Distributed PIR: Scaling Private Messaging via the Users' Machines

Elkana Tovey[*]
Hebrew University
Jerusalem, Israel
elkana.tovey@mail.huji.ac.il

Jonathan Weiss[*]
Hebrew University
Jerusalem, Israel
jonathan.weiss1@mail.huji.ac.il

Yossi Gilad
Hebrew University
Jerusalem, Israel
yossigi@cs.huji.ac.il

Abstract

This paper presents a new architecture for metadata-private messaging that counters scalability challenges by offloading most computations to the clients. At the core of our design is a distributed private information retrieval (PIR) protocol, where the responder delegates its work to alleviate PIR's computational bottleneck and catches misbehaving delegates by efficiently verifying their results. We introduce DPIR, a messaging system that uses distributed PIR to let a server storing messages delegate the work to the system's clients, such that each client contributes proportional processing to the number of messages it reads. The server removes clients returning invalid results, which DPIR leverages to integrate an incentive mechanism for honest client behavior by conditioning messaging through DPIR on correctly processing PIR requests from other users. The result is a metadata-private messaging system that asymptotically improves scalability over prior work with the same threat model. We show through experiments on a prototype implementation that DPIR concretely improves performance by 3.25× and 4.31× over prior work [1, 4] and that the performance gap grows with the user base size.

1 Introduction

Many messaging services like Whatsapp, Telegram, and Signal use end-to-end encryption to safeguard message content from eavesdroppers and corrupt servers. However, these services often lack metadata protection. The service provider (or eavesdropper) sees details like which users communicate and the timing of their communication and learns much about the conversation itself. Communication metadata helps messaging service providers build extensive profiles about their users and their conversations [24, 25, 37]. Moreover, former intelligence agency directors have famously stated that monitoring the communication's metadata obviates the need to read its content [22, 40, §7]. The research community has invested efforts in designing systems for metadata-private communication. A crucial challenge in all such designs is scalability since supporting a larger user base allows hiding a user's communication in a larger crowd and enables better privacy for all users [16, 20].

Recently, metadata-private messaging systems based on Private Information Retrieval (PIR) [8] were proposed [1, 5]. These designs provide the strongest privacy guarantee we could hope to achieve: they allow a recipient to read messages from a server without revealing any information about which messages they read. They trust the server only for service *availability*, i.e., storing messages and responding to requests. However, even if the attacker controls the system's server and all other clients they cannot breach users' *privacy*, i.e., they cannot learn any information about which users exchange messages. This guarantee comes with a major scalability challenge: the server must process all messages in its database for every recipient. Moreover, with a growing user base, the number of users sending messages also grows, which increases the database size a server needs to process per recipient. The compounding of linear work (in the database size) per recipient, and linear increase in the database size per sender leads to *quadratic computational cost* in the number of users. Current designs for PIR-based communication systems thus suggest using a cluster of powerful servers that process the users' messages [1]. However, since the number of servers, for maintaining any level of performance, rapidly grows with more users, scaling deployments of these designs is problematic in practice. (E.g., would an organization be willing/able to deploy 6 000 – 8 000 processing cores for serving 32k users as the evaluation in [1] uses? would their administrator be willing to deploy servers with $24k - 32k$ cores if the user base doubled to $64k$ users?)

This paper addresses the scalability challenge by introducing DPIR, a communication architecture based on PIR that takes a new approach for scalability: rather than deploying many more servers, it distributes the servers' work to the clients with a novel distributed PIR protocol. Intuitively, since PIR ensures privacy against a rogue server, then having the server distribute its work does not create a privacy risk (we formally prove this intuition). Thus, the more clients there are, the more processors DPIR has for handling its growing workload. We build DPIR and compare it against the state-of-the-art systems.

The key challenge in designing DPIR was ensuring *availability*, i.e., delivering correct responses, even when the server outsources work to misbehaving clients. Since the server outsources work to all users in the system, it must efficiently ensure that honest users can read messages even when the clients processing their requests return bogus answers for their share of the computation (or do not return any response). As the number of clients grows, the potential for such clients increases as well, therefore, solving this challenge is crucial when considering DPIR's scalability goal.

At first glance, detecting misbehaving clients seems easy to solve. Two friends exchanging messages could use end-to-end authentication for their messages so that the recipient could detect and complain to the server when she receives an invalid message. However, sending such a complaint is detrimental to DPIR's privacy goal: a rogue server can change one message in its database and then see which recipient complains. The server then learns that the complaining user was interested in reading *that* message. Thus, to avoid this crucial problem, verification of the clients' work must be done without feedback from the recipient.

In DPIR, we found an efficient way to perform such verification. We leverage the fact that PIR processing is an instance of

[*]Elkana and Jonathan contributed equally to this work

matrix multiplication and outsource this computation to the clients. DPIR's server assembles the product matrix from the clients' responses and efficiently validates them using Freivalds' probabilistic algorithm [19]. Applying Freivalds' algorithm in the context of PIR's matrix multiplication is non-trivial since one matrix of the product's matrices holds the recipients' ciphertext queries, encrypted with a homomorphic encryption scheme, and each recipient uses a different key to encrypt their query. Thus, homomorphic operations involving two queries are ill-defined. This presents a challenge since Freivalds' algorithm involves a computation on all queries. Our insight for handling this challenge is that all ciphertexts are elements of the same algebraic ring, irrespective of which key the recipient has used. We show that it is sufficient to have the underlying ring operations satisfy an associativity property to perform the validation (rather than requiring that homomorphic evaluation on the ciphertexts results in coherent decryption). To the best of our knowledge, this is the first time that ciphertexts encrypted using different keys are evaluated together to generate a coherent result.

DPIR distributes the matrix multiplication work across clients and uses a server to validate their results. Moreover, it distributes the work such that, if validation fails, the output of the verification procedure pinpoints the offending client as soon as it responds. DPIR further leverages this ability to detect misbehaving clients to integrate a natural incentive mechanism for participating in PIR computations: a client must help others read messages to communicate through DPIR. This ensures that the number of processors at DPIR's disposal grows with the users' load.

DPIR's approach of offloading the server's computations to clients results in scaling asymptotically better than the state-of-the-art metadata-private messaging systems with the same privacy goal and threat model, Pung and Addra [1, 5]. Specifically, messaging latency scales with $O(n^{1.5})$, rather than $O(n^2)$ in [1, 5], where n is the number of clients reading and writing messages. Concretely, with about 250 weak clients (dedicating a single processing core) to outsource its work, DPIR provides 4.31× and 3.25× better throughput and latency compared to Pung and Addra (respectively), and our experiments also demonstrate that the relative performance gap grows with clients.

In sum, this work makes the following contributions:

(1) A design of a distributed PIR protocol where the bottleneck of the server computing plaintext-ciphertext matrix product is alleviated by offloading work to the clients.

(2) Novel use of homomorphic encryption, where ciphertexts encrypted under *different* keys are used together to perform the matrix product verification from Freivalds' algorithm.

(3) A scalable private messaging that ensures users' privacy against rogue servers, its prototype and evaluation [44].

2 Related work

Tor [17] is the most popular communication system for hiding metadata, offering excellent scalability, but it does not provide privacy against global adversaries monitoring the network, let alone adversaries controlling all system servers [21, 27, 43]. Sasy and Goldberg recently surveyed approaches for achieving metadata private communication with stronger privacy guarantees [41]. As they point out, onion routing and parallel mixnet designs typically scale well and can handle a large user base [29–31, 45]. However, these techniques require a large portion of honest servers (e.g., 70% or 80%) to ensure privacy, and thus deal with a weaker adversary model than DPIR. Furthermore, the most performant mixnet-based systems [30, 31, 45] allow bounded statistical leakage of information about the communication's metadata. As a result, those designs force users to make a difficult choice about how much privacy they are willing to risk and forces them to stop communication when their "privacy budget" depletes.

Most related to DPIR are systems where the servers are trusted only for availability (facilitating communication) but *untrusted for privacy*. We focus the discussion on PIR-based designs (like DPIR). Other designs are based on DC-nets [12, 14, 39, 47] but are vulnerable to client-side faults, induce very high communication overhead, or require trusting a server for privacy; we elaborate on these designs in Appendix A.

PIR-based designs. In these designs, a rogue server can only prevent communication (e.g., by dropping messages) but learns nothing about which message a user retrieves. The price for this strong privacy guarantee in existing designs comes in the form of scalability (see §1). As a result, in today's designs, supporting a large user base with PIR leads to very high latency. For example, Pung's latency grows quadratically with its user base. With 50 servers, Pung requires 1 minute of computation to deliver 131k messages [5, Figure 7]. Addra has the same asymptotic complexity as Pung but optimizes for small message databases. It uses fewer heavyweight cryptographic operations (compared to Pung) to implement PIR but its query size grows linearly with the size of the database. This makes Addra suitable for a user base of less than 64k [1, bottom of §1]. Moreover, Addra users connect with one another every 5 minutes through Alpenhorn [32], which leaks bounded statistical information, and reduces the security of the overall system, although it uses PIR.

A recent approach for scaling PIR protocols uses database preprocessing, assuming a static database, and amortizes computational costs over many queries [13, 23]. Unfortunately, this approach involves a costly database preprocessing computation, every time the server's database changes. This makes the amortized-cost approach unsuitable for communication systems since updates happen frequently, whenever a user sends a message. In parallel work to DPIR, VeriSimplePIR, which uses an offline-online PIR architecture, introduces a verifiability guarantee ensuring that query answers are consistent with the database [15]. However, its verification mechanism is unsuitable for our use-case since it relies on a costly offline communication phase. A recent theory paper focuses on PIR with dynamic database updates. It achieves sublinear query time by relying on superlinear preprocessing of the database. However, as the authors note in that paper, this construction isn't yet practical [33, §1.5].

Reverse PIR. Riposte [11], Express [18] and Sabre [46] propose *reverse-PIR*, where clients write messages anonymously instead of reading them anonymously to achieve metadata-private communications. Riposte and Express are hindered by their writer-auditing protocols, which have high costs. Sabre cleverly manages to reduce auditing costs and, as a result, outperforms Riposte and Express. Unfortunately, Riposte, Express and Sabre require at least one honest server to maintain user privacy, which is incompatible with DPIR's attacker model, which does not assume an honest server for achieving privacy. Furthermore, Like DPIR, Pung and Addra, all three reverse-PIR systems require the servers to perform linear work in the database size, which burdens the scaling capabilities of these systems.

3 Overview

DPIR allows pairs of users to exchange short messages (e.g., 256B-long to support text messaging) while hiding that they communicate. Users write messages to a server and read messages their friends wrote from that server using PIR. Their clients also serve as workers helping the server process PIR queries. This allows DPIR to scale since the number of workers grows with the user base.

3.1 Communication outline

DPIR requires communicating users to have a shared secret. They use this secret to "dial" each other to coordinate communication (below), and encrypt and authenticate end-to-end the content of messages that they exchange. Establishing this secret between two users happens once and is outside DPIR's scope. Prior work requires a shared secret as well [1, 5, 6, 28–31, 45], and suggests establishing such shared secrets when meeting in person (e.g., by scanning QR codes) or through another metadata-private protocol, such as Alpenhorn's "add friend" [32].

To eliminate leakage of information about the time a (human) user sends or receives messages, DPIR takes the same communication round-based approach as much of the prior work in [1, 5, 6, 11, 12, 18, 28–31, 45, 47]. It divides time into epochs, where users dial one another and then exchange messages with their friends over multiple communication rounds. The server assigns users a dead drop, which is a virtual address to deposit messages they write. Clients then write to their dead drop (overriding the latest message they stored on the server) and use PIR to read a message at every round.

Scaling message exchange in DPIR. In every communication round the clients deposit messages to their dead drops and the server distributes a small portion of these messages to each client. Each client then processes a portion of the PIR queries against the portion of the messages. The clients' processing boils down to matrix multiplication (as we recap in §4). Computing the matrix product is the most computationally intensive part of the PIR protocol: it requires processing all messages for each query, and involves costly homomorphic operations on the ciphertexts that comprise PIR queries. By outsourcing this expensive part to the clients, DPIR alleviates the server-side computational bottleneck from prior work [1, 5].

Instead of computing the PIR responses itself, DPIR's server efficiently validates the clients' outputs and then assembles them into the responses. The validation procedure also pinpoints clients that returned invalid outputs, which the server then boots from the system.

3.2 Goals and threat model

DPIR doesn't require a user to trust the server or other users' clients for privacy. Like other PIR-based communication designs, the server is only trusted for service availability but clients are not (despite offloading the heavyweight processing work to the clients). Next, we formulate our goals.

Privacy. DPIR removes linkability between clients chatting. This includes who communicates with whom, the time they exchange messages and message size. It considers a strong network attacker model that can tap on the communication lines or drop messages. It further assumes that the attacker may control its servers and clients. Attacker-controlled machines can deviate from the protocol to try to learn about the users' communication metadata.

Formally, DPIR provides *relationship unobservability (UO)*, defined by Pfitzmann and Hansen [38]. That is, it is computationally undetectable whether anything is sent out of a set of could-be senders to a set of could-be recipients.

Availability. DPIR allows users to exchange messages as long its server follows the protocol, and despite misbehaving clients from other users. This is crucial since clients are many, and the administrator deploying the system's server may have little control over them. DPIR does not aim to provide availability against a dishonest server, that server could always drop all messages in its database and break connectivity (as in prior work based on PIR-based).

Scalability. In prior PIR-based communication designs, the server performs $O(n^2)$ computations to support n users reading and writing messages. DPIR reduces server-side overhead to $O(n^{1.5})$ by offloading work to clients; the overall work across the entire system, the server and clients, remains $O(n^2)$ but is sharded and performed in parallel across all system participants. We show through experiments (§9) that latency is reduced by 76% – 81% compared to prior work [1, 5] and that the load on each client is reasonable, on the order of a few seconds per communication round even when the client uses just one commodity CPU core.

4 Background

We now give the background on the cryptographic and algorithmic building blocks used in DPIR.

Homomorphic Encryption with BGV. DPIR uses BGV [9], an IND-CPA secure (leveled) fully homomorphic encryption scheme. In BGV, plaintexts are polynomials over the quotient ring $R_t = \mathbb{Z}_t[x]/(x^N + 1)$. These are polynomials of degree no greater than N with integer coefficients modulo t. A BGV ciphertext comprises a sequence of polynomials. Each ciphertext polynomial is of degree no greater than N with integer coefficients mod q, and it belongs to the quotient ring $R_q = \mathbb{Z}_q[x]/(x^N+1)$. The parameters N, q set the level of security for the encryption scheme, and the ratio between q and the parameter $t < q$ trades storing more data in a ciphertext for supporting fewer homomorphic operations on that ciphertext.

PIR as an instance of matrix multiplication. Most PIR protocols can be viewed as an instance of *matrix-vector multiplication.* We describe the classical protocol [26], which we use in DPIR, in matrix multiplication terms following Algorithm 1.

Algorithm 1 PIR allows reading a message from matrix $\mathcal{M}$. The client runs `Query` to compute query vector $\vec{q}$ and keeps the secret *sk* to decrypt the response. The server runs `Respond` which computes the matrix-query product, returning the k^{th} column of $\mathcal{M}$ in ciphertexts. Evaluating this product requires a homomorphic cryptosystem.

```
function QUERY($k$, $num\_cols$)
    $sk, pk \leftarrow$ KeyGen()
    for $i \leftarrow 1$ to $num\_cols$ do
        $b \leftarrow 0$ if $i = k$ else $1$
        $q_i \leftarrow \text{Enc}(pk, b)$
    end for
    return $sk, \vec{q} \leftarrow \langle q_1, \ldots, q_{num\_cols} \rangle$
end function

function RESPOND($\mathcal{M}$, $\vec{q}$)
    return $\mathcal{M} \cdot \vec{q}^T$
end function
```

Algorithm 2 Verifying that $\mathcal{M} \cdot \mathcal{Q} = \mathcal{R}$ for matrices $\mathcal{M}, \mathcal{Q}, \mathcal{R}$ (with failure probability of 2^{-d}). Evaluating the equation in line 4 requires three matrix-vector multiplications. This is more efficient than re-computing the matrix-matrix product $\mathcal{M} \times \mathcal{Q}$ directly.

```
1: function FREIVALDS($\mathcal{M}, \mathcal{Q}, \mathcal{R}, d$)
2:     $\ell \leftarrow \mathcal{M}.RowSize()$
3:     $\vec{c} \overset{R}{\leftarrow} \{0, 2^d - 1\}^\ell$
4:     return $\mathcal{M} \times (\mathcal{Q} \times \vec{c}) - \mathcal{R} \times \vec{c} == \vec{0}$
5: end function
```

#messages	1st PIR query	2nd PIR query	verification
67k	127.21	33.02	4.12
262k	497.79	66.06	7.99
1M	1993.31	132.13	17.14
2M	3990.26	186.92	23.57

Table 1: Single core CPU processing time (in milliseconds) to perform recursive PIR and the amortized cost of verifying a 1st-step PIR response using Freivalds' algorithm.

Consider a server with n messages. The server arranges them in a $\sqrt{n} \times \sqrt{n}$ matrix, denoted by $\mathcal{M}$. In PIR, the client's query is an encryption of a one-hot encoding of the column index of $\mathcal{M}$ where the recipient's message is stored. More formally, the client's PIR query, $\vec{q}$ is a $\sqrt{n}$-long vector of ciphertexts of `0`'s except for one index, where there is an encryption of `1`. Each element in the query vector is encrypted using a semantically secure homomorphic cryptosystem (e.g., BGV). This allows the server to `respond` with the product $\vec{r} = \mathcal{M} \times \vec{q}^T$. Since the query vector contains just a single encrypted `1`, say at index k, the response $\vec{r}$ is a vector where $r_i = \sum_j \mathcal{M}_{i,j} \cdot q_j = \mathcal{M}_{i,k}$; i.e., $\vec{r}$ is the k's column vector of $\mathcal{M}$, which contains the recipient's message.

Since responding to a single PIR query can be seen as a matrix-vector product ($\mathcal{M} \times \vec{q}$), then responding to multiple queries is computing the product of two matrices: The server can arrange the query vectors from clients as column vectors in a $\sqrt{n} \times \#clients$ matrix $\mathcal{Q}$. Column i in the matrix product $\mathcal{R} = \mathcal{M} \times \mathcal{Q}$ is the response for the client that sent the query at column i in matrix $\mathcal{Q}$.

Efficient matrix-product verification. Freivalds' algorithm [19], illustrated in Algorithm 2, receives three input matrices and an integer d, which is the bit length of random numbers the algorithm can generate. In our context, the matrices given are as follows: $\mathcal{M}$, the matrix holding the messages (treated as BGV plaintexts); $\mathcal{Q}$, the matrix whose columns are PIR query vectors; and the claimed matrix product, $\mathcal{R}$. The algorithm performs a randomized check for the matrix product, where the verifier chooses uniformly at random a challenge column vector $\vec{c}$ and tests that $\mathcal{M} \times (\mathcal{Q} \times \vec{c}) - \mathcal{R} \times \vec{c} = \vec{0}$. This check is much more efficient than directly computing $\mathcal{M} \times \mathcal{Q}$ and comparing it against $\mathcal{R}$ (since computing three matrix-vector products is less costly than multiplying the two matrices). However, it introduces a 2^{-d} chance of failure where the verifier might not detect inequality.

Compressed queries and recursive PIR. DPIR leverages a compression optimization for sending PIR queries encrypted through BGV from prior work [4]. With standard BGV parameters and less than $2^{12} \times 2^{12}$ elements in $\mathcal{M}$, this optimization exploits the sparsity of the client's query vector (all but one element are encryptions of 0), enabling a client to send just one ciphertext hiding the index of the non-zero encryption and have the server use BGV's homomorphic operations to expand that ciphertext to the full query vector.

Recursive queries. Another well-known optimization is using a second PIR query vector to retrieve just one element from the PIR result vector rather than an entire column. Each client's PIR query comprises two compressed vectors, the first is a one-hot encoding of the column of the element it wishes to read in ciphertext form (as described above) and the second encodes the row of that element. The server multiplies the client's second query vector by the vector that resulted from multiplying the matrix by the first vector (described above). The output of this multiplication is one ciphertext that contains the element that interests the recipient.

4.1 Identifying the bottleneck

Processing the first PIR step involves computing a matrix-vector product, which induces more polynomial multiplications (representing BGV ciphertexts or plaintexts) than processing the recursive second step, which involves computing the dot product of two ciphertext vectors ($\times O(\sqrt{n})$ more). In Table 1 we benchmark the performance of both PIR protocol steps and Freivalds' verification protocol. We see the scalability challenge, for example, when the number of messages grows from 67k to 2M, the dominating cost of processing the first PIR query grows by 31× (proportionally). In contrast, the costs of the second PIR query and product verification are much lower and increase by 5.5×.

The major performance gap between processing the first and second PIR steps is because multiplying the message matrix by the query vector (first PIR step) induces more polynomial multiplications than multiplying the second PIR query with the result of the first recursive PIR (second PIR step). This efficiency holds even when dealing with a relatively small message matrix of size (e.g., $|messages| = 2^{16}$) and even though the second PIR step requires ciphertext-ciphertext multiplication, which is more CPU-intensive than plaintext-ciphertext multiplication. Thus, by offloading the processing of just the first PIR query, DPIR removes the bottleneck computation from the server. Furthermore, the average cost of verifying a query is much smaller than processing the PIR queries, motivating DPIR's approach of outsourcing the bottleneck computation to clients and verifying their results.

5 Design

In DPIR, users exchange short messages through a server. The communication between the users' clients and the server is over TLS, to ensure that a network attacker cannot spoof or modify messages between them without detection. We next describe how users communicate through DPIR.

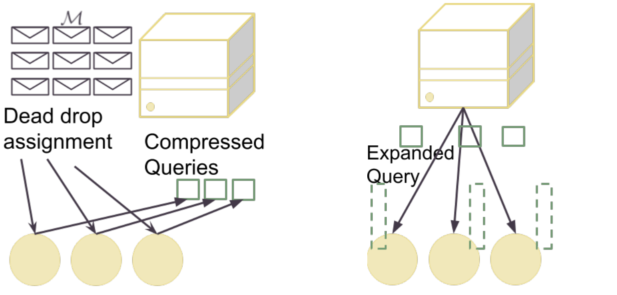
5.1 Epoch setup

In this phase, illustrated in Figure 1, clients connect to the server and register to help in the PIR protocol for the coming epoch. The server assigns registered clients that have "worked" through the previous epoch sequential dead drop addresses. Thus, users must contribute to handling the system's workload in order to communicate through it. The server then arranges the dead drops in a square "message matrix," $\mathcal{M}$ by their address. DPIR uses the BGV homomorphic encryption scheme (see §4). With standard security parameters, a BGV plaintext stores about 10KB of data; so, since users' messages are short texts, the server encodes messages for multiple dead drops into one cell matrix (a BGV plaintext). Given n users that read/write 256B messages (as in prior work [1, 5]), each cell can pack $\lfloor 10KB/256 \rfloor = 39$ messages. Thus, for n clients sending/receiving messages in the epoch (n), $\mathcal{M}$'s dimensions are $\lceil\sqrt{n/39}\rceil \times \lceil\sqrt{n/39}\rceil$. We refer to the matrix dimensions as $m \times m$ to simplify the notation ($m = \lceil\sqrt{n/39}\rceil$).

After clients have dead drops, they dial one another to set up communicating through DPIR in the coming epoch. The (metadata-private) dialing protocol lets two users that share a secret exchange their dead drop addresses. DPIR leverages a known mechanism for dialing. It uses it as a black box, and we defer discussing it to §6 to focus on the mechanisms novel to DPIR's design.

Given the matrix dimensions and a dead drop address (α), the recipient can calculate the cell in the matrix $\mathcal{M}$ that stores the messages from that dead drop (dead drop d maps the cell in row $\lfloor \alpha/m \rfloor$, column $\alpha \bmod m$).

5.1.1 Submitting PIR queries Once dialing completes, each client submits a PIR query to the server (Figure 1a) which will be used for reading messages during the epoch. (Even if a client did not dial a friend, they still send a dummy PIR query to hide they're not exchanging messages.) Although queries stay fixed for the entire epoch, senders and recipients remain unlinkable: The sender always writes to the same dead drop, which looks the same regardless of the recipient, and the PIR query completely hides which of the dead drops the recipient reads. For each epoch, the client generates a private/public BGV key pair and uses the public key to generate a PIR query in compressed form, which targets retrieving from the friend's dead drop in the server's message matrix. DPIR uses the recursive PIR optimization so the query comprises two ciphertexts, one for the column and the other for the row of $\mathcal{M}$ (as described in §4).



(a) At the beginning of the epoch, the server arranges the dead drop addresses in a square matrix. Clients receive a dead drop and upload their PIR queries to the server.

(b) The server distributes compressed PIR queries to clients, which they expand into m-long column vectors that they keep for the entire epoch. Clients process these queries through the epoch.

Figure 1: Epoch setup

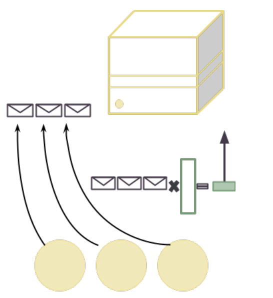
Query distribution. The server splits the clients into disjoint groups of m clients. The assignment of clients to groups is randomized to avoid correlated assignment of faulty clients to the same group (this makes recovering availability more efficient when clients fail, §5.2.1). The server distributes a different subset of m compressed PIR queries to each group (Figure 1b), so all clients in the group handle the same queries, and DPIR covers all queries across all groups (since there are n/m groups and n queries total). In addition to the queries, each client receives matching key-switching keys to expand the queries correctly. These keys are public and are not changing. Thus, they can be stored long on the client's disk.

Every client expands each compressed query vector to a column vector of m ciphertexts (where only one ciphertext is an encryption of 1, and the rest are encryptions of 0, see §4). The client keeps those expanded query vectors for the rest of the epoch since users fetch messages from the same dead drop. Thus, the relatively expensive query-expansion computation (see [4]) is amortized over the length of the epoch. The result is that, after the expansion, each client has an $m \times m$ matrix that covers m PIR query vectors (in its columns). We refer to this matrix as the client group's query matrix (all clients in the same group help process the same set of queries).

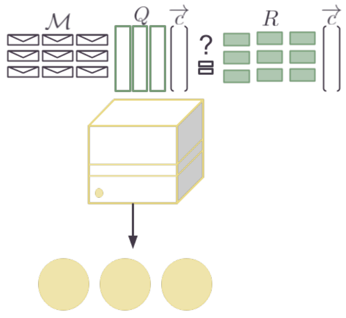
5.2 Communication round

Figure 2 illustrates how clients exchange messages in DPIR. Every round, clients send the server a fixed-size message to store in their dead drop. When users have nothing to send, their clients automatically send a dummy message to hide they don't communicate. Messages are end-to-end encrypted and authenticated using the secret that the sender and recipient share.

In every round, the server distributes to each client a row from its message matrix ($\mathcal{M}$), i.e., m BGV plaintexts per client (Figure 2a). It gives a different row of $\mathcal{M}$ to each client in every group, such that all of $\mathcal{M}$'s rows are distributed across the m clients in each group, as shown in Figure 3. Each client then executes its share



(a) On every communication round, clients write to their dead drops; they then receive a row of $\mathcal{M}$, which they multiply by the expanded queries from the setup phase. They send the product to the server.



(b) The server verifies the clients' results. It does it efficiently using Freivalds' algorithm and without needing the secret query decryption keys. Lastly, the server responds to the recipient.

Figure 2: Communication round

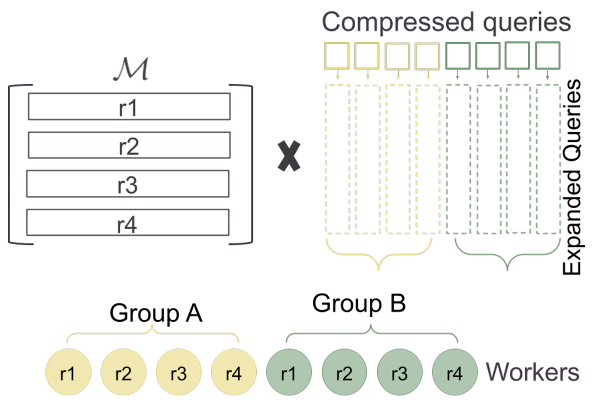


Figure 3: Client-side work. Each client holds a row of the message matrix $\mathcal{M}$ such that the orange and green groups have all the rows across all their clients. Each client also receives their group's queries (in compressed form). In each communication round, it multiplies the message matrix row it receives by the expanded queries.

of the PIR protocol: it multiplies its row of $\mathcal{M}$ by its query matrix, resulting in a row vector with m ciphertexts which it returns to the server.

Since clients in DPIR are untrusted for availability (§3.2), DPIR must deal with clients returning incorrect results. Its server efficiently catches such clients, as illustrated in Figure 2b, by adapting the randomized check from Freivalds' algorithm (§4). Crucially, Freivalds' algorithm verifies complete matrix products, but each of DPIR's clients only responds with just one row vector (the product of its message matrix row and the PIR queries it handles). The server could wait for an entire clients' group, assemble their results vectors as the rows of a response matrix and then run Freivalds' algorithm to check the entire response matrix is correct. However, waiting for an entire group each time before validating a single client induces latency. Instead, we exploit that in DPIR's job distribution across clients, each client affects a different cell in the result vector of Freivalds' randomized check. Thus, it can be verified independently of other clients as soon as a client responds, as we detail next.

On-the-fly verification with Freivalds' algorithm. To understand how DPIR's server verifies the response of an individual client, shown in Algorithm 3, let us observe more closely how the server would verify a batch of responses from an entire client group using Freivalds' algorithm. The server gathers the matrices $\mathcal{M}$ (storing the current round's messages), Q (the PIR queries handled by a clients group in the current epoch), and the result matrix $\mathcal{R}$ (that holds the claimed product $\mathcal{M} \times Q$) where the response from each client in the group is a row. Then, the server chooses an m-long challenge column vector $\vec{c}$ of scalars uniformly at random.

If any of the clients in the group has returned an invalid response, then the server finds that $\mathcal{M} \times (Q \times \vec{c}) - \mathcal{R} \times \vec{c} = \vec{r} \neq \vec{0}$. So, $\vec{r}$ has at least one index, call it i, where $\vec{r}[i] \neq 0$. Crucially, the only client in the group that contributed to $\vec{r}[i]$ was the one handling the i^{th} row of $\mathcal{M}$. The server, therefore, learns that this client is faulty. Moreover, computing r_i only depends on $\mathcal{M}, Q, \vec{c}$ and client i's *individual* response (the i^{th} row of $\mathcal{R}$), see line 7 in Algorithm 3.

Algorithm 3 Verification of a single client's computation. The server caches the computation of vector $\vec{qc}$ for the entire epoch (marked in blue) and computes the scalar mqc in every communication round without waiting for the client's response (marked in violet). When the server reads the response, it multiplies it by the challenge vector $\vec{c}$ and compares against the precomputed mqc (in black).

```
1: function VERIFY_CLIENT(M, Q, client)
       ▷ Compute $\vec{qc}$ once and cache for the entire epoch.
2:     $\vec{c}$ ← random scalar challenge vector of length $m$
3:     $\vec{qc} \leftarrow Q \times \vec{c}$
       ▷ Don't wait for client's response to compute $mqc$.
4:     $\vec{m}$ ← the row of client.id from matrix $\mathcal{M}$
5:     $mqc \leftarrow \vec{m} \cdot \vec{qc}^T$
       ▷ Compute the right side when the client responds.
6:     $\vec{r}$ ← client.read_response()
7:     return $mqc = \vec{r} \cdot \vec{c}$
8: end function
```

Therefore, the server can validate i's response right after receiving it, without waiting for responses from other clients.

Cached verification step. DPIR's server verifies responses from many clients in the same group, this allows caching part of the computation in the verification process. Specifically, the server chooses the challenge vector once per epoch and then computes for each group of clients handling the same $m \times m$ query matrix Q, the matrix-challenge product vector $\vec{qc} \leftarrow Q \times \vec{c}$. Queries are fixed for the entire epoch, so the server can reuse this computation.

Furthermore, the server need not wait for client i's response to compute the left side of the equation in line 7 of Algorithm 3. Since the message matrix, $\mathcal{M}$, is known at the beginning of the round, the server can compute the product of row i in $\mathcal{M}$ and the vector $\vec{qc}$ immediately after distributing client i's job.

5.2.1 *Faulty clients & incentivizing correct work*

The server removes clients that returned invalid responses for the rest of the epoch. It also notifies the clients handling PIR queries from faulty clients to discard them, so they save processing time in future communication rounds. By disconnecting faulty clients and blocking them until the end of the following epoch, DPIR creates an *incentive for correct work*: to communicate through DPIR, clients must continuously provide correct outputs. Deployments of DPIR may include additional penalties such as withholding subscription fees.

DPIR still needs to process the faulty clients' jobs. If the number of faults is small the server processes their jobs. Through benchmarks, we find that processing up to $75m$ of faulty clients' jobs gives comparable overhead to running the verification algorithm plus the second PIR-query processing that the server already does (see §4.1). Setting $75m$ as a maximal threshold for a server-based recovery, therefore, would not dominate the server's processing time. Otherwise, if there are many faulty clients (e.g., $> 75m$), the server redistributes their work across other clients. It does so efficiently as follows: DPIR takes advantage of clients that have the same message matrix row as the faulty client (there is one such client in every group). The server distributes the PIR queries (in compressed form) that the faulty client was supposed to handle evenly across those clients, to share its job's load across many clients. DPIR's hybrid between server/clients-based recovery avoids frequent redistribution in case a few clients fail each time.

Cache update. One detail about work redistribution is that the cached vector $\overrightarrow{qc}$ that the server keeps must be updated for the clients that receive new work: for each new PIR query a client receives, the server chooses a random scalar that extends the challenge vector. It multiplies the i^{th} element in the new query vector by that scalar and adds to the i^{th} element in the vector $\overrightarrow{qc}$ for that client.

5.2.2 *Responding to users*

When the server has all-correct results for the clients' jobs, it can respond to the PIR queries. The response to a particular client, say the client sending the i^{th} PIR query, comprises the i^{th} ciphertext from each client in the group handling that query. The server assembles the response vector and then applies the recursive PIR optimization (see §4) to obtain a single ciphertext containing the recipient's message. As explained in the background, this step requires only light computation relative to the rest of the PIR protocol, so the server does it directly and returns the resulting ciphertext to the recipient's client.

The client decrypts this response using the secret BGV key it chose for the query and extracts the message from the plaintext polynomial. It uses the shared secret with the sender to check the authentication code (so that a rogue server cannot provide invalid messages) and decrypts the message's end-to-end encryption to display its text to the recipient.

5.3 Stragglers

Some of the clients can be slow to compute, have poor connectivity, or experience temporal load or bad network conditions. All these scenarios may cause those clients to respond slower than others. Since the server knows the workload assigned to each client (the message matrix size and the number of PIR queries the client should process), it can set a deadline that it expects the clients to meet.

DPIR handles clients exceeding their deadline as clients returning invalid responses (above). The server removes them and redistributes their jobs. This way, stragglers do not stall following communication rounds. They can rejoin in the following epochs to continue communicating over DPIR (e.g., after upgrading their hardware or connectivity to avoid being classified as stragglers again).

5.3.1 *Client proxies*

Users may want to run clients on mobile devices, with unreliable network connections or limited batteries, that cannot handle DPIR's continuous work demands. Such users can designate a proxy to perform the computational work on the client's behalf. This delegation of tasks can be assigned to a desktop computer.

To support delegation, the client receives from the server a random access token at the beginning of the epoch (§5.1). This token allows the server to link the client to the work its proxy performs. The client gives the token to the proxy, which then connects and uses this token to request work instead of the client. In this setup, the user trusts its proxy; if her proxy turns out to be malicious, the server will block her messages (§5.2.1). Work delegation, however, does *not* impact DPIR's privacy or availability guarantees to other users. Intuitively, this is because the proxy views the same data as the client, which is already untrusted for privacy/availability. Our analysis in §7 captures this intuition.

5.4 Multi-server deployments

Since the server-side computations for each client group are independent, DPIR naturally benefits from multiple servers. In such a deployment, all servers hold a copy of the message matrix (synced every communication round across the servers, similar to previous systems that relied on compute cluster [1, 5]). Each server then handles a small portion of the PIR queries using a portion of the clients.

We show through evaluation that by outsourcing the bulk of work to clients, DPIR allows each server to support more clients compared to prior work (§9), which translates to a system that is more feasible to deploy in practice.

6 Dialing mechanism

DPIR, like many metadata-private communication systems (including the PIR-based designs, Pung and Addra), relies on an orthogonal dialing mechanism to coordinate communication between users. More precisely, when a server starts an epoch, it assigns each user a dead drop and arranges these dead drops into a matrix. Users chatting with each other need to learn each other's dead drop locations in the server's matrix. DPIR leverages keyword PIR to perform this coordination [10]. The keyword PIR-based dialing isn't new (see [5] for example). It matches DPIR's assumptions about the adversary and its privacy goal. However, it introduces a performance overhead, which we discuss here and evaluate in §9. We next overview the dialing technique.

The dialing protocol has two steps, "request" and "accept." At the beginning of an epoch, each user requests to connect with one other user. Alice requests to connect with Bob using a secret they share, sk. To do this, Alice derives an epoch-specific identifier $id = PRF_{sk}(Alice \rightarrow Bob, \text{"}request\text{''}, epoch\#)$ and sends it to the server. If Alice does not request to connect with anyone, she just sends a random identifier (to hide this fact from the server). The server creates a binary search tree, where keys are the pseudorandom request identifiers users send, and the values are their dead drop locations.

Bob also derives this identifier using sk, which Alice would use to connect with him. He then uses PIR to check whether Alice requested to connect with him as follows (following the keyword PIR protocol from [10]). He does this with binary search by treating the nodes in each layer of the server's binary search tree as an array, and issues a PIR query that retrieves the next node he checks in the search. If that node is keyed by an identifier larger than id then in the PIR query for the following layer he retrieves the left child and otherwise the right child. The process continues until the last layer, and Bob learns that Alice requested to connect with him if there is a node keyed by id. At this point, Bob knows Alice's dead drop location and can read from it in DPIR's communication rounds.

DPIR processes the PIR requests using distributed PIR. The server outsources PIR query processing to the clients and validates their outputs (as in §5); here the database consists of the nodes in each layer of the tree rather than messages users exchange. The keyword-PIR-based mechanism for a dialing request (above), requires $\log(n)$ queries, but each query only requires processing a subset of the database (the nodes in the corresponding layer of the tree). The overall cost is identical to that of processing a regular PIR query (i.e., linear in the size of the database). Bob can run this procedure for a fixed number of friends. We envision users running this test for, say, $10-20$ friends an epoch, which is as costly as $10-20$ communication rounds. The span of an epoch amortizes the cost of dialing over multiple communication rounds.

In the second step of dialing, Bob accepts one friend's request, e.g., Alice, by deriving the following identifier and sending it to the server, $id = PRF_{sk}(Bob \rightarrow Alice, \text{"}accept\text{''}, epoch\#)$. This step works the same way as the previous one: the server creates a binary search tree mapping the identifiers it receives to the sender's dead drop location. Alice knows to look for Bob's identifier since she requested to connect with him, so this step invokes the keyword-PIR protocol only once (cf. to the first step). If she finds this identifier, then she knows that Bob accepted her request to connect and learns his dead drop's location in the message matrix. They can both read from each other's dead drops during the epoch.

6.1 Alternative dialing mechanisms

There are alternative proposals for metadata-private coordination between users interested in communicating. Current proposals for dialing include broadcast communication between clients (where clients announce dialing requests, encrypted with the public key of their intended partner), private signaling [36], Alpenhorn [32] and fuzzy message detection [7]. These solutions have drawbacks, either in performance or trust assumptions. Fortunately, since DPIR utilizes the dialing mechanism as a black box, it would enjoy any future improvements to metadata-private dialing protocols. This is an active research area with promising directions. For example, the recent work on oblivious message detection [34] proposes a protocol where users query an *untrusted* server storing encrypted dialing requests; however, deploying this protocol may still not be practical. (It relies on homomorphically re-encrypting every user query for each dialing request the server stores, which is an expensive procedure, that has to be repeated many times.)

7 Analysis

We show that DPIR meets its privacy, availability, and scalability goals from §3.2.

7.1 Privacy

In DPIR, clients submit fixed-size messages at every round to hide the communication's size and time. Its server distributes the work in the PIR protocol to hide which message each user reads. Intuitively, since DPIR's server is the one distributing the workload to the clients, the attacker's "view" about an honest client's query is limited to that of a rogue server's view (the network message transcript, corrupt parties' memory snapshots, etc.). This implies that DPIR hides which users communicate since PIR leaves the server oblivious to what the honest client queries.

Formally, DPIR achieves Pfitzmann and Hansen's relationship unobservability property [38]. To prove it, we define an indistinguishability game and use a simulation-based proof to show that security holds. The simulation captures all information that an attacker may view or corrupt from controlling servers and clients, such as which user writes where, work delegation to clients, and verification of correct work. The following theorem formalizes DPIR's privacy guarantee, due to space limitations we defer the full formal proof to Appendix B.

Theorem 1. DPIR provides relationship unobservability. Given a security parameter λ, for any PPT adversary A there exists a negligible function ϵ such that A cannot distinguish with probability $\geq \frac{1}{2} + \epsilon(\lambda)$ between two scenarios σ^1 and σ^2, where each scenario describes which pairs of users communicate through DPIR.

Proof. Given in Appendix B. □

7.2 Availability

DPIR should respond correctly to queries when its server (or servers, §5.4) is honest, even if clients (or their proxies) are corrupt. There are two ways a misbehaving client can attack availability: (1) it could avoid returning any response to the server (or respond late to delay communication); (2) it may return an invalid response. To solve the former option, the server imposes a deadline for clients to complete their jobs (§5.3). We thus focus the analysis on the latter option; we show that the server detects clients returning wrong results and, therefore, retrofits availability by computing their work itself or redistributing it across the remaining clients (§5.2.1).]

7.2.1 Detecting invalid client response The server verifies any result that the client returns using the randomized check from Freivalds' algorithm. The chance of error depends on the probability that this check does not catch a cheating client, which diminishes exponentially with the bit-length of the scalars in the challenge vector $\vec{c}$, see analysis in [19] (we set this parameter to ensure very low chance of error, in §8).

Typically, Freivalds' algorithm verifies the product of plaintext matrices. However, realizing this algorithm in DPIR's setting involves verifying the product of a plaintext matrix ($\mathcal{M}$) and the PIR-query matrix ($\mathcal{Q}$ in Algorithm 3), which holds BGV ciphertexts encrypted under different public keys (from different clients). And, although the server does not hold the clients' decryption keys, it needs to detect whether the equality $\mathcal{M}\times(\mathcal{Q}\times\vec{c})-\mathcal{R}\times\vec{c} = \vec{0}$ holds for the results from the clients ($\mathcal{R}$) and a random challenge vector $\vec{c}$ (see §4). This equation holds if and only if the $\times$ operations are associative. The following theorem shows that $\times$ associates when multiplying matrices of BGV ciphertexts and plaintexts.

Theorem 2. Let M be a matrix consisting of BGV plaintexts, Q be a matrix consisting of BGV ciphertexts (encrypted under different keys), and $\vec{c}$ be a random vector of d-bit integers. Then $(M \times Q) \times \vec{c} = M \times (Q \times \vec{c})$. For a ciphertext matrix, $R \neq M \times Q$, $R \times \vec{c} \neq M \times (Q \times \vec{c})$ with probability $\geq 1 - 2^{-d}$.

Proof. We start by observing that to compute all matrix-matrix or matrix-vector products in Freivalds' algorithm's test above, there are only two types of BGV arithmetic operations involved: adding two ciphertexts and multiplying a plaintext with a ciphertext. Let us take a closer look at how these operations are defined for BGV, given in Listing 1. In BGV, a ciphertext is a tuple of polynomials in the quotient ring R_q, and a plaintext is a polynomial in $R_t \subset R_q$. The evaluation of Freivalds' randomized check treats plaintext polynomials as members of R_q for polynomial addition and multiplication. Therefore, it is sufficient to show that both these operations are associative to conclude that the computation on matrices and vectors is associative.

```
1      def BGV_Ciphertext_Add(ctx1, ctx2):
2        result1 = ctx1[0] + ctx2[0]  # poly add in $R_q$
3        result2 = ctx1[1] + ctx2[1]
4        return BGVCiphertext([result1, result2])
5
6      def BGV_Plaintext_Mult(ptx, ctx):
7        result1 = ptx × ctx[0]  # poly mult in $R_q$
8        result2 = ptx × ctx[1]
9        return BGVCiphertext([result1, result2])
10
```

Listing 1: Pseudocode for BGV plaintext-ciphertext multiplication and ciphertext-ciphertext addition with a degree one ciphertext (the type of ciphertexts DPIR's clients compute on, which comprises a tuple of two polynomials in R_q).

We observe that each of the two operations results in a ciphertext comprising two polynomials, and that computing one polynomial is independent of computing the other. Thus, the randomized check can be viewed as two independent calculations of Freivalds' algorithm on matrices in R_q: one where instead of each ciphertext matrix (R, Q) we compute on matrices R_0, Q_0 that contain only the first polynomial in each ciphertext's tuple, and the second where we compute the test using matrices R_1, Q_1 which contain only the second element in each ciphertext tuple.

Furthermore, the operations on individual polynomials in R_q (lines 2,3,7,8 in Listing 1) are associative (a property of quotient rings). Thus, Freivalds' algorithm can run independently to check whether $M \times Q_0 = R_0$ and $M \times Q_1 = R_1$. If $M \times Q = R$ then both equalities hold. Otherwise, then either $M\times Q_0 \neq R_0$ or $M\times Q_1 \neq R_1$ and Freivalds' test catches that with probability $\geq 1 - 2^{-d}$. □

7.2.2 Reusing the challenge vector DPIR reuses the challenge vector in the rounds comprising an epoch to reduce calculations. Reusing the challenge vector slightly reduces the chance of identifying a rogue client that responds incorrectly. The following theorem analyzes this degradation, showing that DPIR guarantees the availability promise albeit requires slightly larger space for its challenge vector than if the vector was fresh each time (i.e., requiring a higher parameter d).

Theorem 3. In respect to 2, let there be $\sqrt{n}$ disjoint groups of clients and let each such group reuse the same challenge vector *vvc* consisting of d-bit uniformly random integers. Then, the probability of the adversary corrupting a query and not being caught is upper-bounded by $\frac{n}{2^d - \sqrt{n}}$.

Proof. We prove this theorem by inspecting what information the adversary gains while interacting with the server's verification protocol §5.2. We begin by inspecting the interaction between a server and a single group of clients G of size $\sqrt{n}$. G is designated for work $w = \{\mathcal{M}, \mathcal{Q}\}$, where $\mathcal{M}$ is the message matrix, and $\mathcal{Q} = \{\vec{q_0}, \vec{q_1}, \ldots, \vec{q_{|G|}}\}$ is a set of queries that the group handles, such that the size $\mathcal{Q}$ matches the number of rows in $\mathcal{M}$.

The server generates a random vector $\vec{c} \in_R [0, 2^d - 1]^{\sqrt{n}}$. Then it distributes the work w across the clients in G, and expects each client $c_i \in G$ to send a response $\vec{r_i} = \mathcal{M}[i] \cdot \mathcal{Q}$. The server then validates that $\vec{r_i} \cdot \vec{c} = \mathcal{M}[i] \cdot (\mathcal{Q} \cdot \vec{c})$. If the equality check does not pass, the server considers c_i corrupt and removes it from G (i.e., $G = G/c_i$). The server continues to engage with G as long as it is not empty (§5.2).

Now that we have established the interactions between the server and the group, let us discuss what information the adversary can gain in every round by sending corrupted responses.

If the adversary does not corrupt anything and behaves according to the protocol, the check always passes (regardless of the value of $\vec{c}$). Thus, the adversary can learn nothing about the challenge vector $\vec{c}$ and its entries. Hence, we assume the adversary corrupts at least one response, say $\vec{r_0}$ is the first response the attacker corrupts. If the client corrupts a single entry in $\vec{r_0}[k]$, it observes whether its client was removed from G and learn some information about $\vec{c}$, namely, what number is not occupying $\vec{c}[k]$ (i.e., in $\vec{r_i} \cdot \vec{c} = \mathcal{M}[i] \cdot (\mathcal{Q} \cdot \vec{c})$. Each entry of $\vec{c}$ is affected by a single entry in $\vec{r_i}$), therefore, by corrupting more than a single entry in r_0, say ℓ entries, the adversary learns that at least one of $(2^d)^\ell$ possible options is incorrect but not which one; hence this strategy is inferior (from the attacker's perspective) to corrupting just one entry in $\vec{r_0}$.

We thus analyze the probability that an adversary, which uses its clients to return responses with one invalid entry, can avoid detection with at least one of the clients in the group (at most $\sqrt{n}$). Considering we have discussed how the adversary gains information during the duration of an epoch, using at most $|G| = \sqrt{n}$ clients, we can formulate its probability of being able to return one corrupt response without detection as $\sum_{i=0}^{\sqrt{n}} \frac{1}{2^d - i} \leq \frac{\sqrt{n}}{2^d - \sqrt{n}}$.

Since the server allocates each group an independent challenge vector, we can use the union bound to upper-bound the chance one of the attacker's clients remains undetected after returning an invalid response across all $\sqrt{n}$ client groups, $\sqrt{n} \times \frac{\sqrt{n}}{2^d - \sqrt{n}} = \frac{n}{2^d - \sqrt{n}}$. □

7.3 Scalability

We analyze the asymptotic computation and network cost for the server and n clients. We assume that in a steady state, all clients send and receive messages and provide correct responses to their jobs (since rogue clients get booted as soon as they cheat, §7.2).

Theorem 4. Given n clients reading and writing messages, the computation cost is $O(n\sqrt{n})$ for DPIR's server and $O(n)$ for the client per epoch setup and communication round. The communication cost is $O(n\sqrt{n})$ for the server and $O(\sqrt{n})$ for the client per epoch setup and communication round.

Proof. We review the steps in DPIR's operation to evaluate its asymptotic computation and communication costs on the server and client.

Server computation. At epoch setup (§5.1), the server expands the PIR queries that it receives from each client into $O(\sqrt{n})$-long ciphertext vectors. There are n clients and the cost of every expansion is linear in the length of the expanded vector (see [4]). Thus, the cost of expansion is $O(n\sqrt{n})$ per epoch.

In every communication round (§5.2), the server receives from each client an $O(\sqrt{n})$-long result vector and checks its work using Algorithm 3. The cost of validation is linear in the number of elements in the result, i.e., $O(\sqrt{n})$ per client and $O(n\sqrt{n})$ overall.

In the last step before returning the result to the recipient (§5.2.2), the server multiplies the client's second query vector by the $O(\sqrt{n})$-long result vector from processing the first PIR query. This takes $O(\sqrt{n})$ BGV operations per client, and $O(n\sqrt{n})$ across all clients. Thus, the total computation cost for the server is $O(n\sqrt{n})$.

Client computation. Each client receives $O(\sqrt{n})$ compressed queries per epoch (§5.1). It expands every query into $O(\sqrt{n})$-long vectors, so the total work at the beginning of each epoch is $O(n)$ per client.

On every communication round, the client multiplies its row of the message matrix by the $O(\sqrt{n})$ query vectors it has (also described in §5.2). The row and each query vector's size is $O(\sqrt{n})$, so multiplying one pair of vectors costs $O(\sqrt{n})$ and the total cost for the client is $O(n)$ per round.

Server and client communication costs. The server sends each client $O(\sqrt{n})$ compressed queries at the start of each epoch (§5.1). So the total cost across n clients is $O(n\sqrt{n})$.

The server also sends each client a row of the message matrix on each round (§5.2), which is of size $O(\sqrt{n})$. It receives back an $O(\sqrt{n})$ vector, the product of multiplying the client's row vector by each of the expanded query vectors. In total, the server sends and receives $O(n\sqrt{n})$ communication per round across all n clients. Each client uses $O(\sqrt{n})$ communication. □

8 Implementation

We implemented DPIR in 3 445 lines of `C++` code, on top of a state-of-the-art PIR query compression and expansion algorithm from SealPIR [4]. We have made our prototype publicly available [44]. Our implementation uses the BGV implementation from Microsoft's SEAL library [42], and we elaborate on how we use SEAL in Appendix C.

8.1 System parameters

Epoch length. The epoch length in DPIR represents a trade-off: it amortizes the cost of query expansion (§5.1.1) over multiple communication rounds, at the expense of allowing users to switch the friend they communicate with only on the epoch boundary. To determine a good epoch length, we compare the epoch's setup time against the communication round time. Our experiments (§9) show that an epoch length of a few minutes (e.g., 5 minutes) would dwarf the epoch's setup time relative to the time spent in communication rounds. We believe it provides a good trade-off point that balances performance with usability.

BGV parameters. We use the default 128-bit security parameters for BGV with polynomial degree $N = 2^{12}$ from the SEAL library. In this setting, DPIR uses a 109-bit ciphertext coefficient modulus q and a 20-bit plaintext coefficient modulus t [pp. 26][3]. Further parameter optimization can trade support for larger message matrices (which induce more homomorphic operations) for lower communication and computation costs. The parameters we chose work for matrices of up to 33M messages, allowing support for a large user base.

Challenge vector. Freivalds' algorithm identifies an incorrect matrix product with a probability $\geq 1 - 2^{-d}$, where d is the bit-length of each element in its challenge vector (see Algorithm 3). In DPIR, where we reuse the same challenge vector throughout an epoch, the probability for corruption to occur is higher (see §7.2). To ensure our availability goals, we implemented a scalar-ciphertext multiplication method that supports 60-bit scalars relying on SEAL's internal methods, since SEAL doesn't provide this specific method. This allowed our verification to use challenge vectors comprised of $d = 60$-bit integer elements.

When the server runs Freivalds' verification (Algorithm 3), which multiplies ciphertexts by the challenge vector's scalars, calculations are performed between polynomials in a quotient ring $R_q = \mathbb{Z}_q[x]/(x^N + 1)$ and scalars from Z_q (see §4). Thus, with 2^{20} clients, and 60-bit entries in the challenge vector, DPIR guarantees its availability goal with $1 - \frac{9.09}{10^{12}}$ probability per epoch (see Theorem 3).

Message size. We target supporting text messaging with DPIR. Thus, we set the size of messages users exchange to 256B in our implementation (comparable to SMS message size). With the cryptographic parameters above, each BGV plaintext in the message matrix can pack about 10KB of data, i.e., 39 user messages (see discussion in §5.1.1). E.g., to support clients exchanging 2^{18} messages each round, the server will hold a matrix of 82×82 BGV plaintexts.

9 Evaluation

We evaluate DPIR using our prototype implementation, deployed on 15 nodes on our university's compute cloud, with AMD EPYC 7662 2.00GHz CPUs. Testing DPIR with several servers and a large client

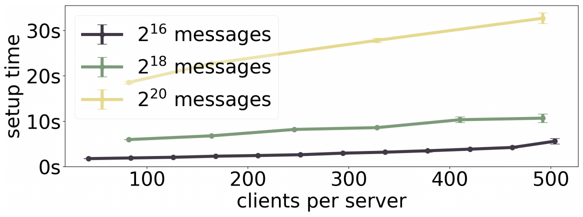


Figure 4: Epoch setup time as a function of the number of clients that DPIR's server handles.

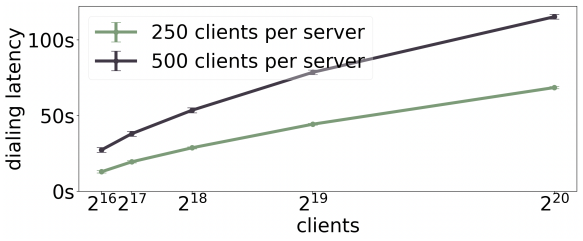


Figure 5: Dialing latency using DPIR's keyword PIR-based mechanism (§6).

base requires more machines than our university cloud has. Thus, to evaluate how DPIR would perform in large deployments, we evaluate a scenario in which each server holds the entire message matrix but handles a subset of the clients and their associated PIR queries. This form of evaluation represents a DPIR deployment well because the system is highly parallelizable: each server can hold a copy of the message matrix and independently serve a disjoint set of querying clients (see §5.4).

Our evaluation setup consists of one server machine and up to 504 clients spread across 14 machines. We use SLURM to allocate resources to the server and client processes: we deploy the server on a 12-core node with 128GB of RAM. We assume clients operate on less powerful machines, so we allocate 1 core per client and 8GB of RAM. Nodes are connected via 100Gpbs Ethernet, and we use Linux's `tc` command to control the nodes' bandwidth in several experiments. Each data point in the graphs below is the average of 5 iterations, we use error bars to show the standard deviation.

Our evaluation quantitatively answers the following:

(1) What is the epoch setup time, and how does it scale with the number of clients?
(2) How does communication latency scale with the number of clients and client-side network limitations, and how does it compare with prior work?
(3) What is the effect of a larger message matrix and user base growth on performance?
(4) How do client faults impact performance?
(5) What are the memory and bandwidth costs from the server and the clients?

In the experiments below we encode the challenge vector's scalars as BGV plaintexts. This encoding method is less efficient than our current implementation (§8). Specifically, the plaintext encodes 20-bit integers and gives similar performance to using SEAL's scalars directly which DPIR's implementation currently uses. In both cases, multiplying an integer by a ciphertext takes about 0.04ms on average (the new approach is 7% faster), but the current encoding further diminishes Freivalds' verification failure rate (since it uses 60-bit integers in its challenge vector). Therefore, we expect DPIR to slightly outperform the measurements below.

9.1 Epoch setup time

Figure 4 illustrates the latency of DPIR's epoch setup time (§5.1) as a function of the number of clients it handles. This graph excludes the time for dialing between users, which we next evaluate separately. The reason for this distinction is that DPIR uses an orthogonal dialing mechanism as a black box, and this component may be easily replaced (see discussion in §3).

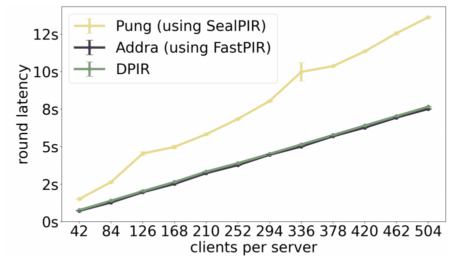
The main cost of the setup time stems from expanding the clients' PIR queries: the server and clients expand the queries they receive to column vectors that match the row length of the message matrix (the more messages there are, the more expensive expansion becomes).

Dialing. Figure 5 details the latency induced by DPIR's keyword PIR-based dialing mechanism (§6). We measure the latency as a function of the whole user base (x-axis) and the number of clients our server serves. We assume that each client checks for connection from one other client, if clients probe for more connections then the dialing time increases proportionally. We observe that performance is inversely proportional to the number of clients each server handles, which illustrates that the dialing phase is highly parallelizable.

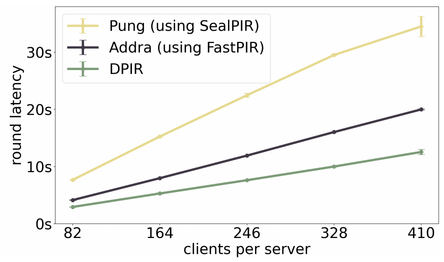
Setup time amortization. The cost of epoch setup amortizes across all communication rounds in the epoch. We envision DPIR's epochs lasting minutes' worth of communication rounds (say, 10 minutes), to dominate its setup time. DPIR may pipeline setup for the next epoch and run it while the current epoch is still running (allowing the users from two epochs ago to participate in the current epoch, rather than the previous one which hasn't ended).

9.2 Communication latency

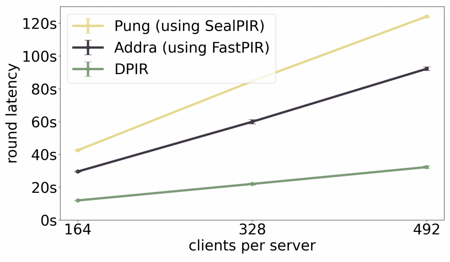
Figure 6 measures the average time it takes DPIR to complete a communication round. Each subfigure fixes the number of messages in the system. The cost of processing each PIR query increases with the number of messages so we expect DPIR to increasingly outperform non-distributed PIR-based systems as it grows. We test DPIR with 2^{16} – 2^{20} messages to measure how communication round latency (the y-axis) scales with the number of messages users exchange in that round. On the x-axis, we vary the server's workload: the more clients a server handles, the fewer servers the system needs to support the user base and the more tangible its deployment becomes. We compare with alternative state-of-the-art designs, Pung [4] and Addra [1]. Pung and Addra use SealPIR and FastPIR respectively, which run computations only on the server. They are built using Microsoft's SEAL library for native BGV implementation as DPIR, and we deploy them using the same BGV parameters (§8.1). This gives an apples-to-apples comparison since our evaluation assumes an already existing



(a) 2^{16} messages



(b) 2^{18} messages



(c) 2^{20} messages

Figure 6: Communication round latency in each system for different numbers of messages. Increasing the number of messages translates to larger anonymity sets and heavier workloads.

database of messages and since the PIR engine is the core of those systems, representing the primary CPU and bandwidth consumer. Our measurements omit the cost of clients sending PIR queries and receiving the response, which is small for DPIR; Pung has the same cost for sending/receiving queries/responses, but it is much more expensive in Addra (see below). The comparison with Pung is also a comparison against a centralized baseline since the Pung server performs all the PIR computations that the DPIR server outsources.

In all designs, the communication latency grows linearly with the number of clients a server handles. However, comparing the latency across the three subfigures shows that DPIR scales much better with the number of messages in the system, allowing for a larger user base and anonymity sets. This is due to the DPIR's scalability improvement: the PIR cost increases for all systems with the number of messages, but it increases slower for DPIR since it utilizes clients to alleviate the bottleneck at the server.

Asymptotic latency improvement, in practice. To further illustrate the asymptotic scalability improvement over prior work, Figure 7 compares the communication round latency while varying the number of messages on the server. In this experiment, we grow the number of clients a server handles proportionally to the number of messages (the x-axis). This captures a real-world deployment where, when the server handles more clients submitting PIR queries (more recipients), it also handles a larger message matrix (since more clients are sending messages too). We observe that as the client load increases, the ratio between the other systems' latencies and DPIR's latency grows: from 1.81× and 1.24× (for Pung and Addra, respectively) with 2^{19} messages to 4.31× and 3.25× with 2^{21} messages. This increase captures DPIR's asymptotic improvement.

Furthermore, we monitored the clients' computation time to check whether the client-side overhead was practical. The dashed line at the bottom of Figure 7 shows the slight increase in work that each client performs with DPIR's message volume, illustrating that clients would perform a few seconds' worth of computation on a single core per round even when the system handles many messages. That is, trading a few seconds of client computation for reducing tens, or even hundreds, of seconds in communication latency (compared with [1, 5]) is a good trade-off.

Clients with limited bandwidth. We next test how client-side network bandwidth constraints impact DPIR's performance. Using Linux's `tc` command, we limit the bandwidth for the clients and

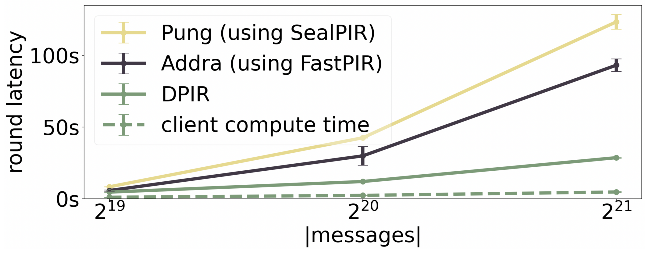


Figure 7: Latency increases with the number of messages in the system and proportional growth in the number of clients.

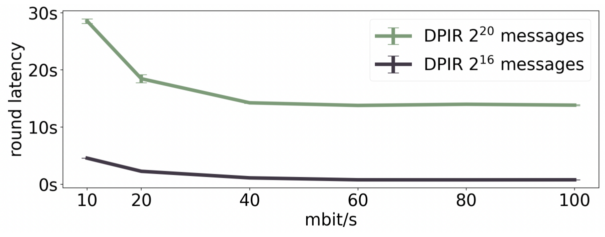


Figure 8: Bandwidth-limited clients

measure the impact on the communication round's latency in Figure 8 (we impose the same limit for upload and download). In this experiment, we set the number of messages in DPIR to 2^{16} or 2^{20} and have its server handle 164 clients so we can compare the results with data-points from Figure 6a and Figure 6c (given the number of messages, the client's communication costs are the same even if each server handles more clients). We observe that with a 10mbps limit, the communication round latency is higher than the latency we observed in Figure 6 without the bandwidth constraint. Notably, this latency is still better than Pung and comparable to Addra unconstrained; see Figure 6c. DPIR's communication latency decreases as we raise this limit and at about 40mbps, it converges to what we saw earlier, with unconstrained clients in Figure 6, suggesting that at this point network costs no longer significantly impact performance. In many countries, e.g., in the US, the average home internet connection speed satisfies a 10 - 40mbps requirement [35].

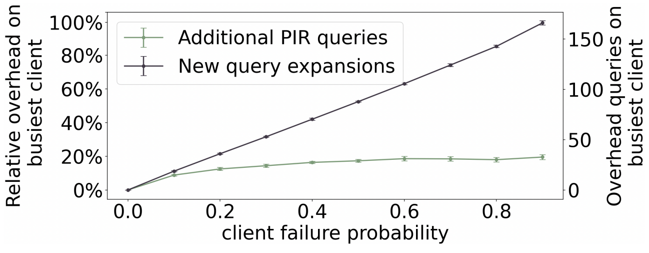


Figure 9: Relative and absolute additional work per client given a client fault rate. Simulation with 1M clients.

9.3 Client faults

When a client returns an incorrect result or becomes unresponsive, DPIR removes that client and its query, reducing both the workforce and the workload (§5.2.1). These opposing effects keep the *average* workload of processing PIR queries constant for the remaining clients. However, DPIR's PIR processing will be bottlenecked by the busiest client. Figure 9 uses a simulation to plot the maximal relative and absolute overhead work (extra PIR queries to handle after discarding queries from faulty clients) that a healthy client needs to perform, as a function of the client fault rate. We observe that even under a high fault rate, the maximal overhead remains under 25%. Thus, after redistributing the faulty clients' work, DPIR's communication rounds should not incur substantial additional latency.

Work redistribution costs. Redistributing a faulty client's work involves a one-time cost when a client fails. This procedure includes the server updating the cached validation step for the clients that handle new queries, transmitting these queries in compressed form to those clients, and having them expand those queries (§5.1). The dominating cost is query expansion, which takes 28.28ms – 122.18ms per query when DPIR has 2^{16} – 2^{20} messages. With a 10% fault rate, the busiest client receives about 20 additional PIR queries. The overhead is small since the work of the faulty 10% is divided across the remaining 90%, leading to a latency of 600ms – 2.5s before the busiest client starts processing the new workload (for 2^{16} – 2^{20} messages). This cost increases to about 100 queries and 2.8s – 12.2s when 50% of clients fail. Beyond the 50% fault rate, we find that performing epoch setup again and distributing queries from scratch is cheaper (see Figure 4).

9.4 Communication and memory costs

To evaluate the cost of operating DPIR, we inspected the client and server's bandwidth cost and peak memory usage while running the experiments above. As mentioned in §3, the client may use/pay a proxy to perform its part in processing DPIR's PIR queries (to simplify support for relatively weak devices such as mobiles). The system supports such delegation and it does not break DPIR's guarantees (our analysis in §7 captures this scenario).

Communication. In our experiments, each client sends to the server 2MB (for 2^{16} messages in the system) – 8MB (for 2^{20} messages in the system). The client also receives 420KB – 1640KB (for the same 2^{16} – 2^{20} messages in the system). The server-side costs are symmetric and aggregate across the clients the server handles (the server sends/receives this data). Though the network costs per client are high relative to the size of messages they exchange, they are on par with recent PIR-based communication systems like Pung's original XPIR engine [5].

Memory. We also monitored the client and server's memory usage through an epoch. While running the experiment from Figure 6 we polled Linux's `top` command every 500ms. With 2^{16} messages in the system, we find that the peak client memory usage is 540MB and that the server uses about 35MB of memory per client. With 2^{20} messages, the peak client memory usage grows to 3.6GB, and the server uses 58MB per client. The dominating factor in memory usage, both for the clients and the server, is keeping PIR queries in decompressed form during the epoch.

10 Limitations

DPIR illustrates an advancement in reducing communication-round latency, but it is essential to outline the expenses for its clients. As the user base grows (and the anonymity pool increases with it), so do the clients' bandwidth, CPU, and memory requirements. In the following paragraphs, we expand on each resource to thoroughly discuss the costs associated with our system and propose mitigations to alleviate these requirements.

Bandwidth. DPIR requires more bandwidth than SealPIR (and less than Addra, for a sufficiently large user base) since clients need to receive their workloads and return a response to the server. In absolute terms, for a user base of 1M exchanging 256B messages, DPIR's client bandwidth usage is about 8MB per round. However, this allows DPIR to considerably reduce the server's computational costs.

We argue that this trade-off between server communication and computation addresses the performance bottleneck in many PIR systems. This bottleneck is demonstrated by prior work SealPIR and Addra, which use considerably less bandwidth than XPIR yet their communication round latency remains relatively close to it (see [4, Figure 10], [1, Figure 6]).

CPU. In DPIR, clients participate in processing queries every round. In addition, once every epoch, all clients have to expand the compressed queries they receive from the server. Although these computational tasks are generally brief compared to the communication-round time (see Figure 7), they represent additional client-side costs over prior work.

Memory. Clients are required to store numerous expanded queries during an epoch. These queries have a large footprint, for instance, with 1M clients in the system, each client stores around 3.8GB of expanded queries.

10.1 Mitigation

Devices that cannot spare the extra compute cycles or may occasionally disconnect from the network, such as mobile devices, might need additional mitigations to participate as clients in a DPIR deployment. We expand on possible strategies in this subsection.

Work delegation Users may delegate their clients' work to other machines, e.g., a desktop computer or a service provider that would process their work for a fee. The delegate would perform the client's work and return its result to the server. Entrusting the work to someone else minimizes the client's resource usage (i.e., similar to systems like Addra and Pung). Such delegation does not impact DPIR's privacy guarantee (the analysis in §7.1 captures this scenario).

Heterogeneous clients An alternative for supporting weak clients without work delegation is to adapt DPIR to assign work proportionally to a client's capabilities. DPIR could further adapt its incentive mechanism to allow clients to send and receive messages at a rate proportional to their contribution.

Making heterogeneous work assignments can be done by splitting each row of the server's message matrix into even chunks, and assigning fewer chunks to weaker clients. The server can then treat each column of chunks as a small matrix and perform our verification protocol on these chunks instead of on the entire matrix. Assigning work in this fashion ensures the server can still catch malicious clients on the fly (Algorithm 3) using Freivalds' algorithm.

For example, if the server assigns some client just the first half of a row of its message matrix, the client can expand the queries assigned to it to columns as in DPIR's original design but discard their bottom half (reducing RAM usage, too). The client then multiplies half of the message matrix row assigned to it in every round by the truncated query columns and returns the result to the server. The total work of verification stays the same, but the server now has to sum the results of the clients to extract a single entry per row.

This technique adjusts the work for each client to its capabilities at the cost of additional bandwidth cost for the server at epoch setup, since it has to send the compressed queries to more clients. This also increases the *average* client cost since more clients have to expand the same queries.

RAM optimization While storing expanded queries can be demanding, we notice that clients only need to keep a few of these queries simultaneously in RAM. Most of the expanded queries are either waiting to be used in computing the client's (message-)vector and (query-)matrix product or have been used already, and are thus not needed in RAM. A memory-optimized implementation of DPIR would create a processing pipeline where queries that are not currently needed are stored on the disk. That is, the client would hold a buffer of queries in RAM and gradually swap processed queries to disk while loading the next queries to be processed to its buffer.

11 Conclusion

DPIR uses a new distributed PIR approach to conceptually depart from recent proposals for metadata-private communication systems by having its servers outsource the bulk of cryptographic computations to the clients. Distributed PIR keeps the clients oblivious about which users exchange messages. It efficiently handles the challenge of ensuring availability despite misbehaving clients by treating PIR as an instance of matrix multiplication and using the insight that it is much faster to verify than to compute. We prove DPIR's privacy and availability properties and analyze its asymptotic scalability. We use a prototype implementation to demonstrate its performance and compare scalability against prior work.

12 Acknowledgements

We thank Gil Segev, Oded Schwartz, Nickolai Zeldovich, the FHE.org community, the anonymous reviewers and shepherd for their comments, suggestions and feedback which have helped improve this paper. This work was supported by Israel Science Foundation grant no. 1104/23, the Hebrew University Cybersecurity Research Center. Yossi Gilad was partially supported by the Alon fellowship. Elkana Tovey was partially supported by the Scharf-Ullman Program

References

[1] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. 2021. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[2] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, et al. 2022. OpenFHE: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 53–63.

[3] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. 2018. *Homomorphic Encryption Security Standard*. Technical Report. HomomorphicEncryption.org, Toronto, Canada.

[4] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. 2018. PIR with Compressed Queries and Amortized Query Processing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy*. 962–979.

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

[6] Ludovic Barman, Moshe Kol, David Lazar, Yossi Gilad, and Nickolai Zeldovich. 2022. Groove: Flexible Metadata-Private Messaging. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 735–750.

[7] Gabrielle Beck, Julia Len, Ian Miers, and Matthew Green. 2021. Fuzzy message detection. In *Proceedings of the ACM Conference on Computer and Communications Security*. 1507–1528.

[8] Amos Beimel, Yuval Ishai, and Tal Malkin. 2000. Reducing the Servers Computation in Private Information Retrieval: PIR with Preprocessing. In *CRYPTO (Lecture Notes in Computer Science, Vol. 1880)*, Mihir Bellare (Ed.). 55–73.

[9] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) Fully Homomorphic Encryption without Bootstrapping. *ACM Trans. Comput. Theory* 6, 3 (2014), 13:1–13:36.

[10] Benny Chor, Niv Gilboa, and Moni Naor. 1998. Private Information Retrieval by Keywords. Cryptology ePrint Archive, Report 1998/003.

[11] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. 2015. Riposte: An Anonymous Messaging System Handling Millions of Users. In *2015 IEEE Symposium on Security and Privacy*. 321–338. https://doi.org/10.1109/SP.2015.27

[12] Henry Corrigan-Gibbs and Bryan Ford. 2010. Dissent: accountable anonymous group messaging. In *Proceedings of the 17th ACM conference on Computer and communications security*. 340–350.

[13] Henry Corrigan-Gibbs and Dmitry Kogan. 2020. Private Information Retrieval with Sublinear Online Time. In *EUROCRYPT (Lecture Notes in Computer Science, Vol. 12105)*, Anne Canteaut and Yuval Ishai (Eds.). 44–75.

[14] Debajyoti Das, Easwar Vivek Mangipudi, and Aniket Kate. 2022. OrgAn: Organizational Anonymity with Low Latency. *Proceedings on Privacy Enhancing Technologies* 3 (2022), 582–605.

[15] Leo de Castro and Keewoo Lee. [n. d.]. Verifiability in SimplePIR at No Online Cost for Honest Servers. ([n. d.]).

[16] Roger Dingledine and Nick Mathewson. 2006. Anonymity Loves Company: Usability and the Network Effect. In *Proceedings of the 5th Workshop on the Economics of Information Security (WEIS)*.

[17] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium*. 303–320.

[18] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. 2021. Express: Lowering the Cost of Metadata-hiding Communication with Cryptographic Privacy. In *30th USENIX Security Symposium (USENIX Security 21)*. 1775–1792. https://www.usenix.org/conference/usenixsecurity21/presentation/eskandarian

[19] Rusins Freivalds. 1977. Probabilistic Machines Can Use Less Running Time.. In *IFIP congress*, Vol. 839. 842.

[20] Yossi Gilad. 2019. Metadata-private communication for the 99%. *Commun. ACM* 62, 9 (2019), 86–93.

[21] Yossi Gilad and Amir Herzberg. 2012. Spying in the Dark: TCP and Tor Traffic Analysis. In *Proceedings of the 12th Privacy Enhancing Technologies Symposium*. 100–119.

[22] Michael Hayden. 2014. The Price of Privacy: Re-Evaluating the NSA. Johns Hopkins Foreign Affairs Symposium. https://www.youtube.com/watch?v=kV2HDM86XgI&t=17m50s.

[23] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. 2023. One Server for the Price of Two: Simple and Fast Single-Server Private Information Retrieval. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3889–3905. https://www.usenix.org/conference/usenixsecurity23/presentation/henzinger

[24] Bastien Inzaurralde. 2018. The Cybersecurity 202: Leak charges against Treasury official show encrypted apps only as secure as you make them. The Washington Post.

[25] kate o'flaherty. 2021. All the data WhatsApp and Instagram send to Facebook. wired.co.uk. https://www.wired.co.uk/article/whatsapp-instagram-facebook-data.

[26] Eyal Kushilevitz and Rafail Ostrovsky. 1997. Replication is NOT Needed: SINGLE Database, Computationally-Private Information Retrieval. In *FOCS*. 364–373. http://www.computer.org/csdl/proceedings/focs/1997/8197/00/index.html

[27] Albert Kwon, Mashael AlSabah, David Lazar, Marc Dacier, and Srinivas Devadas. 2015. Circuit Fingerprinting Attacks: Passive Deanonymization of Tor Hidden Services. In *USENIX Security Symposium*, Jaeyeon Jung and Thorsten Holz (Eds.). 287–302.

[28] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. 2017. Atom: Horizontally Scaling Strong Anonymity. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. 406–422.

[29] Albert Kwon, David Lu, and Srinivas Devadas. 2020. XRD: Scalable Messaging System with Cryptographic Privacy. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[30] David Lazar, Yossi Gilad, and Nickolai Zeldovich. 2018. Karaoke: Distributed Private Messaging Immune to Passive Traffic Analysis. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 711–726.

[31] David Lazar, Yossi Gilad, and Nickolai Zeldovich. 2019. Yodel: Strong Metadata Security for Voice Calls. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. 211–224.

[32] David Lazar and Nickolai Zeldovich. 2016. Alpenhorn: Bootstrapping Secure Communication Without Leaking Metadata. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 571–586.

[33] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. 2023. Doubly Efficient Private Information Retrieval and Fully Homomorphic RAM Computation from Ring LWE. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing* (Orlando, FL, USA) *(STOC 2023)*. 595–608. https://doi.org/10.1145/3564246.3585175

[34] Zeyu Liu and Eran Tromer. 2022. Oblivious message retrieval. In *Annual International Cryptology Conference*. Springer, 753–783.

[35] M-Lab. 2023. Measurement Lab: Measure the Internet, save the data, and make it universally accessible and useful. https://www.measurementlab.net/.

[36] Varun Madathil, Alessandra Scafuro, István András Seres, Omer Shlomovits, and Denis Varlakov. 2022. Private signaling. In *31st USENIX Security Symposium (USENIX Security 22)*. 3309–3326.

[37] Riana Pfefferkorn. 2021. We Now Know What Information the FBI Can Obtain from Encrypted Messaging Apps. justsecurity.org. https://www.justsecurity.org/79549/we-now-know-what-information-the-fbi-can-obtain-from-encrypted-messaging-apps/.

[38] Andreas Pfitzmann and Marit Hansen. 2010. A terminology for talking about privacy by data minimization: Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management. https://api.semanticscholar.org/CorpusID:150929990

[39] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. 2017. P2P Mixing and Unlinkable Bitcoin Transactions. In *NDSS*.

[40] Alan Rusbridger. 2013. The Snowden leaks and the public. The New York Review of Books.

[41] Sajin Sasy and Ian Goldberg. 2023. SoK: Metadata-Protecting Communication Systems. *Cryptology ePrint Archive* (2023).

[42] SEAL 2022. Microsoft SEAL (release 4.0). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA..

[43] Yi Shi and Kanta Matsuura. 2009. Fingerprinting Attack on the Tor Anonymity System. In *ICICS (Lecture Notes in Computer Science, Vol. 5927)*, Sihan Qing, Chris J. Mitchell, and Guilin Wang (Eds.). 425–438.

[44] Elkana Tovey, Jonathan Weiss, and Yossi Gilad. 2024. *DPIR*. https://doi.org/10.5281/zenodo.11115954

[45] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. 2017. Stadium: A Distributed Metadata-Private Messaging System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. 423–440.

[46] Adithya Vadapalli, Kyle Storrier, and Ryan Henry. 2022. Sabre: Sender-Anonymous Messaging with Fast Audits. In *2022 IEEE Symposium on Security and Privacy (SP)*. 1953–1970. https://doi.org/10.1109/SP46214.2022.9833601

[47] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. 2012. Dissent in Numbers: Making Strong Anonymity Scale. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 179–192.

A Comparison with DC-net based designs

Dissent [12], Dicemix [39], OrgAn [14], are based on DC-nets, a protocol for broadcasting anonymous messages without trusted servers (rather than point-to-point communication as in DPIR's context). However, DC-nets require all-to-all communication across the clients and thus scale poorly with the number of users. A follow-up design [47] addresses the communication overhead but weakens the attacker model by assuming an honest server in the system. Furthermore, DC-nets are vulnerable to disruptions by rogue clients and require complex recovery protocols, typically requiring all clients to interact through broadcast messages which creates another challenge to scalability. DPIR handles rogue clients efficiently using Freivalds' matrix product validation: the server checks each client's output without requiring further interactions between the server and client or across clients.

B Full security proof

We now discuss the relationship unobservability property (UO) [38] and then show that this property holds for DPIR's protocol (Our proof closely follows the proof technique from Pung[5, Appendix C]).

UO defines that for any given set of could-be senders S and set of could-be receivers R, an efficient adversary should not be able to detect anything sent from set S to set R. To show that DPIR provides this property, we define a security game that captures this concept.

Intuitively, the security game is defined between a challenger C and an adversary A, where the A is allowed to define two scenarios σ^0, σ^1 and present them to C. These scenarios essentially describe two different worlds, where clients from S send messages to clients in R. During the game, C chooses one of the scenarios given to it via a coin flip and simulates the run of the protocol using the chosen scenario (i.e., to define what messages clients send and what queries they create). Once the simulation is over, A should output which scenario it believes C used to simulate the run of the protocol. If A guesses correctly, it wins. Proving that it can detect the messages being sent between the two sets S and R.

B.1 Security game for UO

The game consists of a protocol with three phases, *setup*, *simulation*, and *guess*, played by a challenger C and an adversary A. We denote the game as follows

$$\mathcal{G}^{b}_{A,\pi,k,n}(1^{\lambda}) = b'$$

where A is the adversary, π is an abstract protocol that models communications through and to a messaging service. The protocol π adheres to the API that our simulation expects. In the security game, k is the number of communication rounds π runs, n represents the number of clients registered to the service. λ is the security parameter. b is the bit representing the coin toss of the challenger, i.e., which scenario it chooses. The adversary wins this indistinguishability game if it outputs $b' = b$.

We next delve into the definition of the setup, simulation and guess phases, and define the protocol π. Using these definitions, we will move to the proof itself.

B.1.1 Setup phase A specifies a set of *clients* who had initialized their dead drops but have not initialized any secret yet, and two scenarios σ^0, σ^1 of communicating pairs of honest clients belonging to the set *clients*. Each scenario exports two methods which the challenger calls: $client_j \leftarrow \sigma^b[client_i].getFriend()$ to fetch $client_j$ the friend client object of $client_i$, and $m \leftarrow \sigma^b[client_i].getMessage(\cdot)$ to collect the message for the specified round (which is $\perp$ when the client has nothing to send). The two scenarios the adversary chooses must adhere to three main restrictions:

First, both scenarios must be of the same size (i.e., message size must be the same and the number of clients in these scenarios must be the same). Otherwise, the adversary can guess b' according to the length of the scenarios, or the length of encryptions.

Second, we assume the adversary's scenarios describe honest clients only. The challenger cannot simulate malicious clients because their behavior is unspecified (In the simulation phase, we will describe how the adversary can integrate and generate outputs for malicious clients). Nevertheless, honest clients can still send or read messages from corrupted clients.

Third, If scenario σ^b details that some honest $client_i$ sends messages to a compromised $client_j$, then σ^{1-b} must have the same transcript for $client_i$ (we don't enforce such restrictions when both clients are honest). That is, $client_i$ must send the same messages to $client_j$ throughout scenario σ^{1-b}. This restriction is fair since our goal of relationship unobservability provides meaningful privacy only if both the sender and recipient are honest 3.2. Otherwise, the adversary can always distinguish between the two scenarios, by decrypting the specific messages $client_i$ sends to $client_j$ in scenario σ^0 which differs from σ^1.

Abstract Protocol interface. The abstract protocol π contains two sub-models, server and client, which together have six functions overall:

(1) `init`(*client*): will securely set each client with its secrets. Each pair of clients receives a shared symmetric secret key and separately gets a BGV private key of their own.

(2) `client.retrieve`($client_i$, $client_j$): outputs a query q using $client_i$'s private BGV key and $client_j$'s deaddrop index. The query is generated by

$$q = (client_i.id, v) = \texttt{QUERY}(client_i.BGVkey, client_j.deaddrop)$$

, a function defined by a secure PIR scheme.

(3) `client.send`($client_i$, m): outputs an encrypted message, which client i wishes to write to its deaddrop.

(4) `server.distribute_work`(*clients,queries*): receives a set of queries, q, generated by `client.retrieve` from different clients and deterministically splits them to partitions

$$\{w_i = (client_i.index, messages, q) | q \subseteq queries, client_i \in clients\}$$

where $client_i$ will receive w_i.

(5) `client.respond_to_work`($client_i$, wrk): receives w_i as described above (which can be untrusted), and treats *messages* as a matrix of *plaintexts*, and the queries as a matrix of column vectors comprised from *ciphertexts*, multiply them and output the result.

(6) `server.trim_queries`(*processed_wrk*, *queries*): receives *processed_wrk* the untrusted output of `client.respond_to_work`($client_i, w_i$), applies Freivalds' algorithm on it, and in case it is invalid, will update *queries* by removing the query of $client_i$, q_i from the list.

B.1.2 Simulation phase The challenger C performs a simulation of running a protocol π according to a chosen scenario. It generates a view for the adversary to observe by following the pseudocode in Listing 2.

The simulation starts by C choosing the scenario bit b by flipping a fair coin. During the simulation, C feeds the scenario's data to π's methods and facilitates the interaction between π and the adversary by moving data between them. That is, during the simulation, C shows A the inputs and outputs it provides or receives from π. This ensures the adversary sees anything sent between the clients and the server.

Because C cannot simulate the adversary A and any client/server under its control, it uses the adversary as an oracle. Allowing A to insert responses for any malicious client/server. Furthermore, this allows the adversary to overwrite, drop, or shuffle the order of honest clients' messages. C can call on A using the following functions: A.`getQueries`(), which states what queries the adversary chooses that C will see during the current simulation from any of the clients (including honest ones). $A.getResponse(\cdot)$ define what response C receives from clients' send requests, if any. A.`getWorkForClient`$(\cdot)$ states what work a client may receive. A.`getProcessedWorkFromClient`() outputs what response the simulation is allowed to receive from any client.

At the end of the simulation, the adversary reflects upon what it viewed. A now decides whether it wants to advance to the *guess* phase and output its answer to conclude the game, or ask to go back to the setup phase (i.e., provide two scenarios and let the simulation run again). A is allowed a polynomial (in λ) number of attempts. Afterward, a default value of $b' = 0$ is assigned to the game's output.

```
def simulation(π, A, σ⁰,σ¹, clients, rounds):
  σᵇ = choose_using_fair_coin(σ⁰, σ¹)
  requests = {}
  responses = {}

  for client_i in clients:
      π.init(client_i)
      req = π.client.retrieve(
            client_i,
            σᵇ[client_i].getFriend()
          )
      requests.append(req)
  queries = A.getQueries(requests)
  responses.append(queries)


  for rnd in rounds:
      msgs = []
      for client_i in clients:
          sreq = π.client.send(
                 client_i,
                 σᵇ[client_i].getMessage(rnd)
              )
          requests.append(sreq)
          resp = A.getResponse(sreq)
          responses.append(resp)
          msgs.append(resp)

      req = π.server.distribute_work(
             clients,
             queries,
             msgs
          )
      requests.append(req)

      for client_i in clients:
          wrk = A.getWorkForClient(
                client_i.index
              )
          responses.append(wrk)

          req = π.client.respond_to_work(
                client_i,
                wrk
              )
          requests.append(req)

          resp = A.getProcessedWorkFromClient(
                 client_i.index
              )
          responses.append(resp)

          req = π.server.trim_queries(
              client_i,
               resp,
              queries
              )
          requests.append(req)
          resp = A.getResponseForClient(client_i.index)
          responses.append(resp)
  return (requests, responses)
```

Listing 2: Pseudocode of the simulation performed by the challenger C

B.1.3 Guess phase Once the simulation is done, A must output $b' \in \{0, 1\}$. A wins if the $b = b'$, that is, A guessed correctly the scenario that the challenger chose.

B.2 Security proof

We define unobservability using the security game from §B.1 and then prove that DPIR satisfies this definition.

Definition 1. *Protocol π* provides unobservability *if given a security parameter λ, for all PPT algorithm A, for any polynomial number of rounds k and correct users n, there exists a negligible function negl such that:*

$$|\Pr[\mathcal{G}^0_{A,\pi,k,n}(1^\lambda) = 1] - \Pr[\mathcal{G}^1_{A,\pi,k,n}(1^\lambda) = 1]| \leq negl(1^\lambda)$$

Where the probability is defined over the random coins tosses of the challenger C.

Theorem 5. DPIR provides unobservability.

Proof. We prove the theorem through a series of hybrid games. I.e., we define three different abstract protocols 0, 1, 2 that differ slightly from each other and show that the adversary gains only negligence advantage to win the game by switching from game i to game $i - 1$. Finally, if the adversary's best strategy for winning game 3 is to flip a coin, then protocol 0 does not give the adversary more than a negligible advantage to win the game.

GAME 0 is the original game $\mathcal{G}^0_{A,\pi,k,n}(1^\lambda)$, where π =DPIR. In this game, $\pi.init(client_i)$ does the bootstrap and sets the client with a shared secret key with the friend the client chats with (described in the scenario), and a FHE secret key to generate queries.

$\pi.client.send(client_i, m)$ outputs (pos, c) where pos is the cell that the client writes to, and the ciphertext c is computed as $AES_{ssk}(m)$, where ssk is the shared secret between $client_i$ and its friend $client_j$.

$\pi.client.retrieve(client_i, client_j)$ uses $client_i$'s secret key to generate a retrieval query from client_j cell, similar to 1; The output of $q = (client_i.id, v) = Query(client_i.sk, pos)$ is a query made out of a vector of BGV ciphertexts, as described in 5, and the unique ID of the client that created this query.

$\pi.server.distribute\_work(clients, queries, messages)$ distributes queries by shuffling them first and then distributing the work evenly across the clients. clients whose query q is not part of *queries* does not receive any work. Furthermore, the method distributes *messages* as a matrix of plaintexts clients write. For the simplicity of the proof, each client receives the full matrix.

$\pi.client.respond\_to\_work()$ is straightforward, given a matrix, and queries, perform matrix multiplication as described in 5, in case it cannot perform it (i.e., due to bad *distribute_work* call), outputs $\perp$ to indicate an error.

Finally, $\pi.server.trim\_queries(resp, client_i, queries)$ uses Freivalds' algorithm to validate whether the client processed the work correctly. If it had not, or $resp = \perp$, it will remove the client's query q from *queries*.

GAME 1 is the same as *GAME 0*, but $\pi.client.send(client, m)$ generates a new random message m' of the same size of the given m, and encrypts it using a newly randomly chosen secret key sk using $AES_{sk}(m)$.

GAME 2 is the same game as *GAME 1*, but $\pi.client.retrieve$ ignores the input and generates a CPIR query for a random deaddrop cell.

Now that we have defined the π protocols for the hybrid games, we want to show that there are methods in the simulation phase that do not provide any new information to the adversary during the simulation phase and can be simulated by the adversary.

Lemma 6. *The methods $\pi.client.respond\_to\_work(\cdot)$, $\pi.server.trim\_queries(\cdot)$ and $\pi.server.distribute\_work(\cdot)$ can be simulated entirely by the adversary.*

Proof. The method $\pi.client.respond\_to\_work(client_i, queries, msgs)$ takes messages *msgs* and treats it as a matrix of plaintexts, and responds with matrix V, where $V[i,j] = \sum_k^{msgs.cols} m[i,k] \cdot queries[j].v_2[k]$. This operation is related to the given queries and messages. This information is public and is provided to the adversary by the simulation. The adversary can apply the same deterministic calculations by itself, even without calling to $\pi.client.respond\_to\_work$, as this method does not rely on information hidden from the adversary like the client's secret key. Therefore, the adversary can simulate this function entirely on its own.

Similar to the method above, both $\pi.server.distribute\_work(clients, queries)$ and $\pi.server.trim\_queries(client_i, resp, queries)$, can be simulated by the adversary because they operate deterministicly over information exposed completely to the adversary's view. As a result, these functions, too, can be simulated by the adversary. □

Because the adversary can simulate all three methods, we change the simulation method described in 2 so there are no calls to $\pi.client.respond\_to\_work(\cdot)$, $\pi.server.trim\_queries(\cdot)$ and $\pi.server.distribute\_work(\cdot)$. Instead, the simulation will use the adversary as an oracle with the following methods: $A.respond\_to\_work(\cdot)$, $A.trim\_queries(\cdot)$ and $distribute\_work(\cdot)$.

Now that we have reduced the amount of methods π uses in the simulation phase, and instead give the adversary the responsibility to simulate them fully, we can focus on the hybrid games and the methods that remain suspected of giving the adversary hints that help it guess b' correctly with more than a negligible advantage.

hybrid proof. Let H_0 be the event that $b = b'$ in *GAME 0* where σ^b is the scenario chosen by the challenger and b' be the guess made by A the adversary. Similarly, let H_1 be the same even but for *GAME 1*, and H_2 for *GAME 2*.

Lemma 7. $\Pr[H_2] = \frac{1}{2}$.

Proof. In *GAME 2*, the outputs of π are unrelated to the chosen scenario. $\pi.client.retrieve$ requests are not related to any scenario-defined client deaddrop. Furthermore, the messages are encryptions of uniformly random messages instead of the message provided by $\sigma^b[\cdot].getMessage(rnd)$. As a result, the requests the adversary observes have no relation to the scenarios at all. A's best action to win is to choose b like the challenger, with a coin flip. □

Lemma 8. $|\Pr[H_2] - \Pr[H_1]| \leq \epsilon_{CPIR}$.

Proof. The difference between H_1 and H_2 is the method $\pi.clientoutput.retrieve(client_i, client_j)$. In H_2 the deaddrop index is random, and in H_1 it is provided by the scenario (i.e., $client_j$'s deaddrop is used to generate the query). However, given the assumption that the CPIR scheme is secure, the advantage of an efficient adversary to distinguish which deaddrop index is requested is ϵ_{CPIR}, which is negligible. □

Lemma 9. $|\Pr[H_1] - \Pr[H_0]| \leq \epsilon_{AES_{ENC}}$.

Proof. In H_1, the method $\pi.client.send(\cdot)$ encrypts randomly generated message m' using AES and the shared secret key $ssk_{i,j}$ of $client_i$ and client $client_j$ generated in the setup. While in H_0, it encrypts the message m specified in scenario σ^b. Because σ^b describes only honest clients, we need to inspect two cases: $client_j$, the recipient is honest, and another case where it is compromised. Assuming the recipient is honest, the adversary doesn't have access to the shared secret key used to encrypt the message and cannot decrypt the messages. Furthermore, A cannot distinguish between ciphertexts of the two scenarios (i.e., AES is semantically secure).

plaintext-ciphertext multiplication	Compute time (ms)
vanilla [42] (not associative)	0.060546
naive solution (via ctxt-ctxt mult.)	0.783706
DPIR's modification	0.142784

Table 2: Cost of BGV plaintext-ciphertext multiplication in the SEAL library [42] and DPIR's modified implementation.

Thus an efficient adversary gains $\epsilon_{AES_{ENC}}$ advantage at most, which is negligible.

Now let us observe the second case, where the recipient $client_j$ is corrupted by the adversary A. Because $client_j$ is compromised, the adversary A has access to the shared secret of $client_i$ and $client_j$. This means that A can decrypt the message, and attempt to distinguish between the scenarios based on the difference between the messages. We remind the reader, at this time, about the restrictions of the adversary, particularly that A is restricted to provide both scenarios with the same message at the respective round and the same friend when the recipient is malicious §B.1.1. That is, exposing the message exchanged between the honest and malicious client does not help the adversary distinguish between the scenarios. thus it does not give the adversary any advantage. □

Combining the lemmas, it holds that

$$|\Pr[H_0] - \frac{1}{2}| \leq \epsilon_{CPIR} + \epsilon_{AES_{ENC}}$$

As wanted.

□

C Using the SEAL cryptography library in DPIR

To use SEAL [42] in DPIR we modified its ciphertext-plaintext multiplication procedure, as SEAL's native implementation of BGV does not ensure associativity of multiplication within the ciphertext space when multiplying ciphertexts with plaintexts with coefficients greater than $t/2$ (where t is the plaintext modulus, see §4), thus it is incompatible with Freivalds' algorithm (see §7.2). Our change does not have any implications on security since the multiplication algorithm does not deal with any secrets (such as private keys).

The issue is not inherent to the BGV encryption scheme, rather it stems from the way SEAL handles the disparity in the size of the plaintext modulus (t) and the coefficient modulus (q). Specifically, SEAL re-maps the plaintext polynomial coefficients using the residue number system (RNS) to support faster multiplication. Under this mapping, plaintext coefficients $\leq t/2$ stay the same in Z_q, but coefficients $> t/2$ map to a different number in Z_q.

One potential solution is adding the plaintext p to a "ciphertext" comprising zero polynomials, this "automatically" maps the plaintext to the ciphertext space (homomorphic plaintext-ciphertext addition results in a ciphertext), and then multiplying the resulting ciphertext encoding of p by the other ciphertext. This technique circumvents the plaintext-ciphertext multiplication issue, but it has a high cost since it relies on ciphertext-ciphertext multiplication which is about 12× slower than vanilla plaintext-ciphertext.

A more efficient solution, which we implemented, is modifying SEAL's plaintext-ciphertext multiplication procedure to split the plaintext p into two plaintexts p', p'' such that $p = p' + p''$, and each of their coefficients is less than $t/2$. Our modified SEAL implementation replaces all plaintext-ciphertext multiplications $p \cdot c$ in the original algorithm with $(p' \cdot c) + (p'' \cdot c)$. This solution is only 2× more costly than SEAL's native implementation (c.f., to 12× with the alternative solution). It is correct due to the polynomial ring distributivity property, and it is safe since it does not involve any secret keys. We benchmarked the performance of our implementation and compare it against the naive solution and SEAL's vanilla implementation in Table 2.

Mitigating memory contention. Many implementations of homomorphic encryption schemes, including SEAL's BGV, suffer from memory contention (e.g., as noted by Badawi et al. [2]). In most homomorphic encryption schemes, the ciphertexts are large (50KB with our standard BGV parameter choice, see below), so reading and writing them from/to RAM creates contention and fitting many of them in the cache memory is challenging. We confirmed this issue while testing our code. DPIR naturally mitigates this problem since it also leverages the *independent memory* resources across all its clients when outsourcing PIR computations.