbb{Z}_q$.
- Let $v \in \mathbb{Z}_q$ be $\mathsf{Round}_\Delta(\hat{d})$, which is $\hat{d}$ rounded to the nearest integral multiple of Δ.
- Return $d \leftarrow v/\Delta \in \mathbb{Z}_p$.

Figure 11: Regev Encryption expressed as linear homomorphic encryption with preprocessing.

Database size N:	2^{26}	2^{28}	2^{30}	2^{34}	2^{38}	2^{42}
Plaintext modulus p:	929	781	657	464	328	231

E.2 Correctness and security of DoublePIR

We now prove the following theorem:

Theorem E.1 (DoublePIR)**.** *On database size $N \in \mathbb{N}$, correctness failure probability δ, adversary runtime T, and security failure probability ϵ, let*

- *$\ell, m \in \mathbb{N}$ be database dimensions (such that $\ell \cdot m \geq N$),*
- *χ be the discrete Gaussian distribution with variance σ^2,*
- *(n, q, χ) be LWE parameters achieving (T, ϵ)-security for $\max(\ell, m)$ LWE samples, and*
- *$p \in \mathbb{N}$ be a plaintext modulus chosen to satisfy*

$$\lfloor q/p \rfloor \geq \sigma p \sqrt{2 \max(m, \ell) \cdot \ln\left(\frac{2(\kappa(n+1)+1)}{\delta}\right)}, \quad (3)$$

where $\kappa \in \mathbb{Z}$ is the scalar $\left\lceil \frac{\log q}{\log p} \right\rceil$.

Parameters: a database size N, a linearly homomorphic encryption with preprocessing scheme $(\mathsf{Setup}, \mathsf{Enc}, \mathsf{Preproc}, \mathsf{Apply}, \mathsf{Dec})$ with plaintext dimension $\sqrt{N}$, plaintext space $\mathbb{Z}_p$, key space $\mathcal{K}$, and public parameters pp, computed as $\mathsf{pp} \leftarrow \mathsf{Setup}()$. Our construction uses the following helper routine:

$\mathsf{ToFuncs}(\mathsf{db}) \to (f_1, \ldots, f_{\sqrt{N}})$:

- View the database db as vectors $\mathbf{d}_1, \ldots, \mathbf{d}_{\sqrt{N}} \in \mathbb{Z}_p^{\sqrt{N}}$.
- For all $i \in [\sqrt{N}]$, define $f_i(\mathbf{x}) := \langle \mathbf{d}_i, \mathbf{x} \rangle \in \mathbb{Z}_p$. (Here, f_i is a linear function on $\mathbb{Z}_p^{\sqrt{N}}$ to $\mathbb{Z}_p$.)
- Output $(f_1, \ldots, f_{\sqrt{N}})$.

$\mathsf{Setup}(\mathsf{db}) \to (\mathsf{hint}_s, \mathsf{hint}_c)$.

- Let $(f_1, \ldots, f_{\sqrt{N}}) \leftarrow \mathsf{ToFuncs}(\mathsf{db})$.
- For all $i \in [\sqrt{N}]$, compute: $\mathsf{hint}_i \leftarrow \mathsf{Preproc}(\mathsf{pp}, f_i)$.
- Set $\mathsf{hint}_c \leftarrow (\mathsf{hint}_1, \ldots, \mathsf{hint}_{\sqrt{N}})$.
- Return $(\_, \mathsf{hint}_c)$.

$\mathsf{Query}(i) \to (\mathsf{st}, \mathsf{qu})$.

- Write i as a pair $(j, k) \in [\sqrt{N}] \times [\sqrt{N}]$.
- Sample $\mathsf{sk} \xleftarrow{R} \mathcal{K}$.
- Compute $\mathsf{qu} \leftarrow \mathsf{Enc}(\mathsf{pp}, \mathsf{sk}, \mathbf{u}_j \in \mathbb{Z}_p^{\sqrt{N}})$, where $\mathbf{u}_j$ is the vector of all zeros with a single '1' at index j.
- Set $\mathsf{st} \leftarrow (\mathsf{sk}, k)$ and return $(\mathsf{st}, \mathsf{qu})$.

$\mathsf{Answer}(\mathsf{db}, \mathsf{hint}_s, \mathsf{qu}) \to \mathsf{ans}$.

- Let $(f_1, \ldots, f_{\sqrt{N}}) \leftarrow \mathsf{ToFuncs}(\mathsf{db})$.
- For all $i \in [\sqrt{N}]$, compute: $\mathsf{ans}_i \leftarrow \mathsf{Apply}(\mathsf{pp}, f_i, \mathsf{qu})$.
- Return $\mathsf{ans} \leftarrow (\mathsf{ans}_1, \ldots, \mathsf{ans}_{\sqrt{N}})$.

$\mathsf{Recover}(\mathsf{st}, \mathsf{hint}_c, \mathsf{ans}) \to d \in \mathbb{Z}_p$.

- Parse $(\mathsf{sk}, k) \leftarrow \mathsf{st}$.
- Parse $(\mathsf{hint}_1, \ldots, \mathsf{hint}_{\sqrt{N}}) \leftarrow \mathsf{hint}_c$.
- Parse $(\mathsf{ct}_1, \ldots, \mathsf{ct}_{\sqrt{N}}) \leftarrow \mathsf{ans}$.
- Return $d \leftarrow \mathsf{Dec}(\mathsf{pp}, \mathsf{sk}, \mathsf{hint}_k, \mathsf{ct}_k)$.

Figure 12: The SimplePIR protocol, expressed in terms of linearly homomorphic encryption with preprocessing.

Then, for random LWE matrices $\mathbf{A}_1 \xleftarrow{R} \mathbb{Z}_q^{m \times n}$ and $\mathbf{A}_2 \xleftarrow{R} \mathbb{Z}_q^{\ell \times n}$, DoublePIR is a $(T - O(n \cdot \ell + m), 4\epsilon)$-secure PIR scheme on database size N, over plaintext space $\mathbb{Z}_p$, with correctness error δ.

Proof. We prove correctness and security separately.

Correctness. Consider a client that queries for the database value at index $(i_{\text{row}}, i_{\text{col}}) \in [\ell] \times [m]$. We observe that the Recover routine in DoublePIR essentially runs $\kappa \cdot (n+1) + 1$ instances of SimplePIR's recovery routine, to recover $\kappa \cdot (n+1) + 1$ elements in $\mathbb{Z}_p$, namely:

- one $\mathbb{Z}_p$-element that encodes the database value at index $(i_{\text{row}}, i_{\text{col}})$ (in the "first level of PIR"),
- κn additional $\mathbb{Z}_p$-elements that encode the n $\mathbb{Z}_q$-elements in row i_{row} of the first-level hint matrix, $\mathbf{A}_1^T \cdot \mathsf{db}^T$ (in the "second level of PIR"), and
- κ additional $\mathbb{Z}_p$-elements that encode the single $\mathbb{Z}_q$-element at position i_{row} in the first level answer vector, $\mathbf{c}_1^T \cdot \mathsf{db}^T$ (in the "second level of PIR").

For the recovery to succeed, all $\kappa \cdot (n+1) + 1$ invocations of SimplePIR's recovery must succeed. To bound this probability, we compute the probabilities of two separate events:

- E_1, the event that the recovery of the $\kappa \cdot (n+1)$ elements in the "second level of PIR" fails, i.e., that

$$\begin{bmatrix} \mathbf{h}_1 \\ \mathrm{a}_1 \end{bmatrix} \neq \mathsf{Recomp}\left(\begin{bmatrix} \mathsf{hint}_s \\ \mathsf{ans}_1 \end{bmatrix} \right) \cdot \mathbf{u}_{i_{\text{row}}},$$

  where right-multiplication by $\mathbf{u}_{i_{\text{row}}}$ extracts the i_{row}-th column of the matrix.
- E_2, the event that the recovery of the single element in the "first level of PIR" fails, i.e., that

$$\mathsf{Round}_\Delta(\hat{d})/\Delta \neq \mathsf{db}[i_{\text{row}}, i_{\text{col}}].$$

We make use of the following two propositions:

Proposition E.2. *Under the parameters in Theorem E.1,*

$$\Pr[E_1] \leq \frac{\kappa(n+1)}{\kappa(n+1)+1} \cdot \delta.$$

Proof. DoublePIR's recovery routine computes:

$$\begin{bmatrix} \mathbf{h}_1 \\ \mathrm{a}_1 \end{bmatrix} \leftarrow \mathsf{Recomp}\left(\mathsf{Round}_\Delta\left(\begin{bmatrix} \hat{\mathbf{h}}_1 \\ \hat{\mathbf{a}}_1 \end{bmatrix} \right) /\Delta \right) \in \mathbb{Z}_q^{n+1}.$$

Define $\mathsf{db}_2 = \begin{bmatrix} \mathsf{hint}_s \\ \mathsf{ans}_1 \end{bmatrix} \in \mathbb{Z}_q^{\kappa(n+1) \times \ell}$ and let E be the event that

$$\mathsf{Round}_\Delta\left(\begin{bmatrix} \hat{\mathbf{h}}_1 \\ \hat{\mathbf{a}}_1 \end{bmatrix} \right) /\Delta \neq \mathsf{db}_2 \cdot \mathbf{u}_{i_{\text{row}}}. \tag{4}$$

Construction: DoublePIR. The parameters of the construction are a database size N, database dimensions ℓ and m (such that $\ell \cdot m \geq N$), LWE parameters (n, q, χ), plaintext modulus $p \ll q$, and LWE matrices $\mathbf{A}_1 \in \mathbb{Z}_q^{m \times n}$ and $\mathbf{A}_2 \in \mathbb{Z}_q^{\ell \times n}$ (sampled in practice using a hash function). The database consists of N values in $\mathbb{Z}_p$, represented as a matrix in $\mathbb{Z}_p^{\ell \times m}$. We define the scalars $\kappa := \left\lceil \frac{\log q}{\log p} \right\rceil$ and $\Delta := \left\lfloor \frac{q}{p} \right\rfloor$. We use the helper routines:

- $\mathsf{Decomp}\colon \mathbb{Z}_q^{a \times b} \to \mathbb{Z}_q^{\kappa a \times b}$, which writes each $\mathbb{Z}_q$ element as its base-p decomposition,
- $\mathsf{Recomp}\colon \mathbb{Z}_q^{\kappa a \times b} \to \mathbb{Z}_q^{a \times b}$, which interprets each $\kappa \times 1$ submatrix of its input as a base-p decomposition of a $\mathbb{Z}_q$ element and outputs the matrix of these elements.
- $\mathsf{Round}_\Delta\colon \mathbb{Z}_q^a \to \mathbb{Z}_q^a$, which rounds each entry to the nearest integral multiple of Δ.

$\mathsf{Setup}(\mathsf{db} \in \mathbb{Z}_p^{\ell \times m}) \to (\mathsf{hint}_s, \mathsf{hint}_c)$.

- Compute $\begin{cases} \mathsf{hint}_s \leftarrow \mathsf{Decomp}(\mathbf{A}_1^T \cdot \mathsf{db}^T) \in \mathbb{Z}_q^{\kappa n \times \ell} \\ \mathsf{hint}_c \leftarrow \mathsf{hint}_s \cdot \mathbf{A}_2 \in \mathbb{Z}_q^{\kappa n \times n} \end{cases}$
- Return $(\mathsf{hint}_s, \mathsf{hint}_c)$.

$\mathsf{Query}(i \in [N]) \to (\mathsf{st}, \mathsf{qu})$.

- Write i as a pair $(i_{\text{row}}, i_{\text{col}}) \in [\ell] \times [m]$.
- Sample $\begin{cases} \mathbf{s}_1 \xleftarrow{\text{R}} \mathbb{Z}_q^n, & \mathbf{e}_1 \xleftarrow{\text{R}} \chi^m \in \mathbb{Z}_q^m, \\ \mathbf{s}_2 \xleftarrow{\text{R}} \mathbb{Z}_q^n, & \mathbf{e}_2 \xleftarrow{\text{R}} \chi^\ell \in \mathbb{Z}_q^\ell. \end{cases}$
- Compute $\begin{cases} \mathbf{c}_1 \leftarrow (\mathbf{A}_1 \cdot \mathbf{s}_1 + \mathbf{e}_1 + \Delta \cdot \mathbf{u}_{i_{\text{col}}}) \in \mathbb{Z}_q^m, \\ \mathbf{c}_2 \leftarrow (\mathbf{A}_2 \cdot \mathbf{s}_2 + \mathbf{e}_2 + \Delta \cdot \mathbf{u}_{i_{\text{row}}}) \in \mathbb{Z}_q^\ell, \end{cases}$
  where $\mathbf{u}_{i^*}$ is the vector of all zeros with a single '1' at index i^*.
- Return $(\mathsf{st}, \mathsf{qu}) \leftarrow ((\mathbf{s}_1, \mathbf{s}_2), (\mathbf{c}_1, \mathbf{c}_2))$.

$\mathsf{Answer}(\mathsf{db}, \mathsf{hint}_s, \mathsf{qu}) \to \mathsf{ans}$.

- Parse $(\mathbf{c}_1 \in \mathbb{Z}_q^m, \mathbf{c}_2 \in \mathbb{Z}_q^\ell) \leftarrow \mathsf{qu}$.
- Compute $\begin{cases} \mathsf{ans}_1 \leftarrow \mathsf{Decomp}(\mathbf{c}_1^T \cdot \mathsf{db}^T) \in \mathbb{Z}_q^{\kappa \times \ell} \\ \mathbf{h} \leftarrow \mathsf{ans}_1 \cdot \mathbf{A}_2 \in \mathbb{Z}_q^{\kappa \times n}, \\ \begin{bmatrix} \mathsf{ans}_h \\ \mathsf{ans}_2 \end{bmatrix} \leftarrow \begin{bmatrix} \mathsf{hint}_s \\ \mathsf{ans}_1 \end{bmatrix} \cdot \mathbf{c}_2 \in \mathbb{Z}_q^{\kappa(n+1)}. \end{cases}$
- Return $\mathsf{ans} \leftarrow (\mathbf{h}, \mathsf{ans}_h, \mathsf{ans}_2)$.

$\mathsf{Recover}(\mathsf{st}, \mathsf{hint}_c, \mathsf{ans}) \to d$.

- Parse:
  - $(\mathbf{s}_1 \in \mathbb{Z}_q^n, \mathbf{s}_2 \in \mathbb{Z}_q^n) \leftarrow \mathsf{st}$,
  - $(\mathbf{h} \in \mathbb{Z}_q^{\kappa \times n}, \mathsf{ans}_h \in \mathbb{Z}_q^{\kappa \times n}, \mathsf{ans}_2 \in \mathbb{Z}_q^\kappa) \leftarrow \mathsf{ans}$.
- Compute:

$$\begin{bmatrix} \hat{\mathbf{h}}_1 \\ \hat{\mathbf{a}}_1 \end{bmatrix} \leftarrow \begin{bmatrix} \mathsf{ans}_h \\ \mathsf{ans}_2 \end{bmatrix} - \begin{bmatrix} \mathsf{hint}_c \\ \mathbf{h} \end{bmatrix} \cdot \mathbf{s}_2 \in \mathbb{Z}_q^{\kappa \times (n+1)},$$

$$\begin{bmatrix} \mathbf{h}_1 \\ \mathrm{a}_1 \end{bmatrix} \leftarrow \mathsf{Recomp}\left(\mathsf{Round}_\Delta\left(\begin{bmatrix} \hat{\mathbf{h}}_1 \\ \hat{\mathbf{a}}_1 \end{bmatrix} \right) / \Delta \right) \in \mathbb{Z}_q^{n+1},$$

$$\hat{d} \leftarrow \mathrm{a}_1 - \mathbf{s}_1^T \mathbf{h}_1 \in \mathbb{Z}_q.$$

- Return $d \leftarrow \mathsf{Round}_\Delta(\hat{d})/\Delta \in \mathbb{Z}_p$.

Figure 13: The DoublePIR protocol.

We have that $\Pr[E] \geq \Pr[E_1]$ as $E_1 \subset E$. By rewriting terms:

$$\begin{aligned} \begin{bmatrix} \hat{\mathbf{h}}_1 \\ \hat{\mathbf{a}}_1 \end{bmatrix} &= \begin{bmatrix} \mathsf{ans}_h - \mathsf{hint}_c \cdot \mathbf{s}_2 \\ \mathsf{ans}_2 - \mathbf{h} \cdot \mathbf{s}_2 \end{bmatrix} \\ &= \begin{bmatrix} \mathsf{hint}_s \cdot \mathbf{c}_2 - (\mathsf{hint}_s \cdot \mathbf{A}_2) \cdot \mathbf{s}_2 \\ \mathsf{ans}_1 \cdot \mathbf{c}_2 - (\mathsf{ans}_1 \cdot \mathbf{A}_2) \cdot \mathbf{s}_2 \end{bmatrix} \\ &= \begin{bmatrix} \mathsf{hint}_s \cdot (\mathbf{A}_2 \cdot \mathbf{s}_2 + \mathbf{e}_2 + \Delta \cdot \mathbf{u}_{i_{\text{row}}}) - \mathsf{hint}_s \cdot \mathbf{A}_2 \cdot \mathbf{s}_2 \\ \mathsf{ans}_1 \cdot (\mathbf{A}_2 \cdot \mathbf{s}_2 + \mathbf{e}_2 + \Delta \cdot \mathbf{u}_{i_{\text{row}}}) - \mathsf{ans}_1 \cdot \mathbf{A}_2 \cdot \mathbf{s}_2 \end{bmatrix} \\ &= \mathsf{db}_2 \cdot \mathbf{e}_2 + \Delta \cdot \mathsf{db}_2 \cdot \mathbf{u}_{i_{\text{row}}}. \end{aligned}$$

Comparing this with Equation (4), we see that E happens if and only if $\|\mathsf{db}_2 \cdot \mathbf{e}_2\|_\infty \geq \Delta/2$. The vector $\mathbf{e} := \mathsf{db}_2 \cdot \mathbf{e}_2$ consists of $\kappa \cdot (n+1)$ elements, which we denote as $e_1, \ldots, e_{\kappa \cdot (n+1)}$. As in our correctness proof for SimplePIR, we use a concentration inequality [59, Lemma 2.2][11, Lemma 2.4] on each of these terms: since χ is the discrete Gaussian distribution with variance $\sigma^2 = \frac{s^2}{2\pi}$ for some $s > 0$, for every $j \in [\kappa \cdot (n+1)]$ and for any $T > 0$, it holds that,

$$\Pr\left[\left| e_j \right| \geq T \cdot s \, \|\mathsf{db}_2[j, :]\| \right] < 2\exp(-\pi \cdot T^2),$$

where $\|\cdot\|$ is the Euclidean norm. Note that since we can represent the elements in db_2 in the range $(-p/2, p/2]$, we have that $\|\mathsf{db}_2[j, :]\| \leq \sqrt{\ell \cdot (p/2)^2} = \sqrt{\ell} \cdot p/2$. Therefore, for every $j \in [\kappa \cdot (n+1)]$, for any $T > 0$, it holds that,

$$\Pr\left[\left| e_j \right| \geq T \cdot s \cdot \sqrt{\ell} \cdot p/2 \right] < 2\exp(-\pi \cdot T^2).$$

Now, we take $T = \Delta/(2s \cdot \sqrt{\ell} \cdot p/2)$, and we see that, for every

$j \in [\kappa \cdot (n+1)]$,

$$\Pr\left[\left|e_j\right| \geq \Delta/2\right] < \frac{\delta}{\kappa(n+1)+1}, \text{ as long as}$$

$$2\exp\left(-\pi \cdot \left(\frac{\Delta}{2s \cdot \sqrt{\ell} \cdot p/2}\right)^2\right) \leq \frac{\delta}{\kappa(n+1)+1}.$$

Substituting $\Delta = \lfloor q/p \rfloor$ and $s = \sigma \cdot \sqrt{2\pi}$, we rewrite the second equation as:

$$\lfloor q/p \rfloor \geq \sigma \cdot p \cdot \sqrt{2\ell \cdot \ln\left(\frac{2\kappa(n+1)+1}{\delta}\right)}.$$

By Equation (3), this condition holds. Thus, we have shown that, for each $j \in [\kappa \cdot (n+1)]$, $\left|e_j\right| < \Delta/2$ holds except with probability $\frac{\delta}{\kappa(n+1)+1}$. Therefore, by a union bound, all $e_1, \ldots, e_{\kappa\cdot(n+1)}$ have an absolute value smaller than $\Delta/2$ with probability $\frac{\kappa(n+1)}{\kappa(n+1)+1} \cdot \delta$, which completes the proof. □

Proposition E.3. *Under the parameters in Theorem E.1,*

$$\Pr[E_2|\neg E_1] \leq \frac{1}{\kappa(n+1)+1} \cdot \delta.$$

Proof. If $\neg E_1$ occurs, we know that

$$\begin{bmatrix}\mathbf{h}_1 \\ \mathrm{a}_1\end{bmatrix} = \mathsf{Recomp}\left(\begin{bmatrix}\mathsf{hint}_s \\ \mathsf{ans}_1\end{bmatrix}\right) \cdot \mathbf{u}_{i_{\text{row}}}$$
$$= \begin{bmatrix}\mathbf{A}_1^T \cdot \mathsf{db}^T \\ \mathbf{c}_1^T \cdot \mathsf{db}^T\end{bmatrix} \cdot \mathbf{u}_{i_{\text{row}}}.$$

Therefore,

$$\begin{aligned}\hat{d} &= \mathrm{a}_1 - \mathbf{s}_1^T \cdot \mathbf{h}_1 \\ &= (\mathbf{c}_1^T \cdot \mathsf{db}^T - \mathbf{s}_1^T \cdot \mathbf{A}_1^T \cdot \mathsf{db}^T) \cdot \mathbf{u}_{i_{\text{row}}} \\ &= ((\mathbf{s}_1^T \cdot \mathbf{A}_1^T + \mathbf{e}_1^T + \Delta\mathbf{u}_{i_{\text{col}}}^T) \cdot \mathsf{db}^T - \mathbf{s}_1^T \cdot \mathbf{A}_1^T \cdot \mathsf{db}^T) \cdot \mathbf{u}_{i_{\text{row}}} \\ &= \mathbf{e}^T \cdot \mathsf{db}^T \cdot \mathbf{u}_{i_{\text{row}}} + \Delta\mathbf{u}_{i_{\text{col}}}^T \cdot \mathsf{db}^T \cdot \mathbf{u}_{i_{\text{row}}}.\end{aligned}$$

Thus, we see that Recover outputs the correct database element, $\mathsf{db}[i_{\text{row}}, i_{\text{col}}]$, if and only if $\left|\mathbf{e}^T \cdot \mathsf{db}^T \cdot \mathbf{u}_{i_{\text{row}}}\right| < \Delta/2$. By essentially the same analysis as in Proposition E.2, the event that $\left|\mathbf{e}^T \cdot \mathsf{db}^T \cdot \mathbf{u}_{i_{\text{row}}}\right| \geq \Delta/2$ happens occurs with probability at most $\frac{1}{\kappa(n+1)+1} \cdot \delta$, given that the condition

$$\lfloor q/p \rfloor \geq \sigma \cdot p \cdot \sqrt{2m \cdot \ln\frac{2(\kappa(n+1)+1)}{\delta}}$$

holds, which is true by Equation (3). □

Combining the two propositions, we conclude that DoublePIR's correctness error is upper bounded by

$$\Pr[E_1 \vee E_2] \leq \Pr[E_1] + \Pr[E_2|\neg E_1] \leq \delta.$$

Security. DoublePIR's security follows from the hardness of LWE with a reused LWE matrix [73]. We rely on the following lemma:

Lemma E.4. *Let $N \in \mathbb{N}$ be the database size, $\ell, m \in \mathbb{N}$ be the database dimensions (such that $\ell \cdot m \geq N$), (n, q, χ) be the LWE parameters, and $\mathbf{A}_1 \in \mathbb{Z}_q^{m\times n}$ and $\mathbf{A}_2 \in \mathbb{Z}_q^{\ell\times n}$ be the random LWE matrices used in DoublePIR. For any $i \in [N]$, we define the distribution*

$$Q_i = \{(\mathbf{A}_1, \mathbf{A}_2, \mathsf{qu}_i) : \_, \mathsf{qu}_i \leftarrow \mathsf{Query}(i)\}.$$

If the (n, q, χ)-LWE problem with $\max(\ell, m)$ samples is (T, ϵ)-hard, then any algorithm running in time $T - O(n \cdot \ell + m)$ has success probability at most 2ϵ in distinguishing Q_i from the distribution $\mathcal{D} = \{(\mathbf{A}_1, \mathbf{A}_2, (\mathbf{r}_1, \mathbf{r}_2)) : \mathbf{r}_1 \xleftarrow{R} \mathbb{Z}_q^m, \mathbf{r}_2 \xleftarrow{R} \mathbb{Z}_q^\ell\}$.

Proof. Consider any index $i \in [N]$, and decompose i into the pair of coordinates $(i_{\text{row}}, i_{\text{col}}) \in [\ell] \times [m]$ as done in DoublePIR's Query routine. Let $\mathbf{u}_{i_{\text{col}}} \in \mathbb{Z}_q^m$ denote unit vector i_{col} in $\mathbb{Z}_q^m$ and $\mathbf{u}_{i_{\text{row}}}$ denote unit vector i_{row} in $\mathbb{Z}_q^\ell$. We define the following distribution:

$$\mathcal{H}_i = \{(\mathbf{A}_1, \mathbf{A}_2, (\mathbf{r}_1, \mathbf{A}_2\mathbf{s}_2+\mathbf{e}_2)) : \mathbf{r}_1 \xleftarrow{R} \mathbb{Z}_q^m, \mathbf{s}_2 \xleftarrow{R} \mathbb{Z}_q^n, \mathbf{e}_2 \xleftarrow{R} \chi^\ell\}$$

Now, consider any algorithm $\mathcal{A}$ that runs in time t and distinguishes between distributions $\mathcal{Q}_i$ and $\mathcal{D}$ with probability p. Let p_0, p_1, and p_2 be the probabilities that $\mathcal{A}$ outputs 1 when given samples from $\mathcal{Q}_i, \mathcal{H}_i$, and $\mathcal{D}$ respectively. Thus, by definition, it must hold that $|p_0 - p_2| = p$. From the triangle inequality we see that $|p_0 - p_1| + |p_1 - p_2| \geq p$.

Then, using $\mathcal{A}$ as a subroutine, we build an algorithm $\mathcal{B}_1$ that distinguishes between the (n, q, χ)-LWE problem with m samples and the uniform distribution, in time $t+O(m\cdot n+\ell)$ and with success probability $|p_0 - p_1|$. Because the (n, q, χ)-LWE problem with m samples is (T, ϵ)-hard, if $t \leq T-O(m\cdot n+\ell)$, it must hold that $|p_0 - p_1| \leq \epsilon$. Similarly, we build an algorithm $\mathcal{B}_2$ that distinguishes between the (n, q, χ)-LWE problem with ℓ samples and the uniform distibution in time $t + O(m + \ell)$ and with success probability $|p_1 - p_2|$. Because the (n, q, χ)-LWE problem with ℓ samples is (T, ϵ)-hard, if $t \leq T - O(m + \ell)$, it must hold that $|p_1 - p_2| \leq \epsilon$.

Thus, if $t \leq T - O(m \cdot n + \ell)$, we see that $2\epsilon \geq |p_0 - p_1| + |p_1 - p_2| \geq p$. We have shown that any algorithm $\mathcal{A}$ that runs in time $T - O(m \cdot n + \ell)$ can distinguish between $\mathcal{Q}_i$ and $\mathcal{D}$ with probability at most 2ϵ.

Algorithm 2 $\mathcal{B}_1$, on input $\mathbf{A}_1, \mathbf{b}_1$:

Compute $\mathbf{c}_1 \leftarrow \mathbf{b}_1 + \lfloor q/p \rfloor \cdot \mathbf{u}_{i_{\text{col}}}$.
Sample $\mathbf{s}_2 \xleftarrow{R} \mathbb{Z}_q^n$, $\mathbf{e}_2 \xleftarrow{R} \chi^\ell$.
Compute $\mathbf{c}_2 \leftarrow (\mathbf{A}_2\mathbf{s}_2 + \mathbf{e}_2) + \lfloor q/p \rfloor \cdot \mathbf{u}_{i_{\text{row}}}$.
Output $\mathcal{A}(\mathbf{A}_1, \mathbf{A}_2, (\mathbf{c}_1, \mathbf{c}_2))$.

Algorithm 3 $\mathcal{B}_2$, on input $\mathbf{A}_2, \mathbf{b}_2$:

Sample $\mathbf{r}_1 \xleftarrow{R} \mathbb{Z}_q^m$.
Compute $\mathbf{c}_2 \leftarrow \mathbf{b}_2 + \lfloor q/p \rfloor \cdot \mathbf{u}_{i_{\text{row}}}$.
Output $\mathcal{A}(\mathbf{A}_1, \mathbf{A}_2, (\mathbf{r}_1, \mathbf{c}_2))$.

□

Lemma E.4 states that any algorithm running in time $T - O(n \cdot \ell + m)$ can distinguish the queries made by a client in DoublePIR from random vectors with success probability at most 2ϵ. Therefore, any algorithm running in time $T - O(n \cdot \ell + m)$ can distinguish the queries made by a client in DoublePIR to any pair of indices $i \in [N]$ and $j \in [N]$ with success probability at most 4ϵ. Thus, we have shown that DoublePIR is $(T - O(n \cdot \ell + m), 4\epsilon)$-secure. □

DoublePIR also meets the Q-query security definition that we give in Appendix C.2. A hybrid argument shows that:

Corollary E.5 (Q-query security of DoublePIR)**.** *Consider any two sequences of Q indices, $I, J \in [N]^Q$, and the query distributions, $\mathcal{D}_i$ and $\mathcal{D}_j$, that they induce, i.e.,*

$$\begin{cases} \mathcal{D}_i = \{(\mathbf{A}_1, \mathbf{A}_2, \mathsf{qu}_k) : \_, \mathsf{qu}_k \leftarrow \mathsf{Query}(I_k)\}_{k \in [Q]} \\ \mathcal{D}_j = \{(\mathbf{A}_1, \mathbf{A}_2, \mathsf{qu}_k) : \_, \mathsf{qu}_k \leftarrow \mathsf{Query}(J_k)\}_{k \in [Q]} \end{cases}$$

If the (n, q, χ)-LWE problem with $max(\ell, m)$ samples is (T, ϵ)-hard, then $\mathcal{D}_i$ is $(T - O(Q \cdot (n \cdot \ell + m)), 4Q\epsilon)$-indistinguishable from $\mathcal{D}_j$.

E.3 Additional extensions and optimizations

The SimplePIR optimizations involving fast matrix multiplication algorithms (that we outline in Appendix C.3) apply to DoublePIR as well. We list some further extensions and efficiency improvements to DoublePIR in this section.

Handling database updates. When the database contents change, the server in DoublePIR needs to update its preprocessed hints, hint_s and hint_c (see Remark 3.1). In more detail, when c rows of the database matrix change (for some number $c \geq 0$), the server needs to:

- perform $O(c \cdot m \cdot n + c \cdot \kappa \cdot n)$ operations in $\mathbb{Z}_q$, and
- send $\min(c \cdot \kappa \cdot n, \kappa \cdot n^2)$ elements in $\mathbb{Z}_q$ back to the client.

This update procedure works as follows. In DoublePIR, the server computes the hints as $\mathsf{hint}_s \leftarrow \mathsf{Decomp}(\mathbf{A}_1^T \cdot \mathsf{db}^T)$ and $\mathsf{hint}_c \leftarrow \mathsf{hint}_s \cdot \mathbf{A}_2$. For each row $\mathbf{d} \in \mathbb{Z}_q^m$ of the database that has changed, the server can update the corresponding columns of hint_s to be $\mathbf{u} \leftarrow \mathsf{Decomp}(\mathbf{A}_1^T \cdot \mathbf{d}^T) \in \mathbb{Z}_q^{\kappa n}$. To update hint_c, for each row of the database that has changed, the server would then have to add the outer product between the column vector, $\mathbf{u}$, and the corresponding row of $\mathbf{A}_2$, to hint_c. The server can perform this computation and send the updated hint_c back to the client; or, if fewer than n rows of the database were changed, the server can send only the associated column vectors, $\mathbf{u}$, back to the client (to save on bandwidth) and have the client update its hint_c locally. Again, additions and deletions of rows from the database can be handled similarly.

Reducing the client storage, when the client makes a bounded number of queries. If the client knows ahead of time that it will only make $Q \ll n$ queries, it can reduce its local storage to $Q \cdot \kappa \cdot n$ (rather than $\kappa \cdot n^2$) elements in $\mathbb{Z}_q$. To do so, the client samples Q pairs of LWE secrets $(\mathbf{s}_1, \mathbf{s}_2)$ in advance (e.g., using a pseudorandom function). Then, rather than store hint_c in its entirety, the client precomputes and stores only the information that it will need for recovery: for each of the Q queries it intends to make, the client precomputes $\mathsf{hint}_c \cdot \mathbf{s}_2 \in \mathbb{Z}_q^{\kappa n}$ (where s_2 denotes the the secret key used for the corresponding query). Finally, the client discards hint_c.

In theory, the client storage could be reduced even further by applying the local rounding trick (see Appendix C.3) to the precomputed information, $\mathsf{hint}_c \cdot \mathbf{s}_2$, stored by the client. After doing so, the client would store only a single element in $\mathbb{Z}_q$—namely, $\mathbf{s}_1^T \cdot \mathsf{Recomp}(\mathsf{Round}_\Delta(\mathsf{hint}_c \cdot \mathbf{s}_2))$—for each query it intends to make. However, doing so would require using a much larger value of $\lfloor q/p \rfloor$ to maintain correctness and is thus not profitable in our parameter regime.

F Proof of Proposition 6.2

Proof of Proposition 6.2. Completeness and privacy follow by construction. Soundness follows by Lemma F.1. □

Lemma F.1. *The approximate membership-test data structure $(\mathsf{Setup}, \mathsf{Query})$ on parameters a and k has soundness error at most c and fails with probability $2^{-\lambda}$, for any $c \in (0, 1)$ that satisfies $|\mathcal{U}| \binom{a}{ca} k^{-ca} \leq 2^{-\lambda}$.*
When $k \geq 8$ (and taking $c = 1/2$), we get a data structure with soundness error $1/2$ and failure probability at most $2^{-\lambda}$, as long as $|\mathcal{U}|\, 2^{-a/2} \leq 2^{-\lambda}$.

Proof. Fix any set $S \subseteq \mathcal{U}$ and consider $D \leftarrow \mathsf{Setup}(S)$. Consider any $c \in (0, 1)$ that satisfies $|\mathcal{U}| \binom{a}{ca} k^{-ca} \leq 2^{-\lambda}$.

For string $\sigma \in \mathcal{U} \setminus S$, let Bad_σ be the event, over the random choice of the random oracle, that there is a set of rows $R \subseteq [a]$ of size ca, such that for all $i \in R$, $D_{i, H_i(\sigma)} = 1$. First, we bound the probability, over the choice of the random oracle, that there exists any bad string σ.

In particular, fix a string $\sigma \in \mathcal{U} \setminus S$. Now we analyze the event Bad_σ as follows. Each row of D contains at most $|S| = N$ ones and has kN entries total. Thus, an independently sampled random element in D is non-zero with probability at most k^{-1}. For a particular set $R \subseteq [a]$ of size ca, let $\mathsf{Bad}_{\sigma,R}$ be the event that for all $i \in R$, $D(i, H_i(\sigma)) = 1$. Since the hash functions are modelled as random oracles, we have that $\Pr[\mathsf{Bad}_{\sigma,R}] \leq k^{-ca}$. By a union bound over all possible sets R,

$$\Pr[\mathsf{Bad}_\sigma] = \Pr\left[\bigvee_R \mathsf{Bad}_{\sigma,R}\right] \leq \binom{a}{ca} \cdot k^{-ca}$$

Let $\mathsf{Bad} := \bigvee_{\sigma \in \mathcal{U} \setminus S} \mathsf{Bad}_\sigma$. By a union bound over all possible strings σ, we get that:

$$\Pr[\mathsf{Bad}] \leq |\mathcal{U}| \cdot \Pr[\mathsf{Bad}_\sigma] \leq |\mathcal{U}| \cdot \binom{a}{ca} \cdot k^{-ca}.$$

So, with probability $1 - |\mathcal{U}| \binom{a}{ca} k^{-ca}$ over the choice of the random oracles, there do not exist any bad strings σ.

Now, consider the event ¬Bad (i.e., there exists no bad string). In this case, for any $\sigma \in \mathcal{U} \setminus S$, we know that, with probability at most c (over the random coins of Query), the query algorithm will sample a row i such that $D_{i,H_i(\sigma)} = 1$. Thus, we know that:

$$\mathsf{Accept}(S, \sigma) = \Pr_{i \xleftarrow{R} [a]}[D_{i,H_i(\sigma)} = 1] \leq c,$$

where the probability is taken over the random choice of i by the Query algorithm.

Finally, taking $k \geq 8$ and $c = 1/2$, we see that $\binom{a}{a/2} \leq 2^a$ and $k^{-ca} \leq 2^{-3a/2}$, which completes the proof. □

G Additional implementation material

In Figure 14, we give sample Go code for SimplePIR (without the extensions of Section 4.3). We do not include the code for the following helper functions:

- `MatrixRand`, which takes as input a matrix height, a matrix width, and the modulus to be used for the matrix entries, and returns a uniformly random matrix of the given dimensions.
- `MatrixGaussian`, which takes as input a matrix height, a matrix width, and the modulus to be used for the matrix entries, and returns a matrix of the given dimensions whose entries were sampled from the discrete gaussian distribution (with the appropriate, hard-coded standard deviation).
- `MatrixMul`, which takes as input two matrices and returns their product, over the appropriate field.
- `MatrixAdd`, which takes as input two matrices and returns their sum, over the appropriate field.
- `MatrixSub`, which takes as input two matrices and returns their difference, over the appropriate field.
- `Round`, which takes as input a value and a base and returns the value rounded to the nearest multiple of the base, and then divided by the base.

```
// Data types
type Matrix struct {
  rows uint
  cols uint
  data []uint
}

type Params struct {
  n          uint    // LWE secret dimension
  db_height  uint    // DB height
  db_width   uint    // DB width
  q          uint    // ciphertext modulus
  p          uint    // plaintext modulus
}

// SimplePIR methods
func Init(p Params) Matrix {
  A := MatrixRand(p.db_height, p.n, p.q)
  return A // 'A' is a public parameter of the scheme
}

func Setup(DB Matrix, A Matrix) Matrix {
  H := MatrixMul(DB, A)
  return H // 'H' is the client hint
}

func Query(i uint, A Matrix,
           p Params) (Matrix, Matrix) {
  s := MatrixRand(p.n, 1, p.q)
  err := MatrixGaussian(p.db_width, 1, p.q)
  query := MatrixMul(A, s)
  query.MatrixAdd(err)
  query[i % p.db_width] += (p.q / p.p)
  return s, query // 's' is the client state
}                 // 'query' is the client query


func Answer(DB Matrix, query Matrix) Matrix {
  ans := MatrixMul(DB, query)
  return ans // 'ans' is the server's answer
}

func Recover(i uint64, H, ans, s Matrix,
             p Params) uint64 {
  interm := MatrixMul(H, s)
  ans.MatrixSub(interm)
  noised := ans.data[i / p.db_width]
  denoised := Round(noised, (p.q / p.p))
  return denoised // the recovered database value
}
```

Figure 14: SimplePIR Go code.