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 single-server private information retrieval scheme known to date. SimplePIR's security holds under the learning-with-errors assumption. To answer a client's query, the SimplePIR server performs fewer than one 32-bit multiplication and one 32-bit addition per database byte. SimplePIR achieves 10 GB/s/core server throughput, which approaches the memory bandwidth of the machine and the performance of the fastest *two-server* private-information-retrieval 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 121 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 (7.4 GB/s/core). Finally, we apply our new private-information-retrieval schemes, together with a novel data structure for approximate set membership, to the task of private auditing in Certificate Transparency. We achieve a strictly stronger notion of privacy than Google Chrome's current approach with 13× more communication: 16 MB of download per week, along with 1.5 KB per TLS connection.

1 Introduction

In a private information retrieval (PIR) protocol [19, 48], a database server holds an array of N records. A client wants to fetch record $i \in \{1, \ldots, N\}$ from the server, without revealing the index i that it desires to the server. PIR has applications to systems for private database search [67, 74], metadata-hiding messaging [7, 8], private media consumption [40], credential breach reporting [51, 64, 72, 75], private contact discovery [46], privacy-friendly advertising [9, 39, 44, 69], and private blocklist lookups [47], among others.

Modern PIR schemes require surprisingly little communication: with a single database server and under modest cryptographic assumptions [14, 35, 62], 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 [10], since otherwise the PIR scheme leaks information about which database records the client is *not* interested in. (A number of recent PIR schemes preprocess the database such that the server can answer a query in time sublinear in N, but all known approaches require either client-specific preprocessing [21, 22, 47, 70, 76] or impractically large server storage [10, 13, 15].) Thus, a hard limit on the throughput of PIR schemes—that is, the ratio between the database size and the server time to answer a query—is the speed with which the PIR server can read the database from memory: roughly 12.4 GB/s/core on our machine [73].

In the standard setting, in which the client interacts with a single database server, the performance of existing PIR protocols is far from this theoretical limit: we measure that the fastest prior single-server PIR schemes [59] achieve a throughput of 259 MB/s/core, or 2% of our machine's memory bandwidth, 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 [19], there exist PIR schemes with server-side throughput of up to 11.5 GB/s/core, or 93% of the memory bandwidth (described in Table 1). However, these multi-server PIR 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 suffer from either poor performance—in the single-server setting—or undesirable trust assumptions—in the multi-server case.

In this paper, we present two new single-server PIR schemes that exceed the throughput of all existing single-server PIR protocols and approach the throughput of multi-server ones. In addition, our schemes are relatively simple to explain and easy to implement: our complete implementation of both schemes, available at `github.com/ahenzinger/simplepir`, requires roughly 1,400 lines of Go code, plus 200 lines of C, and uses no external libraries.

More specifically, our first scheme, SimplePIR, achieves a server throughput of 10 GB/s/core, or 81% of the memory bandwidth, though it requires the client to download a relatively large "hint" about the database contents before making its queries. On a database of N bytes, the hint has size roughly $4\sqrt{N}$ KB. 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 7.4 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 [48]. 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 [68], 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 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 [48]: 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 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 implements this auditing step using a solution that provides k-anonymity for $k = 1000$ [26].

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 [11] and approaches based on PIR by keywords [18] 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 0.003 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, which is 10× larger than in some prior work. 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 [19] introduced PIR in the multi-server setting and Kushilevitz and Ostrovsky [48] 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 bits and any dimension parameter $d \in \{1, 2, 3, \dots\}$, their PIR construction has communication roughly $N^{1/d} F^{d-1}$. The

Scheme	Servers	Communication	No per-client storage (on the server)	polylog(n) compute overhead	Max. achievable throughput/core
DPF PIR [12,45]	2	$\log N$	✓	✓	5,381 MB/s
XOR PIR [19]	2	$\sqrt{N}$	✓	✓	6,067 MB/s
XOR PIR fast♣ [19]	2	$\sqrt{N}$	✓	✓	11,797 MB/s
SealPIR [7] ($d = 2$)	1	$\sqrt{N}$	✗	✓	97 MB/s
MulPIR [6] ($d = 2$)	1	$\sqrt{N}$	✗	✓	69 MB/s♢
FastPIR [4]	1	N	✗	✓	215 MB/s
OnionPIR [60]	1	$\log N$	✗	✓	104 MB/s
Spiral family [59]	1	$\log N$	✗	✓	1,314 MB/s
KO [48]+Paillier [63]	1	N^{ϵ}	✓	✗	0.131 MB/s
XPIR [3] ($d = 2$)	1	$\sqrt{N}$	✓	✓	142 MB/s♡
FrodoPIR♠ [25]	1	$\sqrt{N}$	✓	✓	1,256 MB/s
SimplePIR (§4)	1	$\sqrt{N}$	✓	✓	10,305 MB/s
DoublePIR (§5)	1	$\sqrt{N}$	✓	✓	7,622 MB/s

Table 1: A comparison of PIR schemes on database size N and security parameter n. 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 record size. The database and record sizes used are in Appendix A. The throughput is normalized by the number of cores, i.e., divided by two for two-server schemes. ♣This is a non-constant-time implementation—each server's running time depends on its secret input. We include the performance for comparison, though a side-channel-resistant production implementation might not use this optimization. ♢No open-source code available; this throughput is reported in the MulPIR paper [6]. ♡This XPIR throughput is reported by SealPIR [7]. ♠FrodoPIR is concurrent work and is essentially identical to SimplePIR, up to the choice of lattice parameters (see Section 2).

server must perform roughly NF^{d-1} homomorphic operations in the process of answering the client's query.

The Damgård-Jurik [24] cryptosystem has expansion factor $F \approx 1 + \epsilon$ for any constant $\epsilon > 0$, which yields very communication-efficient PIR schemes [53]. It is possible to construct PIR with similar communication efficiency from an array of cryptographic assumptions [14, 17, 29]. 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") [55]. 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 [3] 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 [7] 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 [4] 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 [6], OnionPIR [60], and Spiral [59] additionally use *fully* homomorphic encryption [33] to reduce the communication cost. In Spiral [59], 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 [34], Spiral shows how to decrease the communication cost while keeping the throughput high: up to 259 MB/s on a database of short records. (With long database records, 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 [68], 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 [6, 7, 59, 60] 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.

Concurrent work: FrodoPIR. FrodoPIR [25] is independent concurrent work that constructs a PIR scheme that is essentially identical to SimplePIR. The default configuration of FrodoPIR has communication cost $O(N)$, on database size N, though rebalancing the scheme gives a $O(n\sqrt{N})$-cost, on lattice dimension n, as in SimplePIR. The additional contributions of our work are: the more communication-efficient DoublePIR scheme, our new data structure for private set-membership queries (Section 6), the application to certificate transparency (Section 7), and an optimized implementation of our schemes.

Preprocessing and PIR. The server in our PIR schemes performs some client-independent preprocessing. Prior work uses server-side preprocessing—either one-time [10, 13, 15] or per-client [21, 22, 47, 70, 76]—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 [10, 21, 22, 66]. 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.

Multi-server PIR. In our PIR schemes, the client communicates with a single database server. In multi-server PIR schemes [19], the client communicates with multiple database servers and client privacy holds only as long an attacker cannot compromise some number of them. In Table 1, we give the throughput of an optimized implementation [45] of a two-server PIR scheme based on distributed point functions ("DPF PIR") [12, 36]. We also report the throughput of a $\sqrt{N}$-communication two-server PIR scheme ("XOR PIR") [19]. It is possible to speed these schemes up by roughly 2× if the server's running time can depend on the *Hamming weight* of the client's query vector ("XOR PIR fast"). The downside of this optimization is that it could potentially leak information about one server's secret query vector to another server via timing information, thereby breaking client privacy. Whether such a performance-leakage trade-off is acceptable in practice likely depends on the application scenario.

Hardware acceleration for PIR. Recent work improves the throughput of both single-server [52] and multi-server [41] 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 [54] and Kales, Omolola, and Ramacher [45] 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.

3 Background and definitions

Notation. For a probability distribution χ, we use $x \xleftarrow{R} \chi$ to indicate that x is a 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 $x \in \mathbb{N}$, we let $[x]$ denote the set $\{1, \dots, x\}$. 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 GB to denote 2^{20} and 2^{30} bytes, respectively.

3.1 Learning with errors (LWE)

The security of our PIR schemes relies on the decision version of the learning-with-errors assumption [68]. 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:

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

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 [68] 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 [19, 48]. 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 all of its 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 [47] and a protocol-specific [56] manner. We address how to handle updates in our schemes in the full version of this paper [42].

A PIR-with-preprocessing scheme [10], over plaintext space $\mathcal{D}$ and database size $N \in \mathbb{N}$, consists of four routines, which all take the security parameter as an implicit input:

$\mathsf{Setup}(\mathsf{db}) \rightarrow (\mathsf{hint}_s, \mathsf{hint}_c)$. Given a database $\mathsf{db} \in \mathcal{D}^N$, output preprocessed hints for the server and the client.

$\mathsf{Query}(i) \rightarrow (\mathsf{st}, \mathsf{qu})$. Given an index $i \in [N]$, output a secret client state st and a database query qu.

$\mathsf{Answer}(\mathsf{db}, \mathsf{hint}_s, \mathsf{qu}) \rightarrow \mathsf{ans}$. Given the database db, a server hint hint_s, and a client query qu, output an answer ans.

$\mathsf{Recover}(\mathsf{st}, \mathsf{hint}_c, \mathsf{ans}) \rightarrow d$. Given a secret client state st, a client hint hint_c, and an answer ans, output a 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 [7] and related schemes, the client builds its queries using persistent, long-term cryptographic secrets. We show in the full version of this paper [42] 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 single-server PIR scheme known to date in terms of throughput per second per core (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; we prove its security and correctness in the full version of this paper [42].

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:

- 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 [5].

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 [16, 48, 53] 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 $(i_{\mathsf{row}}, i_{\mathsf{col}}) \in [\sqrt{N}]^2$. Then, the client builds a unit vector $\mathbf{u}_{i_{\mathsf{col}}}$ 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 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

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 $\Delta := \lfloor q/p \rfloor \in \mathbb{Z}$.

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

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

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

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

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

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

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

- Parse $(i_{\mathrm{row}} \in [\sqrt{N}], \mathbf{s} \in \mathbb{Z}_q^n) \leftarrow \mathsf{st}$.
- Compute $\hat{d} \leftarrow (\mathsf{ans}[i_{\mathrm{row}}] - \mathsf{hint}_c[i_{\mathrm{row}}, :] \cdot \mathbf{s}) \in \mathbb{Z}_q$, where $\mathsf{ans}[i_{\mathrm{row}}]$ denotes component i_{row} of ans and $\mathsf{hint}_c[i_{\mathrm{row}}, :]$ 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.

product of database row i_{row} and encrypted unit vector $\mathbf{u}_{i_{\mathrm{col}}}$, or, equivalently, the encrypted database entry at $(i_{\mathrm{row}}, i_{\mathrm{col}})$. 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 [68]. Let (n, q, χ) be LWE parameters. Then, the Regev encryption of a vector $\boldsymbol{\mu} \in \mathbb{Z}_p^m$ 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 $\mathbf{A} \xleftarrow{R} \mathbb{Z}_q^{m \times n}$, secret $\mathbf{s} \xleftarrow{R} \mathbb{Z}_q^n$, and error vector $\mathbf{e} \xleftarrow{R} \chi^m$.

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 the same matrix $\mathbf{A}$ is used to encrypt polynomially many messages, provided that each ciphertext uses an independent secret vector $\mathbf{s}$ and error vector $\mathbf{e}$ [65]. (We give a proof of this fact in the full version of this paper [42].)
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.

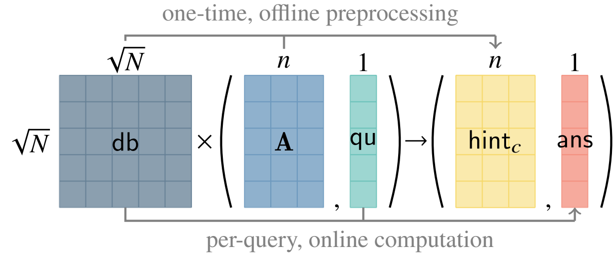

Figure 3: The server computation in SimplePIR. Each cell represents a $\mathbb{Z}_q$ element, and $\times$ denotes matrix multiplication. The server performs the bulk of its work in a one-time preprocessing step. Thereafter, the server can answer each client's query with a lightweight online phase.

In SimplePIR, we leverage these three observations as follows. Consider a client who wishes to retrieve the database entry at $(i_{\mathrm{row}}, i_{\mathrm{col}})$. At a conceptual level, the client's query to the server consists of $\mathsf{Enc}(\mathbf{u}_{i_{\mathrm{col}}}) = (\mathbf{A}, \mathbf{c})$—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 $\mathbf{D} \cdot \mathbf{u}_{i_{\mathrm{col}}} \in \mathbb{Z}_p^{\sqrt{N}}$, 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 $\mathbb{Z}_q$, on lattice dimension $n \approx 2^{10}$ and database size N. 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 each of their queries. Then, the server only precomputes $\mathbf{D} \cdot \mathbf{A}$ 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. Specifically, the server and the clients can derive $\mathbf{A}$ as the output of a public hash function, modelled as a random oracle, applied to a fixed string in counter mode. This saves on bandwidth and storage, as the server and the clients communicate and store only a small seed to generate $\mathbf{A}$.

The security of the SimplePIR construction follows almost immediately from the security of Regev encryption [68] with a reused matrix $\mathbf{A}$ [65], 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 [5]. 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 the full version [42].

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 [4, 7, 8, 40, 59] 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 column i_{col} of the database matrix. So, 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 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 [7, 8], 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. [43] 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-$\sqrt{N}$. If the k records that the client wants to fetch fall into distinct chunks, the client can recover these records by running SimplePIR once on each database chunk. In this case, the hint size remains $n\sqrt{N}$—as in one-query SimplePIR. The communication cost for the client is $k\sqrt{N}$—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 make λ PIR queries to each of the k chunks to achieve failure probability $2^{-\Omega(\lambda)}$ [7, 43]. This optimization saves on server work as long as $\lambda < k$.
- If the client only needs to recover a constant fraction of the k database records, then the client and the server can run this batch-PIR 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 the full version [42].

4.4 Fast linearly homomorphic encryption

In the full version of this paper [42], 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 7.4 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, along with correctness and security proofs, in the full version of this paper [42]. 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 $\kappa = \lceil \log(q)/\log(p) \rceil \approx 4$ (given in the full version [42]). 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 [48] 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., $\left[\mathbf{D} \cdot \mathbf{A} \mid\mid \mathbf{a}\right]^T$). 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 to reduce the total communication. A natural question is whether additional levels of recursion can help, as in standard single-server PIR schemes [48]. After r levels of recursion, the 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, as well as
- per-query upload $r \cdot N^{1/r}$ and download n^{r-1}.

For $r > 2$, the communication is likely too large for databases of interest. An intriguing open question is to construct recursive LWE-based PIR schemes with total communication $n \cdot N^{1/r}$.

5.2 Extensions

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 lattice dimension n, LWE modulus q, and $\kappa = \lceil \log(q)/\log(p) \rceil$, 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 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.) 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 constant fraction among a set of k records from a database of N values in $\mathbb{Z}_p$, on lattice dimension n, LWE modulus q, and $\kappa = \lceil \log(q)/\log(p) \rceil$, 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 *private approximate 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 $\sigma \in S$ without revealing σ to the server. Unlike in private set intersection [32], the server's set S is public. To rule out the trivial solution where the server sends S to the client, we insist on communication sublinear in $|S|$. Our approach is approximate: there is some chance that the client outputs "$\sigma \in S$" when in fact this is not the case. However, this false-positive rate is bounded even when the set S and the string σ are chosen *adversarially*. Looking ahead, our data structure will be at the core of our new scheme for auditing in Certificate Transparency (Section 7).

At a high level, we have the server preprocess its set S into a data structure. Then, the client, holding a string σ, can test whether $\sigma \in S$ by privately reading a few bits of the server's data structure using PIR. The relevant cost metrics are:

- *Number of probes.* How many bits of the server's data structure must the client read?
- *PIR database size.* Over how many bits of the server's data structure does the client perform its private PIR read?
- *Adversarial false-positive rate.* Given an honest server but an an adversarially chosen set S and string σ, what is the probability, *only over the client's secret randomness*, that the client outputs "$\sigma \in S$" when in fact $\sigma \notin S$?

Background: Bloom filters. A Bloom filter [11] is a standard data structure for approximate set membership. A one-hash-function Bloom filter consists of a fixed-length bitstring D and uses a hash function $H\colon \{0,1\}^* \to \{1,\ldots,|D|\}$. Given a set of strings $S \subset \{0,1\}^*$, the setup routine hashes each string $\sigma \in S$ into an index $i \in \{1,\ldots,|D|\}$ and sets the corresponding bit of the data array: $D_{H(\sigma)} \leftarrow 1$. Then, to test whether a string σ is in the set represented by the data structure D, the query algorithm outputs "$\sigma \in S$" if and only if the bit $D_{H(\sigma)} = 1$.

As long as the query string is chosen *independently of the hash function* H, the probability of a false-positive is at most $1/2$ when $|D| \geq 2|S|$. However, when the query string is chosen *adversarially*—as can be the case in our application—an adversary can easily find strings $\sigma \in S$ and $\hat{\sigma} \notin S$ such that $H(\sigma) = H(\hat{\sigma})$. In this case, the one-hash-function Bloom filter will *always* incorrectly output "$\hat{\sigma} \in S$." We present a new data structure—which is a twist on Bloom filters—to address this false-positive issue, without increasing the number of probes or the size of the PIR read required by the query algorithm.

Remark 6.1 (False positives). Some, but not all, applications can tolerate a non-negligible false-positive rate. For example, credential-breach lookups [51, 64, 72] and contact discovery [46] may tolerate false-positive rates as large as 2^{-30}; in contrast, Safe Browsing blocklist checks [37,47] demand a cryptographically negligible false-positive rate, as false positives would cause a legitimate website to be flagged as malicious. In the latter case, other data structures may be more appropriate.

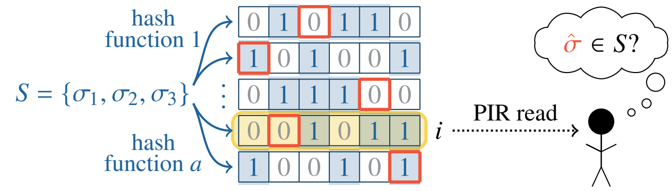

Figure 4: Our data structure for private, approximate set membership with adversarial soundness, when instantiated with a set S consisting of three strings and with $a = 5$ hash functions. We highlight in blue the bits of the data structure that are set, in red the bits that the query string $\hat{\sigma}$ maps to, and in yellow the area covered by the client's PIR read, when the client probes the i-th one-hash-function Bloom filter.

6.1 Our approximate membership test

Our data structure for approximate set-membership, illustrated in Figure 4, is parameterized by integers $a, k \in \mathbb{N}$, a universe of strings $\mathcal{U}$, and a set size N. The data structure consists of a independent one-hash-function Bloom filters [11], each of size kN bits. Crucially, these Bloom filters each use independent hash functions, which are chosen *after* the set S is fixed. In the remainder of this section, we give an informal description of our construction; a formal treatment appears in the full version of this paper [42].

Data-structure setup. The setup algorithm takes as input a set of strings $S \subseteq \mathcal{U}$ of size at most N. The algorithm then chooses a set of a hash functions—one per Bloom filter—and inserts each string in S into each of the a one-hash-function Bloom filters defined by these hash functions. (In practice, we would use a salted hash function with a different salt per filter.)

Query algorithm. Given a query string σ, the query algorithm chooses an index $i \xleftarrow{R} [a]$ at random, and outputs the result of querying the i-th one-hash-function Bloom filter on string σ.

Our data structure has the following properties:

Correctness. For any set $S \subseteq \mathcal{U}$ and any query string $\sigma \in S$, the query algorithm always returns "$\sigma \in S$."

Adversarial false-positive rate 1/2. For any set $S \subseteq \mathcal{U}$ of size at most N, for a random choice of the hash functions used in the data structure, and for any query string $\hat{\sigma} \notin S$—which can depend on the hash functions—the data structure incorrectly returns "$\hat{\sigma} \in S$" with probability at most $1/2$ (taken over the query algorithm's randomness), for an appropriate choice of the parameters a and k. In the full version [42], we prove:

Proposition 6.2: For all $\lambda \in \mathbb{N}$, on parameters $k \geq 8$ and $a \geq 2(\log(|\mathcal{U}|) + \lambda)$, our approximate set-membership data structure has adversarial false-positive rate at most $1/2$. The construction fails with probability $2^{-\lambda}$, over the choice of the Bloom filters' hash functions, modeled as independent random *oracles. Concretely, on* $|\mathcal{U}| = 2^{256}$*, taking* $a = 768$ *and* $k = 8$ *gives false-positive rate* $1/2$ *and failure probability* 2^{-128}.

PIR compatibility. To *privately* test whether a string is in the set, the client can perform a PIR read over only a small fraction of the data structure. More specifically, the query algorithm probes a single bit in one of the Bloom filters. The client can reveal which Bloom filter it wants to probe to the server, as this *does not depend on the query string*. So, while the entire data structure consists of akN bits, the client can execute a private set-membership test with a PIR read over only kN bits.

6.2 Related approaches and comparison

We now compare our solution to other data structures for private approximate set-membership, given in Table 5. One natural alternative would be to use a *single* one-hash-function Bloom filter. (In contrast, our construction uses $a \approx 768$ one-hash-function Bloom filters.) However, this approach is *not* sound in our adversarial setting: as the data structure (including its hash function) is public, an adversary can trivially find a string that causes the query algorithm to *always* return a false-positive result. We can address this issue by using a Bloom filter with $O(\lambda)$ hash functions, which gives security against $2^{O(\lambda)}$-time attacks (where $\lambda \approx 128$ is a security parameter). Unfortunately, the query algorithm of such a data structure is roughly $\lambda\times$ more expensive than ours in terms of both (1) the number of probes and (2) the size of the PIR read required for a private query.

Adversarial Bloom filters [20, 31, 61] provide the false-positive guarantees we require, but do not naturally support private reads via PIR. In particular, they have the client send its string σ to the server; the server then applies a pseudorandom function to σ to determine which bits to probe. It is not clear how to use such a data structure in our setting without relatively expensive general-purpose multi-party computation schemes.

Another approach to private set membership has the client and the server execute a PIR by keywords protocol [18]. On security parameter λ, the server stores a λ-bit hash of each string in its set in a hash table. Thereafter, the client can perform PIR over this hash table to check if a string is present. While the client probes only few locations of the hash table, its PIR read must cover the entire table, or roughly $3\lambda N$ bits.

Finally, prior work constructs other data structures for approximate set membership [27, 38], offering different performance trade-offs. Combining our ideas for efficiently tolerating false positives in an adversarial setting with such data structures is an intriguing direction for future work.

7 Application: Auditing in Certificate Transparency

We now apply our new PIR schemes (Sections 4 and 5), along with our set-membership data structure (Section 6), to solve the problem of privately auditing signed certificate timestamps in deployments of Certificate Transparency [49, 50, 57].

	Probes	*PIR size*	*Adv. false-positive rate*
PIR by keywords [18]	2	$3\lambda N$	0
Standard Bloom filter [11]	$O(\lambda)$	$O(\lambda N)$	0
1-hash-fn Bloom filter [11]	1	$2N$	1 (insecure)
This work	1	$8N$	$1/2$

Table 5: Private set-membership data structures, for sets of N elements from universe $\mathcal{U}$, on security parameter λ. The data structures may fail with probability $|\mathcal{U}|\,2^{-\lambda}$, over their random choice of hash functions, modeled as random oracles.

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., Chrome and Safari both require SCTs from three distinct log operators). 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 a certificate authority has mistakenly or maliciously issued a certificate for their domain. In this setting, the log contents are public; related work investigates scenarios where this is not the case, as in end-user key distribution [58].

SCT auditing. To keep the logs honest, some party in the system must verify that the 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, the client would like to verify that C 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, the set of SCTs that a client sees reveals information about the client's browsing history: the fact that a client has seen an SCT for `example.com` reveals that the client has visited `example.com`. Thus, to protect its privacy, the client should not reveal which SCTs it has seen to the log operators or to any other entity.

Google's recent solutions for SCT auditing [26, 71] 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 to investigate further. Prior work shows how this can be done while keeping the SCT in question hidden [30].

A privacy-protecting solution for SCT auditing must allow the client to test whether its SCT appears in the auditor's set, *without* revealing its SCT to the auditor. This is a private set-membership problem [72]. While on its surface this problem resembles other applications of PIR in the literature [45, 54], the fact that many clients engage in the protocol with the same auditor means that we can tolerate false positives. That is, it is acceptable for a client to incorrectly believe that an SCT is in the auditor's set, since over many clients we can expect that missing SCTs are eventually identified. To summarize, we require the following properties, which we state only informally:

- **Correctness with false positives.** When an honest client holding string σ, chosen independently of the client's secret randomness, interacts with an honest auditor holding set S:
  - if $\sigma \in S$, then the client always outputs "valid," and
  - if $\sigma \notin S$, then the client outputs "valid" with probability at most $1/2$, over the choice of the client's randomness.
- **Privacy for the client.** When an honest client interacts with a malicious auditor, the auditor learns nothing about the client's private input string σ.

We do not require correctness to hold against a malicious auditor; such an auditor could trivially lie about its set of SCTs.

System parameters. There are roughly five billion active SCTs in the web today [26]. Roughly six million of these are added or removed each day as certificate authorities issue certificates and as certificates expire [1]. Google Chrome's current proposal for SCT auditing has a *false-positive rate* of essentially zero: when a client audits an SCT, it correctly learns whether the SCT is valid. However, Chrome's proposal has a *detection rate* of $1/1000$: the Chrome client randomly samples 0.1% of the SCTs associated with its TLS connections, and audits only this small fraction of all SCTs [26]. This random sampling reduces the amortized cost of auditing by $1000\times$, but also reduces the chance that any single auditing client catches a cheating log. Still, across many auditing clients, this randomized SCT auditing catches—with high probability—widely distributed invalid SCTs. After 1000 clients observe an invalid SCT, in expectation one will audit it and implicate the cheating log.

Existing approaches. Two notable proposals for SCT auditing—which do not provide cryptographic privacy—are:

Opt-out SCT auditing. Chrome's current approach [26] has the client reveal the first 20 bits of the hash of its SCT to the auditor. The auditor replies with all ≈ 1000 SCTs in its set that match the 20-bit prefix. This method achieves k-anonymity for $k = 1000$, i.e., it leaks that the client visited one of 1000 sites.

Anonymizing proxy. The client could use proxy servers, such as in Tor [28], to send its SCT to the auditor anonymously [23]. This mechanism is susceptible to timing attacks, which could allow the auditor to deanonymize particular clients.

7.2 Our approach

We propose a new scheme for SCT auditing that achieves cryptographic privacy. The deployment is as follows:

1. Auditor: Data-set construction. The auditor prepares an approximate set-membership data structure holding all SHA256 hashes of all N active SCTs. This data structure consists of $a = 768$ arrays, each $8N$ bits in length, and has false-positive rate $\epsilon = 1/2$ (Proposition 6.2). Then, the auditor runs the PIR Setup routine on each of these a arrays, producing a PIR hints.

2. Client: Hint download. The client chooses a secret, random index $i^* \xleftarrow{R} [a]$ and downloads the i^*-th hint from the auditor, revealing i^* to the auditor in the process. Whenever the client wants to test whether some SCT appears in the auditor's set, the client can now read a single bit from the auditor's i^*-th array. The probability that a cheating log can trick the client into accepting an invalid SCT is at most ϵ, the false-positive rate of the underlying set-membership data structure.

If the client audits an f-fraction of all of its TLS connections, the detection rate is $f \cdot (1-\epsilon)$. In our deployment, we take $f = 1/500$ and $\epsilon = 1/2$. This choice gives an overall detection rate of $1/1000$, matching that of Chrome's current approach [26].

3.Client and auditor: SCT lookup via PIR. Each time the client decides to audit an SCT, the client computes the bit of the auditor's i^*-th array that it needs to check to verify the SCT's validity. The client reads this bit privately by running the PIR protocol's online phase with the auditor, over the i^*-th array.

In this approach, the client reuses the same secret index i^* for multiple SCT lookups. As a result, the events that a client fails to detect invalid SCT A and invalid SCT B are correlated, whereas in Chrome today these events are independent. However, in our approach, the probability that a client catches the *first* invalid SCT that it looks up remains $1 - \epsilon = 1/2$. As such, the client will catch *at least one* invalid SCT with probability 1/1000 and will thus implicate a cheating log with these odds. Any log that cheats more than 1000 clients will be caught in expectation.

Database updates. In our proposal, the client holds a PIR hint that depends on the auditor's set of active SCTs. Whenever this set changes, which happens continuously as certificates are issued and expire, the auditor must update its set-membership data structure. Without extra engineering, the client would have to download a fresh PIR hint from the auditor each time.

Our approach has the client download a fresh hint only periodically—once per week, for example. The auditor specifies the range of *certificate issue dates* that each hint covers. When the client decides to audit an SCT, it checks whether its current hint covers the issue date of the SCT in question. If so, the client tests the SCT's validity; if not, the client caches the SCT so as to test its validity the next time it downloads a hint.

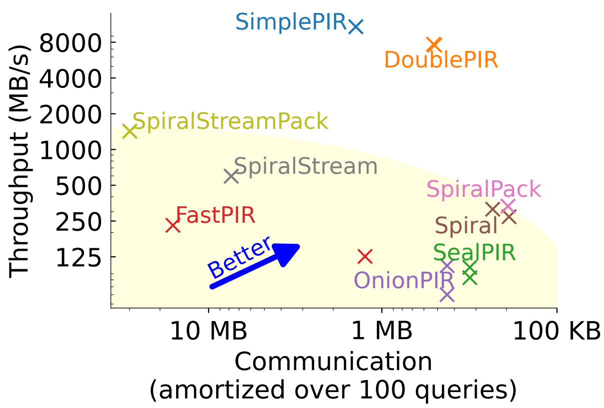

Figure 6: Throughput vs. per-query communication, on a 1 GB database. For each PIR scheme, we display the communication and the corresponding throughput for two choices of entry size: one that maximizes throughput, and another that minimizes communication. (For schemes displayed only once, both entry sizes are the same.) The communication cost is the total (i.e., offline and online) communication, amortized over 100 queries. We highlight prior work in yellow .

In this way, the client eventually audits its full random sample of SCTs, but reuses each hint for multiple SCT lookups. The server must now store multiple versions of the database, which is relatively inexpensive; in a large-scale deployment, one or more physical servers could hold each version in memory.

8 Evaluation

Implementation. We implement SimplePIR in fewer than 1,200 lines of Go code, along with 200 lines of C, and DoublePIR in 210 additional lines of Go code. Our code does not rely on any external libraries and is published under the MIT open-source license at `github.com/ahenzinger/simplepir`.

We use the appropriate data types to natively support operations over $\mathbb{Z}_q$ (e.g., `uint32` for $q = 2^{32}$). 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 10% 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 scheme, on a database roughly 1 GB in size, 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 10. We confirm that these throughputs are indeed the best achievable by measuring each scheme's throughput on each entry size in Figure 7.

	Communication				Throughput
	Offline (MB)		Online (KB)		
	Up.♣	Down.♠	Up.	Down.	(MB/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	121	121	121	10,138
DoublePIR (§5)	0	16	313	32	7,622

Table 8: PIR scheme performance on a database of $2^{33} \times$ 1-bit entries. We highlight in green cells that are within 5× of the best, and in red cells that are within 5× of the worst, in their respective columns. (We leave uncolored cells that are within 5× of the best *and* worst.) We automatically "re-balance" schemes without an automatic parameter selection tool (SealPIR, FastPIR, and OnionPIR), by executing them 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 (see Table 10). ♣The offline upload is equal to the per-client server storage. ♠The offline download is equal to the client storage. ★The throughput here is slightly higher than in Table 1 due to variance in the measurements.

SimplePIR and DoublePIR achieve throughputs of 10.0 GB/s and 7.4 GB/s respectively, which is roughly 8× faster than the best prior single-server PIR scheme designed for the streaming setting (SpiralStreamPack) and 30× faster than the best prior single-server PIR scheme designed for databases with short entries (Spiral). SimplePIR and DoublePIR exceed the *per-server* throughput[1] of some prior two-server PIR schemes: two-server PIR from DPFs [45] has a throughput of 5.3 GB/s/core. Finally, we benchmark the throughput of performing only XORs over a database to provide a hard upper bound on the speed of linear-work, two-server PIR [10,19]. When each server performs a linear scan of XORs over the database, two-server PIR's throughput is 5.9 GB/s/core. When each server performs a linear scan of XORs over *a random half* of the database, two-server PIR's throughput is 11.5 GB/s/core—but this requires a non-constant-time implementation (see discussion in Table 1).

Communication. In Figure 7, 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

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

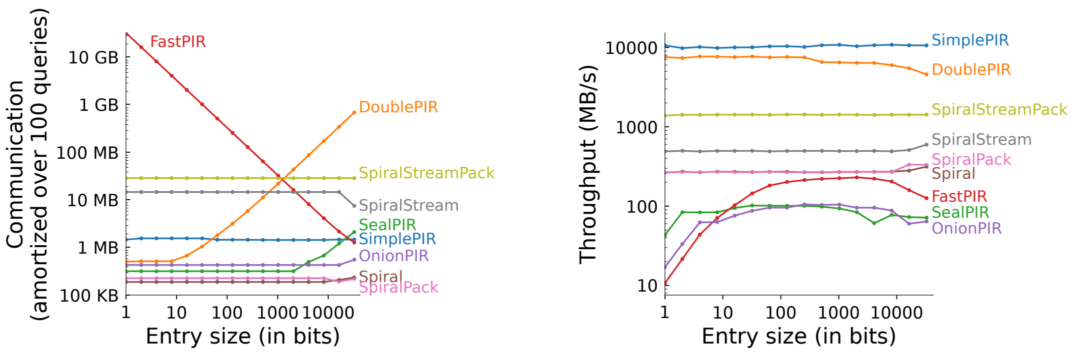

Figure 7: Throughput and per-query communication for each PIR scheme, on a 1 GB database with entries of increasing size. The per-query communication cost is the total (i.e., offline and online) per-query communication, amortized over 100 queries.

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 6, 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 7). 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 6 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.

Comparison on a database of $2^{33} \times 1$-bit entries. In Table 8, 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 (9.9 GB/s and 7.4 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 queries, using the AWS costs for compute (\$$1.5 \cdot 10^{-5}$/core-second) and data transfer out of Amazon EC2 (\$0.09/GB). SimplePIR's per-query cost is \$$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 requires megabytes of online upload, 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 fall into a distinct database chunk, as discussed in Section 4.3) and we derive the expected "successful" throughput—that is, the throughput measured when the server answers that number of queries at once, with a single pass over the database.

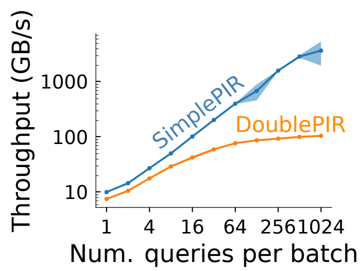

Figure 9: Effective PIR throughput (database size × queries per second), with increasing batch sizes and a fixed-size hint, on a database consisting of $2^{33} \times 1$-bit entries. The shading displays the standard deviation.

Figure 9 shows that 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 $k \geq 16$ and 1000 GB/s on batch size $k \geq 256$. DoublePIR achieves a throughput over 50 GB/s for $k \geq 32$; when $k \geq 256$, the throughput plateaus at roughly 100 GB/s, as the second level of PIR becomes a bottleneck.

In the full version of this paper [42], we give additional benchmarks that measure the server preprocessing time, the client time, and the non-amortized communication of our new PIR schemes, along with tables containing the data displayed in Figures 6, 7 and 9.

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 microbenchmarks in Section 8.1 show 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 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 1.3 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 0.003 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 $\$4 \cdot 10^{-8}$ per TLS connection. For a typical client making 10^4 TLS connections per week [2], we expect this cost to be roughly \$0.1 per year. Since a typical client makes only about 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 the full version of this paper [42].

By comparison, Chrome's SCT auditing scheme [26] provides only k-anonymity for $k = 1000$: 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, negligible server computation, and no client storage (unless the client caches popular SCTs). Again assuming a client making 10^4 TLS connections per week [2], 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 per-core throughput of single-server PIR can approach the memory bandwidth of the machine and the performance of two-server PIR. Two exciting directions remain open: one is to reduce our schemes' communication; another is to combine our ideas with those of *sublinear*-time PIR [21,22] to reduce the computation 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. We thank Yuval Ishai and Matan Hamilis for discussing how best to compare against two-server PIR schemes. Yuval also helpfully suggested the "XOR PIR fast" construction discussed in Table 1. 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] Merkle town. `https://merkle.town/`.

[2] 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.

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

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

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

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

[7] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. PIR with compressed queries and amortized query processing. In *S&P*, 2018.

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

[9] Michael Backes, Aniket Kate, Matteo Maffei, and Kim Pecina. ObliviAd: provably secure and practical online behavioral advertising. In *S&P*, 2012.

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

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

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

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

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

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

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

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

[18] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. Cryptology ePrint Archive, Report 1998/003, 1998.

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

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

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

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

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

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

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

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

[27] Peter C. Dillinger and Stefan Walzer. Ribbon filter: practically smaller than bloom and xor. *CoRR*, abs/2103.02515, 2021.

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

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

[30] Saba Eskandarian, Eran Messeri, Joseph Bonneau, and Dan Boneh. Certificate Transparency with privacy. In *PETS*, 2017.

[31] Mia Filić, Kenneth G. Paterson, Anupama Unnikrishnan, and Fernando Virdia. Adversarial correctness and privacy for probabilistic data structures. *CCS*, 2022.

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

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

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

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

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

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

[38] Thomas Mueller Graf and Daniel Lemire. Xor filters: Faster and smaller than bloom and cuckoo filters. *ACM J. Exp. Algorithmics*, 2020.

[39] Matthew Green, Watson Ladd, and Ian Miers. A protocol for privately reporting ad impressions at scale. In *CCS*, 2016.

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

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

[42] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. Cryptology ePrint Archive, Paper 2022/949, 2022.

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

[44] Ari Juels. Targeted advertising ... and privacy too. In *CT-RSA*, 2001.

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

[46] Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. Mobile private contact discovery at scale. In *USENIX Security*, 2019.

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

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

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

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

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

[52] 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.

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

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

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

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

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

[58] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: Bringing key transparency to end users. In *USENIX Security*, 2015.

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

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

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

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

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

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

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

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

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

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

[69] Sacha Servan-Schreiber, Kyle Hogan, and Srinivas Devadas. AdVeil: A private targeted-advertising ecosystem. Cryptology ePrint Archive, Report 2021/1032, 2021.

[70] 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.

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

[72] 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.

[73] Vish Viswanathan, Karthik Kumar, Thomas Willhalm, Blazej Filipiak Patrick Lu, and Sri Sakthivelu. Intel Memory Latency Checker. `https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html`.

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

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

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

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 10, 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.

	Database size N	$\times$ d	**Max. achievable throughput/core**
Prior two-server PIR			
DPF PIR [45]	2^{25}	32 B	5,381 MB/s*
XOR PIR	2^{33}	1 bit	6,067 MB/s*
XOR PIR fast	2^{33}	1 bit	11,797 MB/s*
Prior single-server PIR			
SealPIR [7]	2^{22}	288 B	97 MB/s
MulPIR [6]	10^{5}	40 KB	69 MB/s†
FastPIR [4]	2^{20}	1024 B	215 MB/s
OnionPIR [60]	2^{15}	30 KB	104 MB/s
Spiral [59]	2^{14}	100 KB	353 MB/s
SpiralPack [59]	2^{15}	30 KB	303 MB/s
SpiralStream [59]	2^{15}	30 KB	518 MB/s
SpiralStreamPack [59]	2^{15}	30 KB	1,314 MB/s
FrodoPIR [25]	2^{20}	1 KB	1,256 MB/s
This work (single-server PIR)			
SimplePIR	2^{20}	1 KB	10,305 MB/s
DoublePIR	2^{33}	1 bit	7,622 MB/s

Table 10: Maximal throughput measured for each PIR scheme, on databases of size roughly 1 GB, 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.