Pirmission: Single-server PIR with Access Control

Andrew Beams
University of Pennsylvania

Sebastian Angel
University of Pennsylvania and Microsoft Research

***Abstract*—Databases often require the flexibility to control which entities can access specific database records. Such access control is absent in works that provide *private* access to databases, namely private information retrieval (PIR) systems. In this paper, we show how to address this shortcoming by introducing Pirmission, the first practical single-server PIR system that allows the enforcement of access control policies. Pirmission's mechanism does not even reveal whether the client passed or failed the access control check—instead the client receives random data if they are not authorized to access a database record. To demonstrate the usefulness and practicality of Pirmission, we use it to build a private contact discovery platform that allows users to only be discoverable by their friends (who have permission). Compared to state-of-the-art single-server PIR protocols that do not provide access control, Pirmission increases the server's response time by around 2.8× (much less for databases with large records), and requires only one additional ciphertext to be sent by the client.**

1. Introduction

Databases are a fundamental component of software systems. In many applications, one would like to hide the users' specific queries from the database server itself. For example, users of a subscription service (e.g., news site) might wish to keep their content consumption history hidden from the provider. This is the purpose of *private information retrieval* (PIR) [26]—a client can query for an element at some index (or in some variants with a keyword [25]) without the server being able to determine which element was requested. Databases often also require some form of access control. For example, a bank requires consent by a loan applicant before accessing the applicant's credit report from a credit bureau database; social media services require permissions (e.g., established friendship, group membership) before showing certain posts; subscription services require memberships before accessing premium content. Variants of these applications that wish to hide access patterns still need to ensure that a client can only access the data for which they have permission.

The problem and setting. Combining access control with PIR is challenging because the interplay between the two desired properties (control and privacy) is actually quite subtle; straightforward solutions are either broken (often in non-obvious ways) or too expensive to be useful in practice. Consider the following strawman which explains our setting and also highlights some of the issues: one or more *data producers* encrypt records that they then upload to a database maintained by an untrusted database operator (e.g., a cloud provider). The data producers then give *clients* the decryption key(s) to the records in the database that they are allowed to access. A client then issues a standard PIR query to the database and obtains an encrypted record, which they can decrypt only if they have access.

At first glance this proposal appears reasonable, but it has one of the following two issues.

Lack of forward secrecy. A client can issue a PIR query for a record for which they do not have access at time t_0. Suppose a data producer then changes the content of the record (e.g., declassifies a redacted version of a previously classified file) at time t_1, and then gives access to the client at time t_2. The client can then decrypt the previously (unredacted) record (acquired at t_0) even though it never had access to it. One possible way to address this is by changing the encryption key each time a record is updated. Besides the obvious complexity of managing changing keys for each update, this proposal leaks information, as follows.

Lack of access control. A client issues a PIR query for a record for which they do not have access at time t_0, and gets back a ciphertext c encrypted under a key the client lacks. Suppose a data producer updates the record at time t_1 but the client is not given access to this new record. The client then issues a PIR query for the record again at time t_2 and gets the updated ciphertext c'. Even though the client had no access to the record, they were able to observe that the record has changed since the last time they queried it. This leaks metadata about records, which existing access control mechanisms on non-PIR databases would not leak.

An alternative that achieves access control and forward secrecy is as follows. Data producers give to the database operator one or more access tokens for each record. A client issues a query by submitting a standard PIR query and a zero-knowledge proof that it knows a valid access token for the index being queried (without revealing the index). If the proof is valid, the database operator processes the PIR query; else, the database operator rejects the query. There are many challenges in instantiating this high-level approach such as making the proofs efficient, but a key one is that the authorization check can leak information, as follows.

Lack of authorization privacy. The database operator learns whether a client has permission to access a record or not (even if it does not know which). This property is crucial, as otherwise a database operator could, for example, change the access tokens associated with a particular record

to some invalid value. If the client's proof still passes, then the operator learns that the client was not requesting that particular record. Note that the client cannot tell whether the database operator changed an access token or not, as doing so would leak when a data producer changes an access token—this violates access control, as described earlier.

Our solution. This paper introduces *Pirmission*, a PIR system that extends existing efficient computational single-server PIR schemes [4, 8, 9, 54, 55] to provide access control, forward secrecy, and authorization privacy. In addition, Pirmission preserves PIR's standard properties of *correctness* (the protocol retrieves the record at the desired position), *query privacy* (the database operator does not learn anything about the queried index), and *non-triviality* (the total communication costs are sublinear in the number of records in the database). The key idea behind Pirmission is to perform the access control and process the PIR query *simultaneously*, in such a way that if a client has the right to access a record at a given index it receives the record's contents, and otherwise it receives random data; the database operator does not learn which of these two events took place.

In more detail, the database consists of a *token table* and a *data table*. Data producers store access tokens (in plaintext) in the token table, and clients can query the database using PIR in addition to sending an encrypted access token. If the provided encrypted token matches the entry in the token table for the specific index (which Pirmission can check by reusing the PIR query on the token table), Pirmission's response (from the data table) is as expected; otherwise the response is masked by a random pad. Pirmission's mechanisms are orthogonal to the nature of the access tokens: they could be random strings, derived from ABE credentials [11, 62], etc.

We implement Pirmission on top of SealPIR [9]. When records are very small, Pirmission requires twice as much server storage (for the token table), $2.8\times$ longer for the server to produce a response, and a small increase in query size (one extra ciphertext). When records are large, Permission's overhead over the baseline is under 25%. Despite these added costs, to our knowledge, Pirmission is the first construction and implementation of a single-server PIR with access control system (as we discuss in Section 2, there are prior *oblivious transfer* protocols with notions of access control but they have linear communication, lack forward secrecy, and are not implemented).

Applications. Pirmission is useful for any situation where one wants a database to preserve access control while still keeping queries private. Further, Pirmission can be used as a building block to support other useful applications. As one example, we consider the case of *private contact discovery* (PCD), whereby a user wants to figure out which of their friends are part of of one or more services (e.g., social network, chat app, video game) without revealing to the operator of these services their friends' identities. PCD designs [12, 32, 39, 41, 53] typically use phone numbers as identifiers. A new user has a phone number for each contact, and the application has a large database of the phone numbers of its users. The client and the service interact to discover which phone numbers they have in common.

An issue with existing PCD systems is that phone numbers are not private information. A malicious user could discover if another person is using some service without that person's consent, simply by looking them up on that service's PCD. This could be problematic in settings such as social networks for particular religious or ideological views, sexual orientation, or demographics.

To address the above, we use Pirmission to build a *authenticated* version of PCD that allows users to restrict who can discover them in the different services. This system can be used in combination with metadata-private messaging [7, 10, 43, 46, 47, 66, 67] to further prevent the services themselves from learning which users are friends with each other while still providing messaging and voice chat functionality.

In summary, this work's contributions are:

- A model that combines private queries and access control in real settings (§3).
- Pirmission, which extends PIR with access control (§4).
- The design of a authenticated private contact discovery scheme based on Pirmission (§5).
- The implementation of Pirmission and authenticated PCD, and their experimental evaluation (§7–8).

2. Background and related work

Private information retrieval (PIR) [26] is a cryptographic primitive in which a client makes a query to a database server to obtain the object at a specific index without the database learning the index queried and with communication costs being sublinear in the number of elements in the database. *Single-server* PIR [42] is a variant of PIR in which the privacy guarantee is provided under computational assumptions, and has the benefit of being useful even when the database is operated by a single entity. In recent years there has been a resurgence of single-server PIR protocol that provide low communication and computation costs in practice [4, 5, 8, 9, 54, 55]. This line of work serves as the foundation of our proposal; we provide the necessary details in later sections.

Another line of relevant work is that of (1-out-of-n) *oblivious transfer* (OT) [61]. OT provides a similar privacy guarantee to PIR, but it does not require sublinear communication and has the additional requirement that the client can only fetch information about one element. In a sense, this is a type of basic access control, though we aim for more meaningful capabilities. A variant of both OT and PIR is known as *symmetric PIR* [36]: it provides the aforementioned access control property of OT, and the sublinear communication requirement of PIR. There are even transformations that can convert a single-server PIR scheme into a symmetric PIR (and therefore an OT) scheme [6, 56].

Most closely related to our work are OT schemes that support more general access control policies [3, 17–19, 48, 68]. These proposals have two drawbacks when used in our

setting since they require sending an encrypted version of the *entire* database to the client at the beginning. First, they do not support removing or updating elements: if elements change then either the client learns about which specific element was updated (a leak in the database's metadata), or the protocol must be re-run from scratch (expensive and also leaks metadata). Second, they incur communication that is linear in the database size and hence not practical.

There are also symmetric PIR schemes with access control [44, 45] but they rely on a very specific symmetric PIR protocol [49] that is not efficient in practice (we compare with these works in Section 8). Further, the model used in these schemes requires that a data producer give a client a new token for each successive query. That is, if an element changes frequently and a client wants to query it multiple times, the data producer must provide a unique token each time. In contrast, our work is compatible with state-of-the-art efficient PIR schemes and requires the client to get new tokens only when the data producer wants to change the access policy (allow the client to access data it could not access before or revoke the access from the client).

3. Setting and threat model

In this section we discuss the setting, properties, and threat model for Pirmission. The goal is to design a scheme that is practical, has access control capabilities, and supports the requirements of real databases (e.g., supports updates and deletions). For now we will assume that access tokens are distributed to clients out of band. We discuss some possibilities for token distribution in Section 6.

3.1. Setting

There are three types of parties: a *database server*, one or more *data producers*, and one or more *clients*. The server could be, for example, a cloud provider. Data producers are parties who add, delete, and modify data in the database. The server could be a data producer as well. Finally, clients are users who access the database and fetch data from it. For example, they might be users in a chat application or front-end servers of some Web application (running in a different administrative domain than the database).

A database consists of two tables: a *data table*, denoted D, and a *token table*, denoted T. Each of these tables has n elements. Data elements and tokens are both from the same plaintext space $\mathbb{M}$. We use $[n]$ to denote the set $\{1, \ldots, n\}$, and let the data element at index $i \in [n]$ be D_i, and its associated token be T_i. We assume that the producer who stores a data element at some index in the data table also stores the associated token at the same index in the token table. Producers may modify either table at any time.

A client can send a query to the database server at any time, and will receive a response according to the properties that we describe in the next section. The query process involves four functions. Three of these functions are run by the client and one is run by the server:

- $\text{KeyGen}(1^\lambda) \to (pk, sk)$, a randomized algorithm executed by the client that takes as input a security parameter λ and generates an encryption and decryption key pair.
- $\text{Query}(pk, i, t) \to q$, a randomized algorithm executed by the client which takes an encryption key pk, an index i, and a token t, and outputs a query q.
- $\text{Respond}(D, T, q) \to c$, a randomized algorithm executed by the server that takes the client's query and the data and token tables, and outputs encrypted response c.
- $\text{Extract}(sk, c) \to x$, a deterministic algorithm executed by the client that takes the decryption key sk and uses it to extract the data x from the server's response c.

3.2. Properties

Correctness. A client who provides the correct token with their query obtains the correct element. That is:

$$pk, sk \leftarrow \text{KeyGen}(1^\lambda)$$
$$\text{Extract}(sk, \text{Respond}(D, T, \text{Query}(pk, i, T_i))) = D_i$$

Query Privacy. The server is unable to distinguish a query for index i from a query for index j. Formally, for any PPT adversary $\mathcal{A}$, security parameter λ, and any pair of indexes i and j in $[n]$:

$$pk, sk \leftarrow \text{KeyGen}(1^\lambda)$$
$$q_0 \leftarrow \text{Query}(pk, i, T_i); \quad q_1 \leftarrow \text{Query}(pk, j, T_j)$$
$$|\Pr[\mathcal{A}(q_0) = 1] - \Pr[\mathcal{A}(q_1) = 1]| \leq \text{negl}(\lambda)$$

Authorization Privacy. The server does not learn if a client's query succeeds or fails because of improper authorization. That is, for any PPT adversary $\mathcal{A}$, security parameter λ, and any index $i \in [n]$ and token $t \in \mathbb{M}$:

$$pk, sk \leftarrow \text{KeyGen}(1^\lambda)$$
$$q_0 \leftarrow \text{Query}(pk, i, T_i); \quad q_1 \leftarrow \text{Query}(pk, i, t)$$
$$|\Pr[\mathcal{A}(q_0) = 1] - \Pr[\mathcal{A}(q_1) = 1]| \leq \text{negl}(\lambda)$$

Access Control. A client without the token for index i (i.e., T_i), cannot obtain any information about D_i by querying the database. We formalize this as follows. Let the following probability distribution capture the view of the client when querying a database with data table D, token table T, for index $i \in [n]$ with token $t \in \mathbb{M}$:

$$\text{View}_{D,T,i,t} := \left\{ q, c, x : \begin{array}{r} pk, sk \leftarrow \text{KeyGen}(1^\lambda) \\ q \leftarrow \text{Query}(pk, i, t) \\ c \leftarrow \text{Respond}(D, T, q) \\ x \leftarrow \text{Extract}(sk, r) \end{array} \right\}$$

If $t \neq T_i$, then $\text{View}_{D,T,i,t} \approx \text{View}_{D',T,i,t}$, where D' is a uniformly random data table. That is, the view of the client when interacting with the server without the right token is statistically indistinguishable from a view when the client interacts with a database of uniformly random entries.

Forward Secrecy. After a data producer modifies D_i, a client cannot obtain the old value of D_i even if the client

acquires the associated token T_i. We formalize this as follows. Let $\text{Update}(D, i, v) \to U$ be a procedure called by the server that updates the i-th entry in the data table with a new value v. Let the following probability distribution capture the view of the client during the following events: (1) client issues a query to the database with data table D, token table T, for index i with token t; (2) server updates an entry in D; (3) client obtains a copy of the token T_i and the updated object U_i.

$$\text{View}_{D,T,i,t} := \left\{ q, c, x, U_i, T_i : \begin{array}{r} pk, sk \leftarrow \text{KeyGen}(1^\lambda) \\ q \leftarrow \text{Query}(pk, i, t) \\ c \leftarrow \text{Respond}(D, T, q) \\ x \leftarrow \text{Extract}(sk, r) \\ U \leftarrow \text{Update}(D, i, \cdot) \end{array} \right\}$$

If $t \neq T_i$, then $\text{View}_{D,T,i,t} \approx \text{View}_{D',T,i,t}$, where D' is a uniformly random data table. This definition can be trivially extended to model forward secrecy for T as well (i.e., if a data producer changes T_i, a client without knowledge of the old T_i does not learn that this change took place).

Non-triviality. The total communication cost between the client and the server must be smaller than the database. Specifically, it must be sublinear in n (number of records).

3.3. Threat Model

Query privacy and authorization privacy are guarantees given to the client against the database server. Access control and forward secrecy are guarantees given to the server (or data producers) against unauthorized clients. Consequently, our threat model is as follows: we assume the database server wants to violate query privacy and authorization privacy and can act arbitrarily maliciously in order to do so. Likewise, we assume that the client wants to violate access control and forward secrecy and may act arbitrarily, including issuing malformed queries.

Note that in PIR protocols, correctness and non-triviality are properties of the protocol itself and hold only when all parties behave honestly. The reason is simple: a malicious server or client could easily send a lot of data (violating non-triviality) or garbage (violating correctness).

The case where the data producer colludes with the client is not interesting, as the data producer could simply give the client the data or the token. The case where the data producer colludes with the server means that the pair becomes aware of which elements the client has access to; so the protocol must simply ensure that the data producer and the server cannot obtain more than this information when they collude. This is equivalent to query privacy.

4. PIR with Access Control

In this section we give our construction of Pirmission. We begin with some background on recent single-server PIR schemes on which we build our work, and then introduce our extensions. A key aspect of Pirmission is its simplicity. Not only does this simplicity translate into lower concrete costs, it also makes our approach easy to understand, prove, and port to future protocols.

```
1: function PIR-RESPOND(D, Q)
2:     c ← PCMULT(D_1, Q_1)
3:     for j ∈ [2, ..., n] do
4:         c ← HOMADD(e, PCMULT(D_j, Q_j))
5:     return c
```

Figure 1: Respond function in Stern's PIR. D is the database represented as a vector of plaintexts. Q is the client's query and consists of a vector of ciphertexts constructed according to Section 4.1. HOMADD is homomorphic addition, and PCMULT is plaintext-ciphertext multiplication.

4.1. Single-server PIR

We summarize the idea behind the scheme introduced by Stern [65] and used in recent works [5, 8, 9, 54, 55]. It assumes an *additively homomorphic* encryption scheme.

Notation. We use lowercase letters (e.g., x) to denote single elements (plaintext or ciphertext) and uppercase (e.g., Q) to denote vectors of elements. We index vectors with subscripts (e.g., Q_i). We denote the encryption of a plaintext x with encryption key pk as $\text{Enc}_{pk}(x)$. Likewise we denote the decryption of a ciphertext y with decryption key sk as $\text{Dec}_{sk}(y)$. The cryptosystem is additively homomorphic if there exists an operation HOMADD such that $\text{HOMADD}(\text{Enc}_{pk}(x), \text{Enc}_{pk}(y)) = \text{Enc}_{pk}(x + y)$. Additively homomorphic cryptosystems also support plaintext-ciphertext multiplication with some operation PCMULT, such that $\text{PCMULT}(x, \text{Enc}_{pk}(y)) = \text{Enc}_{pk}(xy)$.

Construction. Let D be an n-element database where each element is in the plaintext space of the homomorphic cryptosystem $\mathbb{M}$ (we discuss the specifics in Section 4.2).

- $\text{KeyGen}(1^\lambda)$: generate a pair of encryption and decryption keys, pk and sk, for the homomorphic cryptosystem with security parameter λ.
- $\text{Query}(pk, i)$: to query the element at position i, generate an encrypted query vector Q of length n with $Q_i = \text{Enc}_{pk}(1)$ and for all $j \neq i \in [n]$, $Q_j = \text{Enc}_{pk}(0)$. Note that traditional PIR schemes lack access control, so there is no token here.
- $\text{Respond}(D, Q)$: compute the dot product of D and Q and return the resulting ciphertext c to the client. Note that this requires only plaintext-ciphertext multiplications and ciphertext-ciphertext additions, as shown in Figure 1.
- $\text{Extract}(sk, c)$: client can decrypt c with the cryptosystem's decryption key to obtain $D_i = \text{Dec}_{sk}(c)$.

A limitation of this construction is that communication costs are linear in n since the query has n entries. This can be addressed by representing the database D as a hypercube. We discuss this in detail in Section 4.7.

There are many optimizations to further reduce costs. For example, XPIR [4] shows that if the homomorphic cryptosystem is based on Ring-LWE cryptosystems [13, 16, 33],

```
1: function PIRMISSION-RESPOND(D, T, Q, Enc_pk(t))
2:     c ← PIR-Respond(D, Q)                    // See Figure 1
3:     t' ← PIR-Respond(T, Q)                   // See Figure 1
4:     // at this point c = Enc_pk(D_i) and t' = Enc_pk(T_i)
5:     d ← HOMSUB(t', Enc_pk(t))
6:     // d = Enc_pk(0) if and only if token is correct
7:     r ←R M
8:     c ← HOMADD(c, PCMULT(r, d))
9:     return c
```

Figure 2: Pseudocode for Pirmission's Respond function. D and T are the data and token table, respectively. Q is the query vector for index $i \in [n]$, and $\text{Enc}_{pk}(t)$ is an encryption of the token $t \in \mathbb{M}$ for the requested element. $\mathbb{M}$ is the plaintext space of the homomorphic encryption scheme. Pirmission uses BFV [13, 33], so $\mathbb{M}$ consists of polynomials from a quotient ring with integer coefficients.

then one can preprocess the database using the number theoretic transform so that computing PCMULT is much more efficient. SealPIR [9] then shows how to compress an entire query vector Q into a single ciphertext q, rather than computing one ciphertext for each entry. Pirmission uses the BFV cryptosystem, so we apply these and other optimizations in our implementation.

4.2. Adding access control

Pirmission processes the PIR query and the access control check simultaneously to achieve all of the desired properties (§3.2). The KeyGen and Extract algorithms are the same as in the previous section. Below are the new algorithms for Query and Respond.

- $\text{Query}(pk, i, t)$: in addition to the encrypted query vector (see Query in Section 4.1), the client generates ciphertext $\text{Enc}_{pk}(t)$, which is an encryption of the token $t \in \mathbb{M}$ associated with the element at position i. Query outputs the tuple $(Q, \text{Enc}_{pk}(t))$.
- $\text{Respond}(D, T, Q, \text{Enc}_{pk}(t))$: the server computes two dot products, one between the query vector Q and the data table D and another between Q and the token table T, resulting in $\text{Enc}_{pk}(D_i)$ and $\text{Enc}_{pk}(T_i)$, respectively. The server then chooses an element r uniformly at random from the encryption scheme's plaintext space $\mathbb{M}$ and returns $c = r \cdot (\text{Enc}_{pk}(T_i) - \text{Enc}_{pk}(t)) + \text{Enc}_{pk}(D_i)$, where "$\cdot$", "$-$", and "$+$" are PCMULT, HOMSUB, and HOMADD respectively. If the token given by the client, t, matches the one in the token table, $c = \text{Enc}_{pk}(D_i)$. Otherwise, r ensures that c is an encryption of a uniformly random plaintext. We give the pseudocode in Figure 2.

We can now reason about the properties outlined in Section 3.2. Correctness follows immediately: if the client supplies the right token and the server performs the protocol as in Figure 2, there will be no mask and the client will receive the right entry. Query privacy and authorization privacy also hold. The client's query consists of two parts: (1) the standard PIR query vector Q; and (2) one encryption of the token t, namely $\text{Enc}_{pk}(t)$. Past work [4] proves that the query vector for an element i is computationally indistinguishable from a query vector for any other element. This is not surprising since the query vector is just a vector of ciphertexts, each of which is semantically secure. Similarly, the encryption of the token is computationally indistinguishable from the encryption of any other value.

Note that in PIR, clients can *never* provide feedback to the server (i.e., tell the server whether their query failed for any reason). If the client provides feedback the server could substitute some of the entries in the database with garbage, thereby causing the client to complain only when it fetches one of those entries—this would leak which elements the client was or was not trying to access. Given the absence of feedback, the semantically secure ciphertexts are the only input sent from the client to the server. Furthermore, the server's logic is the same regardless of whether a client's encrypted token is correct or not. As a result, the server cannot distinguish a successful query from one with an invalid token (otherwise one could use such a distinguisher to violate the security of the underlying cryptosystem).

Below we prove that Pirmission also guarantees access control and forward secrecy. We defer non-triviality to later, after we explain how Pirmission makes communication costs sublinear (based on prior work).

Lemma 1. For any data table D, token table T, each consisting of n elements in $\mathbb{M}$, for any index i, and for any token $t \in \mathbb{M}$, if $t \neq T_i$ then Pirmission's respond procedure in Figure 2 guarantees that $\text{View}_{D,T,i,t} \approx \text{View}_{D',T,i,t}$, as defined in the **access control** definition of Section 3.2. Here D' is is a data table of n elements, each of which is independently and uniformly sampled from $\mathbb{M}$.

Lemma 2. For any data table D, token table T, each consisting of n elements in $\mathbb{M}$, for any index i, and for any token $t \in \mathbb{M}$, if $t \neq T_i$ then Pirmission's respond procedure in Figure 2 guarantees that $\text{View}_{D,T,i,t} \approx \text{View}_{D',T,i,t}$, as defined in the **forward secrecy** definition of Section 3.2. Here D' is is a data table of n elements, each of which is independently and uniformly sampled from $\mathbb{M}$.

Proof. The proof for both Lemma 1 and 2 is the same. We claim that if $t \neq T$, the server's response depends only on the public parameters, namely the plaintext space $\mathbb{M}$. Consider an alternate Respond protocol that returns $c' = \text{Enc}_{pk}(r')$ where r' is a random value uniformly sampled from $\mathbb{M}$. In Pirmission, when $t \neq T$, the distribution of c' is statistically indistinguishable from the actual distribution of c (the output of Respond in Figure 2).

Suppose we generate c' as above. The server's actual response is computed as $c = \text{Enc}_{pk}(r \cdot (T_i - t) + D_i)$. Substituting c' for c, we get:

$$\begin{aligned} \text{Enc}_{pk}(r') &= \text{Enc}_{pk}(r \cdot (T_i - t) + D_i) \\ r' - D_i &= r \cdot (T_i - t) \\ r &= (r' - D_i) \cdot (T_i - t)^{-1} \end{aligned}$$

This is defined as long as $T_i - t$ is invertible. □

There are two things to note about this proof. First, it is predicated on $(T_i - t)$ always having a multiplicative inverse, but this not the case in the cryptosystems used by state-of-the-art PIR schemes where $\mathbb{M}$ is a ring. Second, the proof shows that if the token is incorrect (i.e., $t \neq T_i$), then the client learns nothing about D_i from the response. However, what if the client guesses T_i by chance? In this case, the client learns D_i because it can trivially confirm that it has guessed the correct token. For example, the client can issue multiple queries with the same (correct) token and the same index; since the token is correct, there is no mask applied to the response so the client obtains the same result each time—confirming its guess. Hence, the lemmas are only meaningful if the space from which tokens are sampled is large enough to make brute force guessing intractable.

We address both of these concerns next.

4.3. Guaranteeing the correctness of the mask

Pirmission builds on SealPIR [9] which uses the BFV [13, 33] cryptosystem (most recent PIR schemes use BFV or a similar cryptosystem, and Pirmission's key idea applies to them as well). In BFV, plaintexts are polynomials with integer coefficients from the ring $R_p = \mathbb{Z}_p[x]/(x^N+1)$, where N is a power of 2 and p is the *plaintext modulus* that specifies how much data can be stored in each coefficient of the polynomial. Both N and p must be such that, along with a careful selection of the ciphertext space, satisfy the decisional Ring-LWE assumption [50]. Ciphertexts are also polynomials of a different ring, but we will omit the details since they are not relevant to our discussion.

Encoding data elements. While the plaintext space of these cryptosystems is the polynomial ring R_p, objects in a database are not elements of this ring. Instead, they are usually an HTML document, a PDF, or some arbitrary binary blob. To use PIR systems based on BFV data producers need to encode such data as one or more polynomials in R_p. The standard approach is to split the database record into small chunks, so that each chunk—when interpreted as an integer—is smaller than p. In that way, one can simply create a polynomial in R_p where each of its coefficients corresponds to one of the chunks; one then adds this polynomial to the PIR database. When a client fetches this polynomial using PIR, it extracts each of the coefficients from it and reconstructs the original object (e.g., HTML document, binary blob).

Problem with tokens. Observe that for the masking operation to be well defined in Line 8 of Figure 2, the tokens must be from the same plaintext space as the data in the PIR database. That is, they must also be from R_p. The challenge is that if T_i and t are both from R_p, then $(T_i - t)$ does not always have an inverse. This is problematic since then we cannot use our proof of Lemmas 1 and 2 to argue that Pirmission's masking approach perfectly hides D_i when the token supplied by the client is incorrect.

To address this, Pirmission restricts p to be a prime congruent to $1 \pmod{2N}$ and uses *Chinese Remainder Theorem* (CRT) batching [14, 64]. We defer the details of CRT batching and its application to BFV to other documentation [22], but the following key property is useful in understanding why it helps. With CRT batching, a BFV plaintext polynomial actually encodes a 2-by-$(N/2)$ matrix, where each element is in $\mathbb{Z}_p$. All ciphertext operations (HOMADD,HOMSUB,PCMULT) then act element-wise. For example, if $p = 17$ and $N = 8$:

$$\mathrm{Enc}_{pk}\left(\begin{bmatrix} 0 & 2 & 6 & 12 \\ 3 & 13 & 8 & 5 \end{bmatrix}\right) = \mathrm{PCMULT}\left(\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}, \mathrm{Enc}_{pk}\left(\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix}\right)\right)$$

In a sense, with CRT batching we are morally transforming the plaintext space from the ring R_p to a matrix of independent elements, each in the finite field $\mathbb{Z}_p$.

CRT encoding, tokens, and masks. To encode the data, the data producer will do the same thing as before: split their database object into small chunks where each chunk is smaller than the integer p, and store each chunk in one of the entries in the matrix. The same idea for the tokens. Therefore, a token will effectively be a matrix where each of its entries will be sampled uniformly from $\mathbb{Z}_p$. As a result, when we perform the masking procedure in Figure 2, operations will act on each entry of the matrix as we discussed earlier. And since each entry in T_i and t is defined in the finite field $\mathbb{Z}_p$, every entry in $T_i - t$ is guaranteed to have an inverse unless $T_i[x, y] = t[x, y]$, where x denotes the row and y denotes the column of the matrix. In such case, the client has provided the correct entry in the token matrix, and Pirmission does not mask the corresponding chunk of D_i, namely $D_i[x, y]$—as expected.

4.4. Preventing brute force guessing of tokens

We now return to the second concern raised at the end of Section 4.2: if tokens are sampled from a small space, a malicious client can just guess the token and then confirm if their guess is correct. This confirmation can be done by querying a few more times with the same index token and token; if all responses are the same, then the client gains confidence that its guess is correct (since it would give evidence that there is no randomized mask applied). In our case, each token is technically made up of N independent sub-tokens, each of which is sampled from $\mathbb{Z}_p$. If we set p to be large enough, say a 60-bit prime, then we are done: we can guarantee that the client cannot guess and verify any of the sub-tokens except with negligible probability. Note that 60 bits is more than sufficient as a security parameter because, unlike in other settings, the client must issue a PIR query for each guess. With 60-bit tokens, the expected number of PIR queries required to guess and verify a given sub-token is over a *quintillion*. Not only is this more than any client can issue, but also no server could ever process that many PIR queries given PIR's high computational costs.

Unfortunately, setting BFV's plaintext modulus to a 60-bit prime incurs high costs for the PIR part of the protocol (Lines 2 and 3 in Figure 2): it requires ciphertexts to be

larger (be made up of many polynomials) to ensure that the *noise*[1] is low enough to allow for correct decryption after homomorphic operations. Indeed, to remain competitive in performance with state-of-the-art PIR systems we must use values of p that are around 15 to 20 bits. This means that a malicious client could determine each sub-token by making 2^{14} queries in expectation, which makes the system insecure.

To address this, Pirmission ensures that multiple sub-tokens contribute to the mask of each entry in D_i, effectively increasing the "logical" size of sub-tokens.

Naive solution. A simple way to achieve this is to have 4 token databases and to have the client submit 4 encrypted tokens. Then, one can do PIR on each of the token databases, perform the homomorphic subtraction, multiply by a random plaintext, and then apply 4 independent masks to D_i homomorphically (essentially Lines 3–8 in Figure 2 but with 4 different token databases). This works because for an entry at a given row and column of D_i, if any of the 4 sub-tokens provided by the client for that row and column is incorrect, then the entry is blinded by a uniformly random mask. Hence, the only way to get any information about D_i is to supply the 4 correct 15-bit sub-tokens—which is akin to guessing a single 60-bit sub-token.

Actual design. Instead of having 4 token databases and asking the client to provide 4 encrypted tokens which significantly increases computation and communication costs, we can achieve the same effect with just a single token database and a single encrypted token by using an automorphism supported by BFV and related cryptosystems called *slot rotations* [35]. Here the notion of a "slot" refers to an entry in the matrices produced by the CRT batch encoding.

Definition 1 (Rotation)**.** Given a ciphertext c which encrypts a plaintext x, and given an integer $0 < j < N/2 - 1$, the operation $\text{HomRot}(c, j)$ produces an encryption c' of the plaintext x' which contains every entry in x but rotated j positions to the right (rows rotate independently and entries wrap around). For example, if $N = 8$ and $j = 1$, then:

$$x = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}, \quad x' = \begin{bmatrix} 4 & 1 & 2 & 3 \\ 8 & 5 & 6 & 7 \end{bmatrix}$$

The key idea of using rotations is to observe that if a client does not guess every sub-token correctly, then $T_i - t$ will contain non-zero entries. With rotations, the server can ensure that these non-zero entries "cover more ground" and help mask more than their respective entry of D_i. Figure 3 gives the extended Pirmission Respond pseudocode which applies s masks (when p is 15 bits, $s = 4$ is enough).

One technical detail worth mentioning is that rotations require the client to supply a *Galois key* (also known as a *key switching matrix*). The reason for this is that when the server performs, for example, $c' \leftarrow \text{HomRot}(c, 1)$, the result c' is technically an encryption of the rotated plaintext but *under a different secret key* than c. So one cannot do any homomorphic operations that use both c and c' since both ciphertexts are defined under different keys. Instead, one uses the Galois key to "switch" the key of c' back to the original key under which c is encrypted. This operation is known as *key switching* [15]. We abstract the rotation and its corresponding key switch with our HomRot operation.

The reason we discuss this low-level detail is that one might wonder: do these Galois keys introduce additional costs, and can a malicious client provide a bogus Galois key that undermines the correctness of the rotation (e.g., causing c' to not rotate at all or to have some or all entries be 0), thereby violating our goal of having each coefficient mask multiple slots. The answer to both of these questions is no. In terms of cost, the existing PIR schemes we consider [5, 8, 9, 55] all use rotations already and send to the server the appropriate Galois keys—Pirmission imposes no additional overhead. We discuss the issue of correctness in more detail in Appendix A, but the key idea is that: (1) since the rotation happens *before* the key switch the client cannot undo the rotation by providing a bad key; (2) since the client does not know the value of $T_i - t$ (unless it knows T_i in which case there is no need to perform this attack), it cannot produce a Galois key that transforms those entries into a value chosen by the client (e.g., 0).

```
1:  function PirMission-Respond2(D, T, Q, Enc_pk(t), s)
2:      c ← PIR-Respond(D, Q)                    // See Figure 1
3:      t' ← PIR-Respond(T, Q)                   // See Figure 1
4:      d_1 ← HomSub(t', Enc_pk(t))
5:      r_1 ←R M
6:      c ← HomAdd(c, PCMult(r_1, d_1))
7:      // Add s − 1 additional masks
8:      for j = 2 to s do
9:          d_j ← HomRot(d_{j−1}, 1)
10:         r_j ←R M
11:         c ← HomAdd(c, PCMult(r_j, d_j))
12:     return c
```

Figure 3: Pseudocode for Pirmission's Respond function with additional masks. This is identical to Figure 2 except for the for loop which depends on a new parameter s that indicates how many masks to add to the response. HomRot is a homomorphic rotation, as defined in Section 4.4.

4.5. Supporting small database records

If a database record is small, say a 280 byte message on Twitter, then one still needs to use an entire BFV plaintext to represent it. For $N = 4096$ and a 15-bit p, the 2-by-$N/2$ matrix can store up to 5 KB of data. To make good use of this space, a common approach in standard PIR (without access control) is to pack k database records into a single BFV plaintext. In this way, the database essentially "shrinks" (in the number of entries) by a factor of k. To do so, one can split the database record into chunks, where each chunk is an integer smaller than p. Then store each of these integers in a different entry of the matrix. If there is space left, one can add another database record until all of

1. In LWE-based homomorphic cryptosystems, encryption adds noise to the underlying plaintext. As operations are performed on ciphertexts (e.g., additions, multiplications), the noise grows. At some point, the noise is so high that one cannot recover the correct result when decrypting.

the entries of the matrix are filled or no more whole database records fit in the remaining entries. This can also be done in Pirmission, but there is the additional challenge of dealing with the tokens of different elements.

To explain how Pirmission packs data and tokens, we use an example. Let $N = 16, p = 97$, and $s = 2$ (so we want to have 2 masks per entry). Suppose each record occupies 4 entries in a BFV plaintext. This means that we can pack up to $N/4 = 4$ records per plaintext. Let the four records be x_1, x_2, x_3, x_4. Define the j-th chunk of record x_i as $x_{i,j}$. Likewise, we will have four tokens, one for each record: y_1, y_2, y_3, y_4, and the j-th sub-token of y_i is $y_{i,j}$. Pirmission constructs the packed plaintext for the data and token as:

$$D_i = \begin{bmatrix} x_{1,1} & x_{2,1} & x_{1,2} & x_{2,2} & x_{1,3} & x_{2,3} & x_{1,4} & x_{2,4} \\ x_{3,1} & x_{4,1} & x_{3,2} & x_{4,2} & x_{3,3} & x_{4,3} & x_{3,4} & x_{4,4} \end{bmatrix}$$

$$T_i = \begin{bmatrix} y_{1,1} & y_{2,1} & y_{1,2} & y_{2,2} & y_{1,3} & y_{2,3} & y_{1,4} & y_{2,4} \\ y_{3,1} & y_{4,1} & y_{3,2} & y_{4,2} & y_{3,3} & y_{4,3} & y_{3,4} & y_{4,4} \end{bmatrix}$$

Suppose a client is interested in record x_2. Then the client will supply, in addition to the PIR query that selects the appropriate packed plaintext from the database (i.e., that selects D_i), an encryption of the token t where:

$$t = \begin{bmatrix} 0 & t_1 & 0 & t_2 & 0 & t_3 & 0 & t_4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Observe Line 4 of Figure 3 when the server computes $d_1 = \text{Enc}_{pk}(T_i - t)$, and suppose the client correctly guesses $t_1 = y_{2,1}$, but its guess of t_2, t_3, and t_4 is wrong. Then d_1 will be an encryption of (the red entries are $y_{2,j} - t_j$):

$$\begin{bmatrix} y_{1,1} & 0 & y_{1,2} & y'_{2,2} & y_{1,3} & y'_{2,3} & y_{1,4} & y'_{2,4} \\ y_{3,1} & y_{4,1} & y_{3,2} & y_{4,2} & y_{3,3} & y_{4,3} & y_{3,4} & y_{4,4} \end{bmatrix}$$

This leads to the masking operation in Line 6 not correctly hiding the first chunk of the record, namely $x_{2,1}$, since the client correctly guessed its sub-token. However, in Line 9, Pirmission rotates d_1 by 2 positions to the right (instead of 1, as in the non-packed algorithm). In this case, d_2 will encrypt:

$$\begin{bmatrix} y_{1,4} & y'_{2,4} & y_{1,1} & 0 & y_{1,2} & y'_{2,2} & y_{1,3} & y'_{2,3} \\ y_{3,4} & y_{4,4} & y_{3,1} & y_{4,1} & y_{3,2} & y_{4,2} & y_{3,3} & y_{4,3} \end{bmatrix}$$

Hence, the operation in Line 11 will correctly apply a uniformly random mask to the entry at position $x_{2,1}$ and the client will not be able to obtain its value. The same is true for all other entries.

Note that if the client had provided the correct value for all sub-tokens, then all the red entries would have been 0 as well, and the rotations would not have had any effect.

Summary. To pack many small records into a single BFV plaintext, it suffices to split the records and spread the chunks across the matrix. The spreading should be done in such a way that one can pick an appropriate rotation value that ensures that only tokens of the same record interact with each other without affecting the tokens of other records.

4.6. Supporting large database records

In the case when database records are large, one could select large security parameters to ensure the BFV plaintexts can fit the large database records. However, this makes all the homomorphic operations very expensive. A more practical approach is as follows. The server splits the large records into k chunks, where each chunk fits within a single BFV plaintext. Then, the server creates k data tables, $D^1, \ldots, D^k$, and stores the j-th chunk of each record in data table D^j.

To query for the i-th database record, the client submits a single query vector Q and the encrypted token $\text{Enc}_{pk}(t)$ to the server. The server then computes the dot product between Q and each of the k data tables (i.e., PIR-Respond in Figure 1 with Q and D^j for $j \in [k]$) to obtain k (unmasked) responses: $c_1, \ldots, c_k$. The server then computes the dot product between Q and the token table T to obtain $\text{Enc}_{pk}(T_i)$. From this, the server computes the difference between the stored and the provided token $d = \text{HomSub}(\text{Enc}_{pk}(T_i), \text{Enc}_{pk}(t))$. Finally, the server samples k uniformly random BFV plaintexts, $r_1, \ldots, r_k \in \mathbb{M}$ and computes the k masked responses as: $c_j = \text{HomAdd}(c_j, \text{PCMult}(r_j, d))$ for $j \in [k]$. The client can recover D_i by decrypting each of the k responses with its decryption key sk, extracting the data from each BFV plaintext polynomial, and concatenating the results. This description is for $s = 1$, but it generalizes to multiple masks.

Note that Pirmission's server processes the token table once. As a result, Pirmission's server processes $k+1$ tables, whereas the baseline PIR scheme (with no access control) processes k. Indeed, Pirmission's relative overhead is highest when $k = 1$, and decreases as k increases.

4.7. Achieving sublinear communication

The PIR protocol described so far requires communication that is linear in the number of database records since the query vector Q contains as many entries as there are entries in the database. Pirmission adopts the following idea from Stern [65] to make costs sublinear. Instead of representing the data table D as a vector of n BFV plaintexts, Pirmission represents D as a matrix with $\sqrt{n}$ rows and $\sqrt{n}$ columns. Suppose the client is then interested in the plaintext at row i and column j. The client sends 2 query vectors, Q_{row} and Q_{col}, each of which contains $\sqrt{n}$ entries. The vector Q_{row} has the encryption of 1 at position i, while Q_{col} has the encryption of 1 at position j. The server, upon receiving Q_{row} and Q_{col}, computes the following matrix-vector product: $E_j = D \cdot Q_{col}$, where each multiplication is between a plaintext (elements in D) and ciphertexts (elements in Q), and additions are over ciphertexts. Observe that E_j is a vector of ciphertexts that contains the encryptions of the entries in column j of D.

The server can then compute a dot product between E_j and Q_{row} to obtain the result. There are two ways to do this. The first is to notice that E_j contains ciphertexts and Q_{row} also contains ciphertexts, so the multiplications and additions both need to be over ciphertexts which requires

the underlying cryptosystem to be additively and multiplicatively homomorphic (this is the case in BFV). This is what MulPIR [8] does, but the tradeoff is that security parameters are larger and the PIR computation is more expensive.

The second option, adopted by several schemes [4, 9, 55] and by Pirmission, is to think of each ciphertexts in E_j not as a ciphertext per se but just as a binary blob of data (essentially a random-looking database record). Then encode each binary blob into a fresh BFV plaintext. The catch is that ciphertexts are much larger than plaintexts (in BFV, they are around 10 times larger) and hence we need to use the technique in Section 4.6 to split the binary blob into k chunks, encode each chunk into a BFV plaintext, and put each BFV plaintext into a different data table $E_j^1, \ldots, E_j^k$. Finally, one can just run the PIR-Respond procedure of Figure 1 with each of these data tables and Q_{row}. This results in k encrypted chunks: $\mathrm{Enc}_{pk}(c_1), \ldots, \mathrm{Enc}_{pk}(c_k)$, where $\mathrm{Enc}_{pk}(D_{i,j}) = c_1 || \ldots || c_k$. The client can recover $D_{i,j}$ by decrypting each encrypted chunk, concatenating the result to form a ciphertext, and then decrypt that ciphertext. Pirmission then applies the optimization in SealPIR [10] to compress Q_{row} and Q_{col} so that each of them is a single ciphertext rather than a vector of ciphertexts.

In Appendix B we discuss how to add access control to this new representation. This enables Pirmission to achieve sublinear (and concretely low) communication and therefore satisfy the non-triviality property (§3.2).

5. Authenticated PCD

We demonstrate the flexibility of Pirmission by designing a authenticated variant of private contact discovery.

5.1. Private Contact Discovery (PCD)

When a user signs up for a service, the user and service need to discover which of the user's contacts are also on the same service. The non-private form of this contact discovery simply involves the user sending the service his entire list of contacts (e.g.., a phone's address book). The service can then scan its user database and discover any matches. However, this exposes information about the user's friends who have not consented to use that service.

Private contact discovery generally focuses on preventing this particular leakage—that of a user's friends who are not also users of the service. A user carries out some protocol with the service such that at the end, the user, or both parties, receive the intersection of the user's contact list and the service's user database. A common method is for the user to hash each of their contacts before sending it to the service. The service can then do the comparison using hashes. While this is efficient, it has two main flaws. First, the values being hashed are phone numbers, a small enough space to brute force. Second, the service can detect when two users share a contact, even if the service does not know the identity of this contact. Signal supports PCD via Intel's SGX [53], but SGX has a growing list of vulnerabilities [57].

One cryptographic primitive that can be used to facilitate PCD is *private set intersection* (PSI) [34]. In PSI, two parties compute the intersection of their inputs without leaking the other elements in their inputs. In the PCD setting, the sizes of these two sets are very different, with the service's user database dwarfing the user's address book by many orders of magnitude, motivating the need for imbalanced PSI designs [23, 32, 39, 41]. To further eliminate the need for additional infrastructure, the "key" that identifies a friend in PCD is often a phone number or email address. However, this is not a private piece of information. As a result, in PSI-based designs, a malicious client can add a victim's phone number to their address book to query for the victim's presence on different services.

A variation of PSI is *authorized PSI*, where the client proves that it possesses authorization for each element in their set [20, 21, 30]. We compare our approach to these works in Section 8; the key distinction is that these works require communication linear in the size of the larger set, whereas our approach is sublinear, and these works do not have a notion of deletions or updates. Finally, to our knowledge, we are the first to highlight the need of restricting queries to PCDs to avoid leaking participation in online services, and to provide a concrete design and implementation (the above works target other applications).

5.2. Vision and model

Imagine many online services, each of which offers users a PCD so that they may privately find out if their friends are using the service. What we want is the following: each user will share with their friends a token that allows them to discover them on the different services. If Alice tries to look up Bob in a PCD but Alice has not been authorized by Bob to do so, she will learn nothing about Bob's existence in any of the services. Conversely, if she has been authorized and uses the token provided by Bob, she can learn whether Bob is part of any of the many online services. We call this an *authenticated* PCD protocol.

Authenticated PCD. Each service takes on the role of the database server in Pirmission, while each user can simultaneously act as a data producer and a Pirmission client. To simplify explanations, we will call two specific users Alice and Bob. Alice will model a new user to a service, who wishes to determine whether or not another user, Bob, is also using the service. Alice will query the service, and depending on a number of factors, will receive either a negative response, or an identifier which she can use to communicate with Bob within the service. We describe the properties we require in the next section.

More formally, an authenticated PCD protocol consists of a tuple of algorithms (*Initialize*, *Insert*, *Update*, *Delete*, *Query*, *Respond*, *Extract*). The first four, described below, are executed by the server.

- $\mathrm{Initialize}() \rightarrow (D, pub)$, a randomized algorithm that creates the database D and publishes public information.

In our specific construction, the public information is k hash functions, $h_1 \dots h_k$.

- $\text{Insert}(D, uid, d_{uid}, t_{uid}) \rightarrow D'$, a deterministic algorithm that inserts a user (uid, d_{uid}, t_{uid}) into D.
- $\text{Update}(D, uid, d_{uid}, t_{uid}) \rightarrow D'$, a deterministic algorithm that updates $data$ and t_{uid} for uid.
- $\text{Delete}(D, uid) \rightarrow D'$, a deterministic algorithm which deletes a user from the database.

The remaining algorithms are similar to Pirmission's:

- $\text{Query}(uid, t, pub) \rightarrow Q$, a randomized algorithm executed by the client that creates a query for user uid.
- $\text{Respond}(D, Q) \rightarrow e$, a randomized algorithm executed by the server which takes the query provided by the client and outputs an encrypted response e.
- $\text{Extract}(e) \rightarrow (uid, d_{uid}) \mid \perp$, a deterministic algorithm executed by the client that returns the data associated with the queried user if the user is present in the database and the provided access token was correct, or $\perp$ otherwise.

Threat model. Similarly to Pirmission, we allow both users and services to behave maliciously. Our design does not prevent services from denying service or returning malformed or false responses, but it does prevent services from learning information about the users' friends through their queries.

5.3. Properties

Given a service S, and two users, Alice and Bob, Alice queries S for the presence of Bob.

1) S cannot identify Bob. Depending on the application, after the protocol finishes, Alice may need to tell the application about Bob. In other contexts (e.g., metadata-private messaging [10, 67]) the users can interact without S learning that they are friends. Formally, for any PPT adversary $\mathcal{A}$, security parameter λ, for any token t, and any user IDs uid, uid' ($uid \neq uid'$):

$$\Pr[\mathcal{A}(\text{Query}(uid, t)) = 1] - \Pr[\mathcal{A}(\text{Query}(uid', t)) = 1] \leq \text{negl}(\lambda)$$

2) S cannot tell if Alice is friends with Bob. For any PPT adversary $\mathcal{A}$, security parameter λ, and for any $t \neq t'$:

$$\Pr[\mathcal{A}(\text{Query}(uid, , t)) = 1] - \Pr[\mathcal{A}(\text{Query}(uid, t')) = 1] \leq \text{negl}(\lambda)$$

3) If Alice is friends with Bob, and Bob uses S, then Alice learns that Bob uses S and his data (e.g., Bob's public key for the service). If $t = t_{uid}$ and $uid \in D$:

$$\text{Extract}(\text{Respond}(\text{Query}(uid, t, pub))) = (uid, d_{uid})$$

4) Alice cannot discover whether Bob uses S unless she has permission to query him. This is analogous to the access control guarantee of Pirmission.
5) If Bob terminates his account on S and then becomes friends with Alice, Alice will not be able to learn that Bob used S before they were friends. This is analogous to the forward secrecy property in Pirmission. In this setting, if Bob terminates his account on S, then:

$$\Pr[\text{Alice guesses Bob used } S | \text{Alice has Bob's token}] - \Pr[\text{Alice guesses Bob used } S] \leq \text{negl}(\lambda)$$

5.4. Construction

At a high level, we first observe that standard PIR is not a good fit for PCD. The reason is that users have a *uid* (phone number, email, etc.) for their friend, but not an index into some array. Instead, we use *PIR by keywords* [25] which allows one to query with an arbitrary identifier rather than an index; we use a construction based on cuckoo hashing [8].

Initialize and insert. The service S maintains k tables of users, corresponding to k hash functions $h_1, h_2, \dots h_k$. S also maintains k tables of tokens as per Pirmission. For each user of the service, S hashes their uid with its first hash function and stores a tuple of the identifier and associated data in the corresponding location of the first table. When a collision occurs, the server will replace the original item in that table with the new item, and move the original item to the location in the next table, pointed to by the value of the next hash function.

Update and delete. For updates, S finds to which table and position the user's identifier maps, and updates its associated data and/or token. Deletions are similar except that the data and token are removed.

Query, respond, and extract. To query for a user with identifier uid, the client makes k simultaneous queries, one for each $h_i(uid)$. Each of these queries is a Pirmission query as described in Section 4, although the same encrypted token is used for all queries. Note that the querier does not know a priori in which hash table uid is stored; however, if uid is in the database, one of the queries should return their information. Further, because the querier submits a single encrypted token, the remaining queries reveal no information about other users due to Pirmission's access control guarantee.

In Appendix C we show the security of this construction and discuss a small leakage caused by hash collisions. One might be able to avoid even this small leakage by leveraging a recent single-round PIR by keywords proposal [52].

6. Token distribution

Pirmission requires a sub-token for each coefficient in the BFV polynomial (§4.3), but the client can simply provide the client and the service a *seed* to a PRG; both clients and the service can generate all the sub-tokens from this seed. One thing to keep in mind is that if a service has a seed from a user and the user reuses the seed in other services, then a malicious server can look up the user in those other services. This can be addressed by deriving a service-specific seed from a master seed and only giving each service the service-specific seed. Then, the producer

can give the master seed to each client. Clients can then derive the service-specific seed, use the PRG to obtain the token, and send the encryption of the token to the service.

Another possibility is for a producer to use a *ciphertext-policy attribute-based encryption* scheme [11] to encrypt one or more seeds under an appropriate policy and publish these ciphertexts on some Web site or bulletin board. The producer can then give each client a suitable decryption key. After this, the producer can update the ciphertexts without needing to send a message to the data client. Note that the publication of a ciphertext, and the number of published ciphertexts, could leak information about how many services the data producer is using.

7. Implementation

Our implementation of Pirmission builds on SealPIR [1] but we note that Pirmission's techniques apply broadly: they can used with XPIR [4], FastPIR [5], MulPIR [8], OnionPIR [55], and Spiral [54] to enhance those protocols with access control. We update SealPIR to support the latest version of the SEAL homomorphic library (v4.0.0) [2] and to support symmetric encryption as recommended by Ali et al. [8] and recursive modulus switching [28]. We also pack small records into a single BFV plaintext using SEAL's BatchEncoder for BFV, which performs the CRT batching operation described in Section 4.3. We use this updated and slightly faster SealPIR as our baseline. Adding access control to SealPIR required fewer than 200 lines of C++ code.

Authenticated PCD. We extend the server in Pirmission to embed a cuckoo hash table with three hash functions (similarly to PIR-PSI [32]) inside its database. We use Pearson hashing [60] to hash each phone number. In addition to 10-digit phone numbers, for each user, the database also stores either a 64-byte or 128-byte string of additional information. This additional information could contain a public key or additional profile information. Including a single byte used for status information, our element size is either 75 or 139 bytes. Our PCD implementation requires 500 lines of code on top of Pirmission.

8. Evaluation

In this section we evaluate the performance of Pirmission and authenticated PCD, and compare them to other works in the literature. We answer three main questions:

- What are the costs of adding access control to PIR?
- How do the guarantees and asymptotic costs of Pirmission compare to related works?
- What is the performance of our authenticated PCD implementation, and how does it compare to other works in terms of costs and guarantees?

We run all of our experiments on an Azure F16s v2 instance (2.60 GHz Intel Xeon Platinum 8272CL CPU and 64 GB) with Ubuntu 20.04. All of our results are on a *single core*. PIR is an embarrassingly parallel workload so performance improves linearly with the number of cores.

Parameters. We instantiate SealPIR and Pirmission with BFV polynomial degree 4096 (this is the smallest degree for which SEAL 4.0 supports homomorphic rotations). Note that SealPIR also uses rotations to compress queries, so this is not a limitation specific to Pirmission. We configure SealPIR with a 20-bit plaintext modulus and Pirmission with a 15-bit plaintext modulus; Pirmission requires the smaller modulus to ensure that the noise of the extra operations it performs does not lead to an incorrect result (see Footnote 1 in Section 4.4). Since 15-bits is not enough to prevent brute force guessing of sub-tokens (§4.4), we boost security to 60-bits with $s = 4$ masks; a malicious client without the token needs to issue over a quintillion PIR queries to learn the database record. To achieve sublinear communication both Pirmission and SealPIR represent the database as a 2 dimensional matrix (§4.7), and use SealPIR's technique [9] to compress entire vectors. A PIR query is therefore 2 ciphertexts, one for each dimension.

8.1. The cost of access control

We study the overhead of Pirmission over SealPIR when the database records are small (§4.5) and large (§4.6).

Small records. We let the database consist of small 256 byte records. Small records are common in applications like messengers that use PIR to hide metadata [5, 10]. Both systems pack records into as few BFV plaintexts as possible. Figure 4 tabulates the results.

The time that the server takes to generate a reply with Pirmission is around 2.8–3× higher than the time needed by SealPIR. This overhead stems from a few sources: (1) Pirmission queries two tables, the data table and the token table; (2) Pirmission uses a smaller plaintext modulus so it can pack fewer 256-byte records into plaintexts (i.e., the data table is 25% larger in Pirmission than in SealPIR); and (3) Pirmission also performs 1 HomSub and 4 HomAdd, HomRot, and PCMult to compute the 4 masks.

The query size of Pirmission consists of 3 ciphertexts (2 for the PIR query and 1 for the encrypted token), whereas SealPIR's is only 2 (no token). The response size of Pirmission is about 50% larger than the response size of SealPIR. This stems from the lower plaintext modulus in Pirmission: when Pirmission performs the technique in Section 4.7, it needs more plaintexts to represent the ciphertexts (interpreted as binary blobs) that are produced as a result of the dot product between D and Q_{col}. Another way to say this is that the cryptosystem's expansion factor F—which measures the ratio between a ciphertext and the largest plaintext that can be encrypted in one ciphertext—is larger when the plaintext modulus is smaller. This factor impacts the response, which consists of F ciphertexts.

Large records. In this experiment with let the database have $n = 2^{14}$ large records, each of size $k \cdot 7.2$ KB, where k is a parameter that we vary. These records are too large to fit in a single BFV plaintext and hence both Pirmission and

	SealPIR ($n = 2^{14}$)	**SealPIR** ($n = 2^{16}$)	**SealPIR** ($n = 2^{18}$)	**Pirmission** ($n = 2^{14}$)	**Pirmission** ($n = 2^{16}$)	**Pirmission** ($n = 2^{18}$)
Query Generation (ms)	1	1	1	2	2	2
Reply Generation (ms)	51	115	292	151	345	835
Reply Decode (ms)	1	1	1	1	1	1
Query Size (KB)	93	93	93	139	139	139
Response Size (KB)	185	185	185	278	278	278

Figure 4: Comparison of SealPIR and Pirmission when processing on a database with small records that are each 256 bytes.

	SealPIR ($k = 1$)	**SealPIR** ($k = 3$)	**SealPIR** ($k = 6$)	**Pirmission** ($k = 1$)	**Pirmission** ($k = 3$)	**Pirmission** ($k = 6$)
Query Generation (ms)	1	1	1	2	2	2
Reply Generation (ms)	1,006	2,473	4,670	1,748	3,404	5,760
Reply Decode (ms)	1	3	7	1	5	10
Query Size (KB)	93	93	93	139	139	139
Response Size (KB)	185	555	1,110	278	1,112	1,668

Figure 5: Comparison of SealPIR to Pirmission, with $n = 2^{14}$ elements each of size $k \cdot 7.2$ KB. As k increases, elements become larger and the relative overhead of Pirmission over SealPIR decreases.

SealPIR must use the technique described in Section 4.6. We give the results in Figure 5.

Unlike the small records experiment, the relative overhead of Pirmission over SealPIR is only 23% when records are large ($k = 6$). This is because SealPIR performs the dot product between the query and k databases, whereas Pirmission does so for $k+1$ databases. Indeed, Pirmission's overhead is highest for the $k = 1$ case. In both systems the query size remains the same since the same query can be reused for all k databases, and the response increases by a factor of k since the client receives a response from each of those k databases.

8.2. Comparison with other works

We compare Pirmission to two other bodies of work that combine access control with private queries. We summarize the difference in Figure 6.

At a high level, the works based on OT provide access control but they have no notion of database updates or deletions in their models and have linear communication. If a producer wishes to update an element, the client needs to fetch a new copy of the entire encrypted database (expensive!) and the client learns that there is an update—which is weaker than the guarantee provided by Pirmission.

On the other hand, accredited SPIR [44] supports deletions (but not updates) and has lower communication complexity than Pirmission. However, the concrete costs are significantly higher—to the point that accredited SPIR is not usable in practice. Specifically, accredited SPIR relies on a PIR scheme due to Lipmaa [49] that has no known implementation and has many practical drawbacks. First, Lipmaa states that the communication costs of his protocol are worse than the scheme we give in Figure 1 until N grows past at least 2^{40}. Second, Lipmaa's PIR scheme requires a length-flexible additively homomorphic scheme, and Lipmaa uses Damgård-Jurik [29], a generalization of Paillier [58]. The authors of XPIR [4] found that lattice-based cryptosystems like BFV (the one SealPIR relies on) are orders of magnitude more efficient than Paillier. More recently, Ali et al. [8] confirmed that schemes based on Damgård-Jurik are significantly more expensive than modern PIR schemes like SealPIR—to the extent that the largest database that they evaluate has only $2,000$ elements.

8.3. Authenticated Private Contact Discovery

Finally, we evaluate our proposed authenticated PCD design. To our knowledge, there are no previous PCD schemes that require users to prove friendship, so we evaluate as follows. First, we position our design in the context of PCD designs in general, and authorized PSI designs in particular. Second, we provide our empirical results.

The majority of PCD schemes use PSI, not PIR. This has an immediate consequence in that in PSI, the client's input is a set of elements, whereas in PIR, it is a single index, or in the case of batch PIR [9, 10, 38] a fixed number of indexes. However, in the context of PCD, the sizes of the two sets are generally very different, as the server's user database often dwarfs the size of the client's contact list. Therefore, standard PSI schemes, which include communication linear in the size of the server's set, are undesirable. Unbalanced PSI [23, 24], which targets this use case, aims for sublinear communication in the size of the server's set. Unbalanced

Work	Removing Elements	Updating Elements	Communication Complexity	Computation Complexity
OT-AC [17–19, 68]	No	No	$O(n)$	$O(n)$
Accredited SPIR [49]	Yes	No	$O(\log^2(n))^\star$	$O(n)$
Pirmission (this work)	Yes	Yes	$O(n^{1/d})$	$O(n)$

Figure 6: Comparison of Pirmission to related schemes. d is the dimension of the database's hypercube representation (§4.1); for good concrete costs d is usually 2 or 3. *Concrete costs are orders of magnitude higher than Pirmission's (see text).

Work	Additional Data	Forward Secrecy	Communication Complexity
IBE-PPIT [30]	Yes	No	$O(n)$
Schnorr-PPIT [30]	Yes	No	$O(m \cdot n)$
Certified Sets [21]	No	Yes	$O(m \cdot n)$
RSA-PPIT [30]	Yes	Yes	$O(m \cdot n)$
PPSIP [31]	Yes	Yes	$O(m + n)$
This work	Yes	Yes	$O(m \cdot n^{1/d})$

Figure 7: Comparison of our authenticated PCD scheme with authorized PSI and similar schemes. m is the size of the user's contact list and n is the size of the server's database.

PSI designs generally involve the client making a separate query for each item in its set, yielding a multiplicative cost factor in the size of the client's set. We can do the same thing with our design by having the client make a query for each contact. Because we are using PIR, we also get sublinear communication in the size of the server's user database.

Note that to our knowledge, there are no unbalanced PSI constructions which have a notion access control. Consequently, we compare against existing PSIs that do have access control which are not unbalanced [21, 30, 31]. We show the properties of these designs in Figure 7. The most significant difference here is that these designs were not created with PCD in mind; as a result, they all require communication linear in the server's database. In contrast, our authenticated PCD proposal achieves sublinear communication. We also wanted to evaluate the concrete costs of existing PSI with access control designs, but unfortunately, none of them have implementations.

Concrete costs of authenticated PCD. Finally, we test our own PCD implementation. We calculate the server's cost to look up a single contact in a database with a varying number of users. Database entries include the user's identifier in addition to 64 or 128 bytes of client information (e.g., a public key, a URL). We show the setup and response generation times in Figure 8. We do not show the client's query generation times, as it is essentially 10 ms throughout. As expected, the cost of a contact lookup is essentially that of making three consecutive Pirmission queries with an equivalent-sized database.

9. Discussion

Pirmission extends PIR with a notion of access control. Our motivation is that real databases often need to enforce restrictions on which clients can access certain elements, but existing PIR protocols have no mechanism for such enforcement. One of the lessons we learned in designing Pirmission is that the privacy requirements are often subtle (e.g., forward secrecy), and care must be taken to ensure no information leaks. We see this subtlety also reflected in another property that Pirmission does not provide: integrity.

Integrity. In settings where a data producer owns the database and outsources it to some malicious server, integrity does not hold: the server is free to manipulate the actual contents of the database. In standard PIR, the producer could construct a Merkle tree (or a vector commitment) of the database, publish the root, and store Merkle proofs alongside each element (i.e., the element at index i consist of the data and a Merkle proof to the root from leaf i). Then, when a client fetches an element with PIR, it obtains the data and a proof to check its authenticity. However, the above approach does not work if we require our strong access control property. If a client has access to every element in the database except one, then by monitoring root changes it can determine when that one element changes. Maintaining integrity without leaking any information (not even updates) is an interesting open question.

Preprocessing.. There are several recent PIR schemes [27, 37, 51, 59, 63] that perform an offline preprocessing phase whereby clients download *hints* that they can then use later to get quicker responses from the server. It is not clear how to extend these schemes with access control because the hints that clients download already encode information about database elements. Hiding the fact that an element has changed or providing forward secrecy without having to change the key every time an element is updated seems hard to do in this regime. It might be that a weaker notion of access control could be provided for these protocols, and this could be an interesting avenue of future work.

Acknowledgments

We thank Ryan Henry for a helpful discussion related to our notion of forward secrecy that led to an improved definition. We also thank Wei Dai and Yiping Ma for helping us better understand SEAL's key switching mechanism in

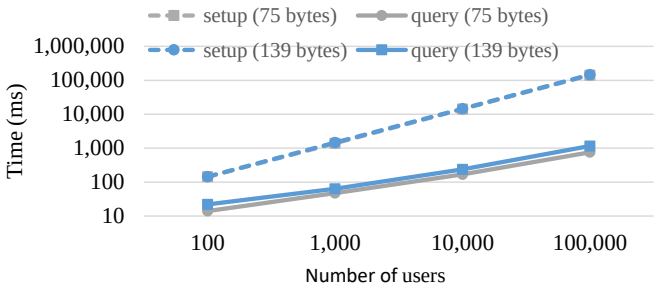

Figure 8: Costs for authenticated PCD with 64 bytes of extra information (75 bytes total, gray lines) and 128 bytes of extra information (139 bytes total, blue lines). Note that setup time includes insertion, scales linearly with users, and is independent of the size of the extra information (the lines are identical). The response time associated with 128 byte extra information is about 2× that of 64 bytes.

the presence of maliciously generated Galois keys. Finally, we thank Srinath Setty and Weidong Cui for helpful conversations that improved the presentation of our work.

References

[1] SealPIR: A computational pir library that achieves low communication costs and high performance. https://github.com/microsoft/sealpir, 2020.

[2] Simple encrypted arithmetic library — SEAL. https://github.com/microsoft/SEAL, 2022.

[3] M. Abe, J. Camenisch, M. Dubovitskaya, and R. Nishimaki. Universally composable adaptive oblivious transfer (with access control) from standard assumptions. In *Proceedingsinproceedings of the ACM workshop on Digital identity management*, Nov 2013.

[4] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIR: Private information retrieval for everyone. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[5] I. Ahmad, Y. Yang, D. Agrawal, A. E. Abbadi, and T. Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 313–329. USENIX Association, July 2021.

[6] W. Aiello, Y. Ishai, and O. Reingold. Priced oblivious transfer: How to sell digital goods. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 01 2001.

[7] N. Alexopoulos, A. Kiayias, R. Talviste, and T. Zacharias. MCMix: Anonymous messaging via secure multiparty computation. In *Proceedings of the USENIX Security Symposium*, Aug. 2017.

[8] A. Ali, T. Lepoint, S. Patel, M. Raykova, P. Schoppmann, K. Seth, and K. Yeo. Communication–computation trade-offs in PIR. In *Proceedings of the USENIX Security Symposium*, 2021.

[9] S. Angel, H. Chen, K. Laine, and S. Setty. PIR with compressed queries and amortized query processing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2018.

[10] S. Angel and S. Setty. Unobservable communication over fully untrusted infrastructure. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2016.

[11] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, June 2007.

[12] N. Borisov, G. Danezis, and I. Goldberg. DP5: A private presence service. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, June 2015.

[13] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2012.

[14] Z. Brakerski, C. Gentry, and S. Halevi. Packed ciphertexts in LWE-based homomorphic encryption. In *Proceedings of the International Conference on Practice and Theory in Public Key Cryptography (PKC)*, 2013.

[15] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the Innovations in Theoretical Computer Science (ITCS) Conference*, Jan. 2012.

[16] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from Ring-LWE and security for key dependent messages. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2011.

[17] J. Camenisch, M. Dubovitskaya, R. R. Enderlein, and G. Neven. Oblivious transfer with hidden access control from attribute-based encryption. In I. Visconti and R. De Prisco, editors, *Security and Cryptography for Networks*, pages 559–579, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[18] J. Camenisch, M. Dubovitskaya, and G. Neven. Oblivious transfer with access control. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2009.

[19] J. Camenisch, M. Dubovitskaya, G. Neven, and G. M. Zaverucha. Oblivious transfer with hidden access control policies. In *Proceedings of the International Conference on Practice and Theory in Public Key Cryptography (PKC)*, 2011.

[20] J. Camenisch, M. Kohlweiss, A. Rial, and C. Sheedy. Blind and anonymous identity-based encryption and authorised private searches on public key encrypted data. In *Proceedings of the International Conference on Practice and Theory in Public Key Cryptography (PKC)*, 2009.

[21] J. Camenisch and G. M. Zaverucha. Private intersection of certified sets. In *Proceedings of the International Financial Cryptography Conference*, 2009.

[22] H. Chen, K. Han, Z. Huang, A. Jalali, and K. Laine. Simple encrypted arithmetic library v2.3.0-4. https://sealcrypto.org, Dec. 2017.

[23] H. Chen, Z. Huang, K. Laine, and P. Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2018.

[24] H. Chen, K. Laine, and P. Rindal. Fast private set intersection from homomorphic encryption. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2017.

[25] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. Cryptology ePrint Archive, Report 1998/003, Feb. 1998. http://eprint.iacr.org/1998/003.

[26] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1995.

[27] H. Corrigan-Gibbs and D. Kogan. Private information retrieval with sublinear online time. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 2020.

[28] A. Costache, K. Laine, and R. Player. Evaluating the effectiveness of heuristic worst-case noise analysis in fhe. Cryptology ePrint Archive, Report 2019/493, 2019. https://ia.cr/2019/493.

[29] I. Damgård, M. Jurik, and J. Nielsen. A generalization of paillier's public-key system with applications to electronic voting. *International Journal of Information Security*, 9:371–385, 04 2003.

[30] E. De Cristofaro, S. Jarecki, J. Kim, and G. Tsudik. Privacy-preserving policy-based information transfer. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, June 2009.

[31] E. De Cristofaro and G. Tsudik. Practical private set intersection protocols with linear complexity. In *Proceedings of the International Financial Cryptography and Data Security Conference*, Jan. 2010.

[32] D. Demmler, P. Rindal, M. Rosulek, and N. Trieu. PIR-PSI: Scaling private contact discovery. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2018.

[33] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, Mar. 2012. https://eprint.iacr.org/2012/144.pdf.

[34] M. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, June 2004.

[35] C. Gentry, S. Halevi, and N. P. Smart. Fully homomorphic encryption with polylog overhead. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, Apr. 2012.

[36] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin. Protecting data privacy in private information retrieval schemes. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, page 151–160, New York, NY, USA, 1998. Association for Computing Machinery.

[37] A. Henzinger, M. M. Hong, H. Corrigan-Gibbs, S. Meiklejohn, and V. Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. Cryptology ePrint Archive, Paper 2022/949, 2022. https://eprint.iacr.org/2022/949.

[38] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Batch codes and their applications. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, June 2004.

[39] D. Kales, C. Rechberger, T. Schneider, M. Senker, and C. Weinert. Mobile private contact discovery at scale. In *Proceedings of the USENIX Security Symposium*, USA, 2019.

[40] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.*, 39(4):1543–1561, dec 2009.

[41] A. Kiss, J. Liu, T. Schneider, N. Asokan, and B. Pinkas. Private set intersection for unequal set sizes with mobile applications. *Proceedings on Privacy Enhancing Technologies*, 2017, 10 2017.

[42] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1997.

[43] A. Kwon, D. Lu, and S. Devadas. XRD: Scalable messaging system with cryptographic privacy. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2029.

[44] M. Layouni. Accredited symmetrically private information retrieval. In *Advances in Information and Computer Security*, 2007.

[45] M. Layouni, M. Yoshida, and S. Okamura. Efficient multi-authorizer accredited symmetrically private information retrieval. In *Information and Communications Security*, 2008.

[46] D. Lazar, Y. Gilad, and N. Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2018.

[47] D. Lazar, Y. Gilad, and N. Zeldovich. Yodel: Strong metadata security for voice calls. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[48] B. Libert, S. Ling, F. Mouhartem, K. Nguyen, and H. Wang. Adaptive oblivious transfer with access control from lattice assumptions. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2017.

[49] H. Lipmaa. An oblivious transfer protocol with log-squared communication. In *Information Security*, 2005.

[50] V. Lyubashevsky, C. Peikert, , and O. Regev. On ideal lattices and learning with errors over rings. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 2010.

[51] Y. Ma, K. Zhong, T. Rabin, and S. Angel. Incremental offline/online PIR. In *Proceedings of the USENIX Security Symposium*, Aug. 2022.

[52] R. A. Mahdavi and F. Kerschbaum. Constant-weight pir: Single-round keyword pir via constant-weight equality operators. In *Proceedings of the USENIX Security Symposium*, 2022.

[53] M. Marlinspike. Technology preview: Private contact discovery for Signal. https://signal.org/blog/private-contact-discovery/, Sept. 2017.

[54] S. J. Menon and D. J. Wu. Spiral: Fast, high-rate single-server PIR via FHE composition. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2022.

[55] M. H. Mughees, H. Chen, and L. Ren. Onionpir: Response efficient single-server pir. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2021.

[56] M. Naor and B. Pinkas. Oblivious transfer and polynomial evaluation. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1999.

[57] A. Nilsson, P. N. Bideh, and J. Brorsson. A survey of published attacks on intel sgx, 2020.

[58] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 1999.

[59] S. Patel, G. Persiano, and K. Yeo. Private stateful information retrieval. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[60] P. K. Pearson. Fast hashing of variable-length text strings. *Communications of the ACM*, 33(6):677–680, June 1990.

[61] M. O. Rabin. How to exchange secrets with oblivious transfer. Cryptology ePrint Archive, Report 2005/187, 2005. https://ia.cr/2005/187.

[62] A. Sahai and B. Waters. Fuzzy identity-based encryption. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2005.

[63] E. Shi, W. Aqeel, B. Chandrasekaran, and B. Maggs. Puncturable pseudorandom sets and private information retrieval with near-optimal online bandwidth and time. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2021.

[64] N. P. Smart and F. Vercauteren. Fully homomorphic SIMD operations. *Designs, codes, and cryptography*, 71(1), 2014.

[65] J. P. Stern. A new and efficient all-or-nothing disclosure of secrets protocol. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, Oct. 1998.

[66] A. Vadapalli, K. Storrier, and R. Henry. Sabre: Sender-anonymous messaging with fast audits. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2022.

[67] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2015.

[68] Y. Zhang, M. H. Au, D. S. Wong, Q. Huang, N. Mamoulis, D. W. Cheung, and S.-M. Yiu. Oblivious transfer with access control: Realizing disjunction without duplication. In *Proceedings of the International Conference on Pairing-Based Cryptography*, 2010.

Appendix A. Effect of malicious Galois keys

In Section 4.4 we discuss how Pirmission uses rotations to cause each entry in the plaintext $T_i - t$, where T_i is the token at index i and t is the client's guess (both of which are 2-by-$N/2$ matrices), to contribute to different entries in each of the masks of D_i $(d_1, \ldots, d_s)$. One might wonder whether specifying a bad Galois key can lead to the following behavior (please see Figure 3 for the accompanying pseudocode).

Assume we have two operations $Rot(c, j)$ and $KeySwitch(c, gk)$, such that HOMROT is Rot followed by $KeySwitch$, and gk is the Galois key provided by the client. As a simplified example, let us assume that we have $N = 4$ and $p = 11$. Note that this parameter choice is technically invalid since $p \not\equiv 1 \pmod{2N}$ but the smallest valid parameter setting requires matrices with 8 entries which is far too verbose and it obscures the point we want

to highlight. Let:

$$T_i = \begin{bmatrix} 1 & 3 \\ 5 & 8 \end{bmatrix}$$
$$t = \begin{bmatrix} 6 & 7 \\ 3 & 4 \end{bmatrix}$$
$$d_1 = \mathrm{Enc}_{pk}(T_i - t) = \mathrm{Enc}_{pk}\left(\begin{bmatrix} 6 & 7 \\ 2 & 4 \end{bmatrix}\right)$$
$$tmp = Rot(d_1, 1) = \mathrm{Enc}_{pk'}\left(\begin{bmatrix} 7 & 6 \\ 4 & 2 \end{bmatrix}\right)$$
$$d_2 = KeySwitch(tmp, gk)$$

The question is whether a bad gk can cause d_2 to encrypt

$$\begin{bmatrix} v_1 & v_2 \\ v_3 & v_4 \end{bmatrix}$$

where at least one of $v_1, \ldots, v_4$ is chosen by the client (i.e., the client has control and knows what this value is). Notice that if the client could do that, then a trivial attack is to set one (or all) of these values to 0, so those entries will contribute nothing as a mask during Line 11 in Figure 3.

This is not possible. The reason is as follows.

Assume that d_1 requires secret key sk to be decrypted (sk is generated by the client during KeyGen, see Section 3.1). Assume tmp requires secret key sk' for decryption. Let us look deeper at the structure of tmp: it actually consists of two polynomials $(tmp[0], tmp[1])$ since ciphertexts in BFV [13, 33] are elements in a different polynomial ring than the plaintexts and usually consist of two or more polynomials per ciphertext. Galois keys are symmetric encryptions of sk' (to be precise, the data of sk' in a different form) under sk. $KeySwitch$ is basically the homomorphic evaluation of decryption $tmp[0] + tmp[1] \cdot sk'$ under sk. Since the client provides the Galois key, we can define sk' to be whatever the Galois key gk is decrypted to with secret key sk. A malicious client can only affect sk' since that is all of the information provided by gk: $tmp[0]$ and $tmp[1]$ are both computed by the server independently of gk (i.e., Rot does not require any client-supplied value). To get d_2 to encrypt a matrix with all/some zeros (or some other value v), the client needs to know what tmp encrypts (which is a permutation of $T_i - t$) so that it can put the information of $T_i - t$ into the Galois key to affect the result.

Another way to think about this is the following. If some bad Galois key provided by the client make d_2 become all zeros when

$$T_i - t = \begin{bmatrix} 6 & 7 \\ 2 & 4 \end{bmatrix}$$

then the same Galois key will not make d_2 become all zeros when

$$T_i - t = \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$$

Not only is trying to brute force the Galois key morally equivalent to brute forcing t in the first place, in existing PIR schemes the client specifies the Galois key once and this key is reused over many PIR queries to amortize the cost of transferring the Galois key.

Note that this does not mean that the client cannot provide a bad Galois key—it can. But doing so will just cause d_2 to have entries not under the control of the client, and whenever these entries are non-zero they will be multiplied by a random plaintext in Line 11 of Figure 3; the uniform random mask will still be applied.

Appendix B. Access control and sublinear communication

Pirmission can perform the access control and masking when the data table is expressed as a 2-dimensional matrix. In this case, Pirmission only needs to perform the masking *once* right after the matrix-multiplication between D and Q_{col} that produces E_j. To do so, Pirmission also represents the token table T as a matrix, and performs the matrix-multiplication between T and Q_{col}. The result is a vector P that contains encryptions of the elements in the j-th column of T. Then, the server homomorphically subtracts the client's encrypted token $\mathrm{Enc}_{pk}(t)$ from each of the entries in P. The server then samples some random BFV plaintext $r \in \mathbb{M}$, and multiplies it with the result of each subtraction. Notice that if the token is the correct token for the element in row i and column j, then $P'_i = \textsc{PCMult}(r, \textsc{HomSub}(P_i, \mathrm{Enc}_{pk}(t))) = \mathrm{Enc}_{pk}(0)$; it will be some random value otherwise. All other entries in P' will also contain the encryption of a random BFV plaintext (assuming the token provided by the client does not match any of the tokens in those other entries).

Finally, the server homomorphically adds P' and E_j entry-wise. At this point E_j will contain encryptions of random elements in all positions except for those for which the token was correct. When the server then performs the dot product between Q_{row} and E_j (as described in Section 4.7) it will either result in an encryption of $D_{i,j}$ or some uniformly random value in the plaintext space.

Generalization. The above protocol can be generalized to more than 2 dimensions by structuring the data and token tables as a d-dimensional hypercube and having the client issue d queries, each of size $n^{1/d}$. Note that after processing a query against each dimension the size of the response grows exponentially since every time the resulting vector of ciphertexts needs to be split into k chunks to be processed in the next dimension. Most prior works stick to $d = 2$. Regardless, the key point is that the masking is only performed once right after the *first* dimension: the masking either has no impact on the values (if the token is correct) or erases all signal so there is no need for further masking in the remaining dimensions.

Appendix C. Leakage of authenticated PCD

There are two sources of "leakage" in our proposed authenticated PCD design. However we think neither one

represents a significant threat to the privacy of the client or of the users in the database. The first happens in the following situation: suppose that Alice queries for her friend Bob, and finds the correct response in the first hash table. Some time later, Alice queries for Bob again, and the correct response has moved to the second hash table. Alice then learns that there is at least one user in the table whose identifier hashes to the same value as Bob's identifier under h_1. Either this user was just added or another user was added which led to the eventual displacement of Bob. Therefore, we can not argue indistinguishability of two user databases over time, because an adversary can generate situations in which this happens. The second source of leakage is similar: because we are not using a stash with cuckoo hashing, the probability of an insertion causing a cycle in the cuckoo hash table is not negligible. Whenever this happens, the table has to be rebuilt with new hash functions, and this is visible to all clients.

Here is a concrete attack that gives an adversarial client Alice information about whether or not a user Bob, who she is not friends with, is using the service. For each of the k hash functions, Alice creates three users of the form $U_{i,j}$ with $1 < i \leq 3$ and $1 < j \leq k$ such that $h_j(U_{i,j}\ id) = h_j(Bob\ id)$. If this does not trigger a rebuild of the database, then Alice knows that Bob was not using the service. Otherwise, Bob might have been using the service.

We can make the probability of a rebuild negligible by adding a stash s [40]. If an insertion would cause a cycle, then the offending element will be added to the stash. s_i will refer to the $i-th$ element of the stash and sT_i will refer to the corresponding token. A client query will then consist of k Pirmission queries and one encrypted token, $E_k(t')$. In addition to the k responses, the database server will also compute $(sT_i - E_k(t')) \cdot r_i + s_i$ for each element of the stash and random elements r_i. The database will return these $|s|$ values to the user, who can decrypt and check each one, but will only be able to read the element which matched the encrypted token. This addition of the stash does not affect the query size at all, adds $|s|$ additional ciphertexts to the response size and a corresponding amount of server computation. For our desired database sizes and security parameters, $|s|$ can be very small, only around 10 elements [40].

However, even with the stash, a client still learns when an insertion causes one of their friends to move tables or move to the stash. This can still leak information about users who the client is not friends with. Further, even if we require that all insertions be done before all queries, a similar form of leakage is present. As an example, consider a cuckoo hash table with two tables consisting of two slots each, $T_1 = 0, 1$ and $T_2 = 2, 3$. Call the two hash functions h_1 and h_2 and consider two users, A and B. Let $h_1(A) = h_1(B) = 0$, $h_2(A) = 2$ and $h_2(B) = 3$. Consider an alternative set of users A' and B' with $h_1(A') = 0$, $h_1(B') = 1$, $h_2(A') = 2$, and $h_2(B') = 3$. Even if we randomize the order in which we insert users and the order in which we apply the hash functions, the distributions of the final location of user A (A') will be distinguishable.

C.1. Security

Next, we justify the security properties of our system. Properties (1) and (2) concern the privacy of the client and properties (4) and (5) concern the security of the server, which Pirmission partially achieves.

Client privacy. This follows directly from the security of Pirmission. The only difference is in this case we have one encrypted token and three encrypted query vectors.

Server security. Here, we must be more careful. In our ideal server for Pirmission, we could assume that a client would not learn any unauthorized information as long as it queried an element to which it was granted access. As we discuss in Section C, this is not true when we introduce the cuckoo hash table, as a client learns additional information in the slot in which their friend is located. However, the adversary still can only ever learn information when they query for one of their friends. If we bound the number of friends to be $\ll n$, then in the worst case the adversary can only learn about users who hash to the same slots as the adversary's friends.

We note that a recent single-round PIR by keyword protocol [52] does not rely on Cuckoo hashing and can avoid this issue altogether. We plan to update our implementation with this new protocol to avoid all leakage.