Batch PIR and Labeled PSI with Oblivious Ciphertext Compression

Alexander Bienstock, *New York University;* Sarvar Patel and Joon Young Seo, *Google;* Kevin Yeo, *Google and Columbia University*

https://www.usenix.org/conference/usenixsecurity24/presentation/bienstock

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

Batch PIR and Labeled PSI with Oblivious Ciphertext Compression

Alexander Bienstock[*] *Sarvar Patel*[†] *Joon Young Seo*[‡] *Kevin Yeo*[§]

Abstract

In this paper, we study two problems: oblivious compression and decompression of ciphertexts. In oblivious compression, a server holds a set of ciphertexts with a subset of encryptions of zeroes whose positions are only known to the client. The goal is for the server to effectively compress the ciphertexts obliviously, while preserving the non-zero plaintexts and without learning the plaintext values. For oblivious decompression, the client, instead, succinctly encodes a sequence of plaintexts such that the server may decode encryptions of all plaintexts value, but the zeroes may be replaced with arbitrary values. We present solutions to both problems that construct lossless compressions only 5% more than the optimal minimum using only additive homomorphism. The crux of both algorithms involve embedding ciphertexts as random linear systems that are efficiently solvable.

Using our compression schemes, we obtain state-of-the-art schemes for batch private information retrieval (PIR) where a client wishes to privately retrieve multiple entries from a server-held database in one query. We show that our compression schemes may be used to reduce communication by up to 30% for batch PIR in both the single- and two-server settings.

Additionally, we study labeled private set intersection (PSI) in the unbalanced setting where one party's set is significantly smaller than the other party's set and each entry has associated data. By utilizing our novel compression algorithm, we present a protocol with 65-88% reduction in communication with comparable computation compared to prior works.

1 Introduction

Protecting user privacy is becoming a core problem in today's society with the continuing growth of cloud-based applications. There are many important cloud services that provide databases of essential information that need to be retrieved by users. In many cases, it is necessary to hide the queried database entry to preserve the user's privacy. This privacy requirement has appeared in many cloud services provided by large organizations including certificate transparency [3], contact discovery [10], device enrollment [7], password leak check [5, 6] and URL blocklists [8].

Private information retrieval (PIR) [24] is an important cryptographic protocol that enables an user to retrieve entries from a public database without revealing the identity of queried entries. PIR has been studied in both the single- and multi-server settings. The main difference is that multi-server PIR requires stronger assumptions of non-colluding servers. In our work, we will study both types of PIR protocols.

For many use cases, it is required that users retrieve a batch of multiple entries from the same public database. Some examples include anonymously retrieving encrypted recent messages from a communication system [15], privately fetching relevant advertisements according to user interests [38, 54] and checking validity of multiple certificates [42, 49].

To solve this problem, prior works have studied the notion of batch PIR where the user retrieves a set of t entries in a single query. The naive approach of executing t single-query PIR has the high computational overhead of $O(tn)$ as state-of-the-art single-query PIR schemes still require $O(n)$ server computation linear in the database size. Instead, current batch PIR solutions drastically decrease computation at the cost of increased communication. Angel *et al.* [14] presented a solution that reduce computation to $3n$ that required performing $1.5t$ independent PIR queries on smaller databases. Unfortunately, the number of requests and responses are 50% larger than the naive approach of t single-query PIR executions.

To reduce communication, prior works packed multiple multiple single-query PIR requests into a single ciphertext [13, 14] as well as encoding multiple PIR responses for small database entries into a single ciphertext using vectorization techniques [52]. However, this still requires explicitly encoding $0.5t$ "dummy" requests and responses. Furthermore, response techniques only apply for small entries where ci-

[*] New York University, abienstock@cs.nyu.edu. Part of work done while interning at Google.

[†] Google, sarvar@google.com.

[‡] Google, jyseo@google.com.

[§] Google and Columbia University, kwlyeo@google.com.

	Encoding Size	Encoding Time	Decoding Time
Choi *et al.* [23]	$O(t\lambda)$	$O(n\lambda)$	$O(t\lambda)$
Liu and Tromer [48]	$O(t\log^2 t\log\lambda)$	$O(nt)$	$O(t^3)$
Fleischhacker *et al.* [31]	$O(t)$	$O(n\log n)$	$O(t\sqrt{n})$
Fleischhacker *et al.* [31]	$O(t\lambda)$	$O(n\lambda)$	$O(t\lambda)$
Ours: LSObvCompress	$(1+\varepsilon)t$	$O(n\lambda)$	$O(t\lambda)$

Figure 1: Comparison of ciphertext compression for n ciphertexts with t non-zero values for failure probability at most $2^{-\lambda}$. Encoding size is measured in number of ciphertexts.

phertexts can pack multiple entries. In this work, we present compression techniques to avoid encoding non-essential values that are applicable regardless of the database entry size.

While PIR considers the setting of public database, the same problem also occurs for retrieving mutliple entries from private databases with sensitive data where users should not receive information irrelevant to them. This problem has been studied as labeled private set intersection (PSI) or batch symmetric PIR. We will use labeled PSI throughout the rest of our work. Examples use cases with a private database include discovering any contacts using a service [44] and checking all credentials of a user against a database of leaked credentials [13]. In our work, we will also study ways to reduce communication for labeled PSI using ciphertext compression.

1.1 Our Contributions

We identify two compression problems and present efficient schemes for both problems relying only on additive homomorphism of the underlying encryption scheme. This leads to improved batch PIR and labeled PSI constructions (when the encryption scheme used for these constructions is indeed somewhat homomorphic, as required in prior state-of-the-art work).

Oblivious Ciphertext Compression. We study the problem of *oblivious ciphertext compression* where a compressor is given n ciphertexts of which $t < n$ are non-zero. The compressor is unaware of the identity of the t non-zero ciphertexts. The decompressor has the private key for decryption and knows the location of the t non-zero ciphertexts. The goal is to enable the compressor to construct a succinct encoding that may be correctly decoded by the decompressor.

We present LSObvCompress with encodings of $(1+\varepsilon)t$ ciphertexts (where ε is a configurable parameter) while requiring only homomorphic addition of ciphertexts. In practice, we achieve ε to be as small as 0.05. Note, this is only 5% larger than the optimal compression rate that would consist of only t ciphertexts. Furthermore, oblivious compression requires only $O(n\lambda)$ homomorphic additions and decompression requires only $O(t\lambda)$ plaintext additions such that decompression is successful except with probability $2^{-\lambda}$. Our protocol, LSObvCompress, utilizes novel techniques to compress ciphertexts by encoding them as random linear systems that are efficiently solvable.

	Client Storage	Request Overhead	Response Overhead
Baseline	$O(1)$	1x	1x
Cuckoo Hashing [14]	$O(n)$	$\lceil 1.5/r\rceil$x	1.5x
Vectorized [52]	$O(n)$	$\lceil 1.5/r\rceil$x	$\lceil 1.5/d\rceil$x
Keyword [57]	$O(1)$	$\lceil 1.5/r\rceil$x	1.5x
Distributed Point Function (DPF)* [19]	$O(1)$	1.5x	1.5x
Ours: Single-Server	$O(1)$	$\lceil (1+\varepsilon)/r\rceil$x	1.5x
Ours: Single-Server	$O(1)$	$\lceil 1.5/r\rceil$x	$\lceil (1+\varepsilon)/d\rceil$x
Ours: Two-Server*	$O(1)$	1.5x	$(1+\varepsilon)$x

Figure 2: Keyword batch PIR comparisons for retrieving ℓ entries from n-entry database. Request and response overhead is compared to baseline of performing ℓ independent single-query PIR executions. We use r and d to denote the number of requests and plaintext database entries that can fit into a single ciphertext. Asterisks(*) denote two-server PIR protocols.

LSObvCompress significantly outperforms any prior compression schemes applicable to our setting. In particular, all prior schemes with efficient encoding and decoding produce encodings with $O(t\lambda)$ ciphertexts that is significantly larger than LSObvCompress in practice. Only one previous solution produced encodings with $O(t)$ ciphertexts [31], but the decoding is prohibitively expensive requiring computation of $O(t\sqrt{n})$ discrete logarithms. See Figure 1 for more comparisons. To be fair, we note that prior works study a more challenging version of this problem (see Section 2.1).

Oblivious Ciphertext Decompression. *Oblivious ciphertext decompression* switches the roles of compressor and decompressor. The compressor is given n plaintexts $\mathbf{p} = [p_1,\ldots,p_n]^T$, a subset $I \subset [n]$ of $t < n$ indices and a private encryption key. The goal is to produce a succinct encrypted compression of $\mathbf{p}$. The decompressor must be able to correctly retrieve the ciphertext vector $\tilde{\mathbf{c}} = [\tilde{c}_1,\ldots,\tilde{c}_n]^T$ such that $\tilde{c}_i$ must be an encryption of p_i for all $i \in I$. There are no requirements for any $i \notin I$. The decompressor must decompress obliviously without knowledge of the indices, $I \subset [n]$.

We present LSObvDecompress with nearly identical efficiency as LSObvCompress, with encodings of size $(1+\varepsilon)t$ ciphertexts. In practice, we get ε to be as small as 0.05 that is only 5% larger than optimal.

For decoding failure probability at most $2^{-\lambda}$, compression requires $O(t\lambda)$ plaintext additions while decompression requires $O(n\lambda)$ homomorphic additions. To our knowledge, no prior works are applicable to this specific problem.

Batch PIR. We apply our compression techniques to obtain state-of-the-art batch PIR schemes with reduced communication in both the single- and two-server settings. Our techniques work for both small and large entry databases. See Figure 2 for detailed comparisons with prior works.

The cuckoo hashing framework of Angel *et al.* [14] transforms any single-query PIR into a batch PIR protocol. To retrieve ℓ entries, the batch PIR performs 1.5ℓ single-query PIR executions. Recent work by Mughees and Ren [52] used vectorization techniques to pack multiple small entries into a single ciphertext. If d database entries fit into a ciphertext, vectorized batch PIR returns $\lceil 1.5\ell/d \rceil$ ciphertexts. However, this only works for small entries where $d \geq 2$.

In our work, we first present a batch PIR that can reduce response size regardless of database entry sizes. In the cuckoo hashing framework, at most ℓ PIR responses will be encryptions of relevant values while the remaining 0.5ℓ will be encryptions of zero. Using LSObvCompress, we reduce the response size from 1.5ℓ to 1.05ℓ PIR responses. To our knowledge, this is the first response reduction for batch PIR with large database entries. We also show that our techniques are compatible with the vectorization techniques of Mughees and Ren [52]. If d entries fit into a ciphertext, our techniques reduce the response size from $\lceil 1.5\ell/d \rceil$ ciphertexts to $\lceil 1.05\ell/d \rceil$ ciphertexts.

Similar ideas may also be used to reduce the request communication as well. Again, the client only cares about ℓ PIR requests and the remaining 0.5ℓ may be ignored. We leverage LSObvDecompress to reduce the total request size by only compressing values of the ℓ important requests and, essentially, ignoring the other 0.5ℓ requests. Combined with packing techniques [14] where r single-server PIR requests may fit into a ciphertext, we reduce request sizes from $\lceil 1.5/r \rceil$ to $\lceil 1.05/r \rceil$ ciphertexts. We stil apply apply vectorization [52] to obtained $\lceil 1.5/d \rceil$ response ciphertexts.

Finally, we show similar response reduction may also be obtained in two-server batch PIR protocols by applying LSObvCompress to prior constructions [19].

Labeled PSI. Next, we show that LSObvCompress and LSObvDecompress may also be used to construct improved schemes for labeled PSI. In particular, one can combine our above batch PIR construction, leveraging LSObvCompress and LSObvDecompress, with any oblivious PRF (OPRF) to obtain a labeled PSI protocol. Our labeled PSI schemes provides a 65-88% reduction in communication with comparable computation over prior solutions [21, 25].

2 Preliminaries

Linear Algebra. We denote $\mathbf{v}$ as column vectors and $\mathbf{v}^T$ as row vectors. We denote the i-th entry of $\mathbf{v}$ by $\mathbf{v}_i$. For two vectors n-length vectors $\mathbf{v}$ and $\mathbf{u}$, we denote the dot product operator as $\mathbf{v} \cdot \mathbf{u} = \sum_{i=1}^{n} \mathbf{v_i} \cdot \mathbf{u_i}$. We define a $n \times m$ matrix using its column vectors as $\mathbf{M} = [\mathbf{v}_1, \dots, \mathbf{v}_m]$ where the i-th column vector is $\mathbf{v}_i$ of length n. We may also define a matrix using its row vectors as $\mathbf{M} = [\mathbf{v}_1^T, \dots, \mathbf{v}_n^T]$ where $\mathbf{v}_i^T$ is the i-th row vector of length m. We denote the matrix-vector product $\mathbf{M} \cdot \mathbf{u} = [\mathbf{v}_1 \cdot \mathbf{u}, \dots, \mathbf{v}_n \cdot \mathbf{u}]$ where $\mathbf{u}$ is a m-length vector. We solve the linear system associated with $n \times m$ matrix $\mathbf{M}$ and n-length vector $\mathbf{u}$ by computing m-length vector $\mathbf{v}$ such that $\mathbf{M} \cdot \mathbf{v} = \mathbf{u}$.

For a vector $\mathbf{v}$ of length n and subset $I = \{i_1, \dots, i_k\} \subseteq [n]$, we denote by $\mathbf{v}_I = [\mathbf{v}_{i_1}, \dots, \mathbf{v}_{i_k}]$ containing the entries of $\mathbf{v}$ with indices in I. For $n \times m$ matrix $\mathbf{M} = [\mathbf{v}_1^T, \dots, \mathbf{v}_n^T]$ and subset $I = \{i_1, \dots, i_k\} \subseteq [n]$, we denote the sub-matrix consisting of row vectors with indices in I as $\mathbf{M}_{\mathsf{r}(I)} = [\mathbf{v}_{i_1}^T, \dots, \mathbf{v}_{i_k}^T]$. Similarly, for a $n \times m$ matrix $\mathbf{M} = [\mathbf{v}_1, \dots, \mathbf{v}_m]$ and subset $I = \{i_1, \dots, i_k\} \subseteq [m]$, we denote the sub-matrix consisting of column vectors with indices in I as $\mathbf{M}_{\mathsf{c}(I)} = [\mathbf{v}_{i_1}, \dots, \mathbf{v}_{i_k}]$.

Homomorphic Encryption. Throughout our work, we will define ciphertexts using $\tilde{c}$. A vector of ciphertexts will be defined as $\tilde{\mathbf{c}} = [\tilde{c}_1, \dots, \tilde{c}_n]$.

In our work, we will mainly consider lattice-based somewhat homomorphic encryption (SHE) where parameters are chosen to support a limited number of homomorphic operations, as used in prior state-of-the-art constructions of batch PIR and labeled PSI [21, 22, 25, 51, 53]. Our compression protocols only use additive hommorphism of these schemes, where noise grows additively. We refer to the full version for more details on SHE and recent PIR schemes using SHE [51, 53].

2.1 Oblivious Ciphertext Compression

We define the notion of an *oblivious ciphertext compression* scheme. For this primitive, we only assume additive homomorphism (ciphertext-ciphertext addition). The problem consists of two parties: a compressor and a decompressor. The compressor is given n ciphertexts, $\tilde{\mathbf{c}} = [\tilde{c}_1, ..., \tilde{c}_n]$, to be compressed. Both the compressor and the decompressor know the number of non-zero plaintext entries t. In addition, the decompressor has the private decryption key and the indices of the t non-zero entries, $I \subset [n]$. If $i \in I$, then $\tilde{c}_i$ is an encryption of a non-zero entry. The compressor's job is to produce a succinct encoding of the input ciphertexts with knowledge of only t. The encoding is consumed by the decompressor to recover the original t non-zero plaintext entries. We formally define oblivious ciphertext compression below.

Definition 1 (Oblivious Ciphertext Compression)**.** *Let* $\mathbf{p} = [p_1, ..., p_n] \in \mathbb{F}^n$ *be a vector of n plaintexts with at most t non-zero entries. Let* $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ *be an additive homomorphic encryption scheme, and let* $\tilde{\mathbf{c}} = [\tilde{c}_1, ..., \tilde{c}_n]$ *where* $\tilde{c}_i = \mathcal{E}.\mathsf{Enc}(\mathbf{pk}_{\mathcal{E}}, p_i)$ *for each* $i \in [n]$*. An oblivious ciphertext compression scheme consists of a pair of algorithms (*ObvCompress, Decompress*) satisfying:*

- $\hat{\mathbf{c}} \leftarrow \mathsf{ObvCompress}(\mathbf{pk}_{\mathcal{E}}, \tilde{\mathbf{c}}, t; R)$*: Oblivious compression takes in a public key* $\mathbf{pk}_{\mathcal{E}}$*, n ciphertexts* $\tilde{\mathbf{c}} = [\tilde{c}_1, ..., \tilde{c}_n]$*, the number of non-zero plaintext entries t, and randomness R. It outputs compressed ciphertexts* $\hat{\mathbf{c}}$*.*

- $\mathbf{p} \leftarrow \mathsf{Decompress}(\mathbf{sk}_{\mathcal{E}}, \hat{\mathbf{c}}, I; R)$*: Decompression takes in a secret key* $\mathbf{sk}_{\mathcal{E}}$*, compressed ciphertexts* $\hat{\mathbf{c}}$*, the non-zero* *plaintext entry indices $I \subset [n]$ ($|I| \leq t$) of p, and randomness R. It outputs the non-zero plaintext values $\{i, p_i\}_{i \in I}$.*

Let $\gamma = \gamma(\lambda)$ be the bit length of all n ciphertexts produced by the homomorphic encryption scheme $\mathcal{E}$. An oblivious ciphertext compression is δ-compressing if the bit length of $\hat{\mathbf{c}}$ is at most $\delta \cdot \gamma \cdot |\tilde{\mathbf{c}}|$. The failure probability is at most ε if, for each plaintext vector $\mathbf{p} = [p_1, ..., p_n]$ and associated ciphertexts $\tilde{\mathbf{c}} = [\tilde{c}_1, \ldots, \tilde{c}_n]$ with at most t non-zero values,

$$\Pr[\mathsf{Decompress}(\mathbf{sk}_{\mathcal{E}}, \hat{\mathbf{c}}, I) \neq \{i, p_i\}_{i \in I}] \leq \varepsilon$$

where $\hat{\mathbf{c}} \leftarrow \mathsf{ObvCompress}(\mathbf{pk}_{\mathcal{E}}, \tilde{\mathbf{c}}, t)$.

Comparison with Prior Work. Liu and Tromer [48] implicitly studied oblivious ciphertext compression, without explicitly defining the primitive. Fleischhacker *et al.* [31] considered another variant closer to our compression problem that was also implicitly studied in [48]. where the decompressor is not given the identity of the non-zero plaintext indices, $I \subset [n]$. Therefore, this is a harder setting than our compression problem. It is not surprising that the resulting compression rates or decoding efficiency are significantly worse than our constructions (see Figure 1). To our knowledge, our specific variant of compression has not been explicitly studied previously.

2.2 Oblivious Ciphertext Decompression

Next, we define *oblivious ciphertext decompression* that switches the compressor and decompressor roles. The compressor is given the plaintext vector, $\mathbf{p} = [p_1, \ldots, p_n]$ and a subset of t indices, $I \subset [n]$ with $|I| = t$ to produce a succinct encoding $\hat{\mathbf{c}}$. The decompressor is given $\hat{\mathbf{c}}$ and must produce the ciphertext vector $\tilde{\mathbf{c}} = [\tilde{c}_1, \ldots, \tilde{c}_n]^T$ such that each $\tilde{c}_i$ is an encryption of p_i for all $i \in I$. No correctness is required for $i \notin I$. In other words, $\tilde{c}_i$ needs to be an encryption of p_i only when $i \in I$. However, the decompressor must obliviously decode without any knowledge of the relevant indices, I. In fact, the compressed ciphertexts $\hat{\mathbf{c}}$ must not reveal any information about neither the underlying plaintext values $\mathbf{p} = [p_1, \ldots, p_n]^T$ nor the relevant indices I. To our knowledge, no prior works have studied this setting.

Definition 2 (Oblivious Ciphertext Decompression)**.** *Let $p = [p_1, ..., p_n]^T \in \mathbb{F}^n$ be a vector of n plaintexts and $I \subset [n]$ be a subset of $t < n$ indices. Let $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ be an additive homomorphic encryption scheme. A oblivious ciphertext decompression scheme consists of a pair of algorithms (*$\mathsf{Compress}, \mathsf{ObvDecompress}$*), where:*

- $\hat{\mathbf{c}} \leftarrow \mathsf{Compress}(\mathbf{sk}_{\mathcal{E}}, p, I; R)$*: The compression algorithm takes in a secret homomorphic encryption key $\mathbf{sk}_{\mathcal{E}}$, a vector of n plaintexts $p = [p_1, ..., p_n]^T$, a subset of t indices $I \subset [n]$ and randomness R. Then, it outputs the compressed ciphertexts $\hat{\mathbf{c}}$.*
- $\mathbf{p} \leftarrow \mathsf{ObvDecompress}(\mathbf{pk}_{\mathcal{E}}, \hat{\mathbf{c}}, n; R)$*: The decompression algorithm takes in a public homomorphic encryption key $\mathbf{pk}_{\mathcal{E}}$, compressed ciphertexts $\hat{\mathbf{c}}$, the number of total plaintexts n, and randomness R. Then, it outputs the ciphertext vector $\tilde{c} = [\tilde{c}_1, \ldots, \tilde{c}_n]^T$.*

Let $\gamma = \gamma(\lambda)$ be the bit length of all n ciphertexts produced by the homomorphic encryption scheme $\mathcal{E}$. A oblivious ciphertext decompression is δ-compressing if the bit length of $\hat{\mathbf{c}}$ is at most $\delta \cdot \gamma \cdot |\tilde{c}|$. The failure probability is at most ε if, for each plaintext vector $p = [p_1, ..., p_n]^T$ and subset $I \subset [n]$ of size t, the following holds:

$$\Pr[\exists i \in I \mid \mathsf{Dec}(\mathbf{sk}_{\mathcal{E}}, \tilde{c}_i) \neq p_i] \leq \varepsilon$$

where $\hat{\mathbf{c}} \leftarrow \mathsf{ObvCompress}(\mathbf{sk}_{\mathcal{E}}, p, I)$ and $[\tilde{c}_1, \ldots, \tilde{c}_n]^T \leftarrow$ $\mathsf{Decompress}(\mathbf{pk}_{\mathcal{E}}, \hat{\mathbf{c}})$*. We note that there are no correctness requirements for ciphertexts $\tilde{c}_i$ such that $i \notin I$.*

The scheme is computationally oblivious if, for all pairs of plaintext vectors $p = [p_1, \ldots, p_n]^T$ and $p' = [p'_1, \ldots, p'_n]^T$ and pairs of index sets $I, I' \subset [n]$ of size t, a computationally adversary cannot distinguish between the following:

- $\hat{\mathbf{c}} \leftarrow \mathsf{Compress}(\mathbf{sk}_{\mathcal{E}}, p, I)$
- $\hat{\mathbf{c}}' \leftarrow \mathsf{Compress}(\mathbf{sk}_{\mathcal{E}}, p', I')$.

2.3 Batch PIR and Labeled PSI

Batch (Keyword) PIR. In batch keyword PIR, the client holds a batch of ℓ keys, $\{q_1, \ldots, q_\ell\}$, and the server holds a public database $D \in (\mathcal{K} \times \mathcal{V})^n$ of n key-value pairs with n distinct keys, $\{(k_1, v_1), \ldots, (k_n, v_n)\}$. The client wishes to retrieve the database entries $\{D[q_1], \ldots, D[q_\ell]\}$ from the server. For any $q \in \mathcal{K}$, $D[q]$ denotes the value associated with key q. If $q = k_i$, then $D[q] = v_i$. Otherwise, $D[q] = \perp$. The following two properties must hold:

- *Correctness*: If the protocol is executed correctly, the client recovers $\{D[q_1], \ldots, D[q_\ell]\}$ as desired.
- *Query Privacy*: The server learns no information about the batch query, $\{q_1, \ldots, q_\ell\}$.

One can obtain the definition of single-query PIR if the batch query contains only a single index, $\ell = 1$. Furthermore, one can obtain non-keyword PIR if we restrict the database's key universe to be $\mathcal{K} = [n]$. Throughout our work, we will consider keyword PIR unless otherwise specified.

(Unbalanced) Labeled PSI. In labeled PSI, the receiver and sender hold sets X and Y respectively. The sender also holds a database of associated labels $\{L_y \mid y \in Y\}$. The goal is for the receiver to receive labels that appear in the intersection, $\{(z, L_z) \mid z \in X \cap Y\}$. The following properties must hold:

- *Correctness*: If the protocol is executed correctly, the receiver recovers $\{(z, L_z) \mid z \in X \cap Y\}$ as desired.

- *Receiver (Query) Privacy*: The sender learns no information about the receivers's set X beyond its size $|X|$.
- *Sender (Database) Privacy*: The receiver learns no information about the sender's set Y except for the desired output and its size $|Y|$.

In the unbalanced setting, the receiver's set X is typically much smaller than the sender's set Y, $|X| \ll |Y|$. Note, labeled PSI is similar to batch keyword PIR with the main difference being the additional sender (database) privacy guarantee.

3 Oblivious Ciphertext Compression

In this section, we present our oblivious ciphertext compression scheme, LSObvCompress, based on linear systems. We start with a simpler scheme before presenting our main construction.

3.1 First Attempt: Balls-into-Bins

In this section, we start with a construction which leverages the balls-into-bins random process. Given m bins and n balls, each of the n balls are thrown into one of the m bins uniformly at random. In the context of ciphertext compression, bins correspond to compressed ciphertexts and balls correspond to input non-zero ciphertexts. Throwing a ball into a bin corresponds to homomorphically adding an input ciphertext to one of the compressed ciphertexts. Decompression works by re-simuluating the ball throws for non-zero ciphertexts and decrypting the values at relevant bins. The main observation is that adding a zero-encrypting ciphertext can be thought of as "skipping" the ball throw, as its addition doesn't change the value of the underlying plaintext. Conceptually, the algorithm fails if any of the bins contains more than one ball. We describe the algorithm below.

We suppose that both parties share a hash function H. Upon receiving the input ciphertexts $\tilde{\mathbf{c}} = [\tilde{c}_1, \dots, \tilde{c}_n]^T$ and the number of non-zero plaintext entries t, the compression algorithm first initializes a vector of $m \geq t$ zero ciphertexts $\hat{\mathbf{c}} = [\hat{c}_1, ..., \hat{c}_m]^T$, where $\hat{c}_i = \mathcal{E}.\mathsf{Enc}(\mathbf{pk}_{\mathcal{E}}, 0)$. Then, for each input ciphertext $\tilde{c}_i$, the algorithm executes the following two operations. First, compute index $j = H(i) \in [m]$ where H is a random function with range $[m]$. Next, homomorphically add $\tilde{c}_i$ to $\hat{c}_j$, that is, $\hat{c}_j = \mathcal{E}.\mathsf{Eval}(\mathbf{pk}_{\mathcal{E}}, +, [\tilde{c}_i, \hat{c}_j])$. Finally, the algorithm outputs the resulting vector $\hat{\mathbf{c}}$.

The decompression algorithm receives the compression $\hat{\mathbf{c}} = [\hat{c}_1, \dots, \hat{c}_m]^T$ and non-zero plaintext entry indices I. For every non-zero ciphertext index $i \in I$, the algorithm computes $j = H(i)$ and sets $p_i = \mathcal{E}.\mathsf{Dec}(\mathbf{sk}_{\mathcal{E}}, \hat{c}_j)$. Finally, the algorithm outputs all non-zero plaintext values, $\{i, p_i\}_{i \in I}$.

Note this algorithm can recover the original plaintext vector as long as the hash outputs $H(i)$ are all distinct for every $i \in I$. However, the probability of collision is high unless $m = \Omega(t^2)$ (due to the birthday problem) that is a quadratic blowup with respect to t. Ideally, we would like m to be not much larger than t to obtain an efficient compression rate.

Reformulating as a Linear System. We generalize the aforementioned scheme as constructing and solving a system of linear equations. More specifically, the compression algorithm is responsible for constructing a linear system that the decompression algorithm attempts to solve to recover the original plaintext vector. While this viewpoint seems rather unnecessarily complex, it will serve as an important basis to our main construction. We outline the reformulated algorithm below.

For each $i \in [n]$, the compression algorithm constructs a column vector $\mathbf{v}_i \in \mathbb{F}^m$ where only the $H(i)$-th element is set to 1 and the rest are set to 0. Let $\mathbf{M} = [\mathbf{v}_1, ..., \mathbf{v}_n] \in \mathbb{F}^{m \times n}$ be a matrix. Note that both parties know matrix $\mathbf{M}$ as they share hash function H. The compression algorithm computes and outputs the matrix-vector multiplication $\hat{\mathbf{c}} = \mathbf{M} \cdot \tilde{\mathbf{c}}$.

The decompression algorithm takes in the vector $\hat{\mathbf{c}}$ and produces its decryption $\hat{\mathbf{p}}$. Next, we reconstruct the matrix $\mathbf{M}$ using the random function H. Let $I = \{i_1, ..., i_t\}$ be the set of non-zero plaintext entry indices, and let $\mathbf{M}_{\mathsf{c}(I)} = [\mathbf{v}_{i_1}, ..., \mathbf{v}_{i_t}] \in \mathbb{F}^{m \times t}$ be a sub-matrix of $\mathbf{M}$ consisting of all column vectors whose indices appear in I. Similarly, let $\hat{\mathbf{p}}_I = [\hat{p}_{i_1}, \dots, \hat{p}_{i_t}]$ for entries of $\hat{\mathbf{p}}$ in I. The algorithm solves the linear system associated with $\mathbf{M}_{\mathsf{c}(I)}$ and $\hat{\mathbf{p}}$ to compute $\mathbf{p}_I$ satisfying $\mathbf{M}_{\mathsf{c}(I)} \cdot \mathbf{p}_I = \hat{\mathbf{p}}_I$ to recover the non-zero $p_{i_j} = (\mathbf{p}_I)_j$ for each $j \in [t]$.

We note that the decompression algorithm can correctly recover the plaintext vector if and only if the linear system $\mathbf{M}_{\mathsf{c}(I)} \cdot \mathbf{p}_I = \hat{\mathbf{p}}_I$ has a unique solution (that is, $\mathbf{M}_{\mathsf{c}(I)}$ has full column rank). For our choice of $\mathbf{M}$, this precisely happens when all hash outputs $H(i)$ are distinct for every $i \in I$.

3.2 Second Attempt: Random Matrices

Recall that in the first attempt, the generated matrix $\mathbf{M}$ consists of random column vectors with Hamming weight exactly one corresponding to the balls-into-bins process. This forced us to set the number of rows and the encoding size to $m = \Omega(t^2)$ to avoid collisions. Taking a closer look, we notice that the way we generate the column vectors are unnecessarily restrictive. Indeed, for our scheme to succeed, we only require the $\mathbf{M}_{\mathsf{c}(I)}$ to have a unique solution. There is no need to restrict rows to Hamming weight one vectors.

This crucial observation leads to the following approach. Instead of sampling random column vectors with Hamming weight 1, we instead sample column vectors uniformly at random from $\{0,1\}^m$. To do this, we can imagine the shared hash function $H : [n] \to \{0,1\}^m$ outputs random binary column vectors of length m. Then, the shared matrix is $\mathbf{M} = [H(1), \dots, H(n)]$. This way, the generated column vectors will be linearly independent with high probability even when m is small. The rest of the algorithm stays identical.

Failure Probability and Compression Rate. The algorithm's failure probability and compression rate will be parameterized by ε and t. Let $m = (1+\varepsilon)t$ be the number of rows. Even when ε is very small, the generated $m \times t$ matrix $\mathbf{M}_{\mathsf{c}(I)}$ has a unique solution except with negligible probability. For example, setting $m = t + \lambda$ with very small $\varepsilon = \lambda/t$, the system has full rank with probability $1 - 2^{-\lambda-1}$ (see [33]). The compression rate is almost optimal as the encoding contains $t + \lambda$ ciphertexts that is only λ more than the optimal minimum.

Running Time. Let $m = (1+\varepsilon)t$. We start by analyzing the compression time. Generating a random column vector $\in \{0,1\}^m$ takes $O(m)$ time, so the entire matrix generation takes $O(mn)$ time during compression. Computing the matrix-vector product takes $m \cdot n$ homomorphic ciphertext additions. The compression algorithm performs $O(m \cdot t)$ ciphertext-ciphertext additions. For decompression, we note that solving the linear system associated to $\mathbf{M}_{\mathsf{c}(I)}$ requires $O(m \cdot t^2)$ time using Gaussian elimination.

Comparison to the First Attempt. While the new algorithm can give us very high compression rate, it is computationally very inefficient. Compression requires $O(mt)$ time and decompression requires $O(mt^2)$ time using Gaussian elimination. In practice, this may not be so problematic when $t << n$, but as t grows, the scheme is computationally expensive. Ideally, we would like compression to be close to linear in the number of ciphertexts, n, and decompression to be close to linear in t. In contrast, the first attempt has horrible compression rate of $m = O(t^2)$, but is computationally more efficient. The compression algorithm requires only $O(n)$ time. Furthermore, decompression only used $O(t^2)$ time.

This raises the following question: is it possible to get the best of both worlds - an algorithm that achieves high compression rate but is also practically efficient? We show that this is possible in the next subsection.

3.3 LSObvCompress: Random Band Matrices

In prior attempts, we generated random matrices uniformly at random from $\{0,1\}^{m \times n}$. This allowed the associated random linear systems to be uniquely solvable with high probability even when $m = (1+\varepsilon)t$ was very small. However, solving this linear system is very inefficient, which made the previous scheme impractical for larger t. This is not too surprising, because the generated matrix is very dense. The expected number of non-zero matrix entries is $mn/2$. This suggests that the algorithm for solving the linear system must also have at least $O(mn)$ running time as well.

Looking closely, we again realize that we never needed the generated matrices to be sampled uniformly at random from $\{0,1\}^{m \times n}$. That is, as long as the associated linear system is uniquely solvable with high probability, the distribution itself is irrelevant to the security of the scheme. Therefore, we only require a matrix generation algorithm that generates a "small" linear system that is uniquely and efficiently solvable. For LSObvCompress, we consider random matrices that satisfy these two properties.

Random Band Matrices. There has been extensive research on the core algorithmic problem of generating sparse random matrices that are efficiently solvable. For LSObvCompress, we utilize the random band matrices of Dietzfelbinger and Walzer [28] that is the most efficient to our knowledge.

Random band matrices are constructed such that each row consists of a random band with width w, and all entries outside of the band are zero. Formally, let m be the length of each row of the matrix. For each row, a band start index s is chosen randomly from $[m-w+1]$, and each entry within the band, i.e. in range $[s, s+w)$, is a uniformly random bit from $\{0,1\}$. All other entries outside the range $[s, s+w)$ remain 0.

Intuitively, random band matrices are solvable in $O(nw)$ time because the generated random matrix is "almost diagonal" after the rows are sorted by the band start positions. Furthermore, each row reduction operation maintains an invariant where the number of non-zero entries per rows is $O(w)$ making Gaussian elimination very efficient.

Adaptation for LSObvCompress. Unfortunately, we are unable to directly apply random band matrices for LSObvCompress. Going back to the linear system framework presented in Section 3.1, the client will solve the linear system associated with the matrix $\mathbf{M}_{\mathsf{c}(I)}$. Recall that I is the subset of non-zero plaintexts, $\mathbf{M}$ is the chosen random matrix and $\mathbf{M}_{\mathsf{c}(I)}$ is the sub-matrix of $\mathbf{M}$ consisting of all the column vectors whose indices appear in I. Suppose we chose $\mathbf{M}$ to be a random band matrix. Unfortunately, $\mathbf{M}_{\mathsf{c}(I)}$ is not guaranteed to be a random band matrix. In particular, it is possible that I (and, thus, the columns) are chosen such that each matrix row will have a band much smaller than length w or be all zero. In this case, it is unclear if the matrix $\mathbf{M}_{\mathsf{c}(I)}$ still has a unique solution.

Instead, we will choose our matrix $\mathbf{M}$ using an adaptation of random band matrices to ensure that $\mathbf{M}_{\mathsf{c}(I)}$ is still efficiently solvable for any choice of non-zero plaintext indices I. To do this, we will instead choose $\mathbf{M}$ to be the tranpose of random band matrices. In other words, we will generate each column vector of $\mathbf{M}$ to consist of a random band of width w. To do this, we imagine both parties share two hash functions $\mathsf{H}_1 : [n] \to [m-w+1]$ and $\mathsf{H}_2 : [n] \to \{0,1\}^w$. For the i-th column of the shared matrix $\mathbf{M}$, $\mathsf{H}_1(i)$ denotes the start of the band and $\mathsf{H}_2(i)$ chooses the random w-bit band.

Next, we can consider any subset of non-zero plaintexts I and the associated sub-matrix $\mathbf{M}_{\mathsf{c}(I)}$. As each column vector consists of a random w-length band, $\mathbf{M}_{\mathsf{c}(I)}$ remains a transpose of a random band matrix. As the column and row rank of any matrix is identical, we can rely on the analysis of Dietzfelbinger and Walzer [28] to see that $\mathbf{M}_{\mathsf{c}(I)}$ will have a unique solution with high probability for any choice of I.

The only caveat is that we cannot apply the running time analysis of the random band row matrix construction, as the bands are constructed column-wise instead of row-wise.

Algorithm 1 LSObvCompress.ObvCompress algorithm

```
Input: $\mathbf{pk}_{\mathcal{E}}, \tilde{\mathbf{c}}, t, R$: public additively homomorphic encryption key, vector of $n$ ciphertexts, number of non-zero plaintext entries, and randomness.
Output: $\hat{\mathbf{c}}$: compressed encoding of $\tilde{\mathbf{c}}$.
  $m \leftarrow (1+\varepsilon)t$
  $\mathbf{M} \leftarrow 0^{m \times n}$
  for $i = 1, \ldots, n$ do
      $\mathbf{v_i} \leftarrow \mathsf{GenRandVec}(i, m; R)$
      $\mathbf{M}[:][i] \leftarrow \mathbf{v_i}$                ▷ Set the $i$th column to $\mathbf{v_i}$
  $\hat{\mathbf{c}} \leftarrow \mathbf{M} \cdot \tilde{\mathbf{c}}$           ▷ HE add using $\mathcal{E}.\mathsf{Eval}$ and $\mathbf{pk}_{\mathcal{E}}$
  return $\hat{\mathbf{c}}$
```

Algorithm 2 GenRandVec algorithm

```
Input: $i, m, R$: column index, column vector length, and randomness
Output: $\mathbf{v_i}$: generated random column vector
  $w \leftarrow$ band width
  $s \leftarrow \mathsf{H}_1(R \mid\mid i)$          ▷ Random value from $[m-w+1]$
  $\mathbf{u} \leftarrow \mathsf{H}_1(R \mid\mid i)$               ▷ Random $w$-bit band.
  $\mathbf{v_i} \leftarrow 0^m$
  for $j = 0, \ldots, w-1$ do
      $\mathbf{v_i}[s+j] \leftarrow \mathbf{u}[j]$
  return $\mathbf{v_i}$
```

Algorithm 3 LSObvCompress.Decompress algorithm

```
Input: $\mathbf{sk}_{\mathcal{E}}, \hat{\mathbf{c}}, I, R$: secret additively homomorphic encryption key, compressed encoding of ciphertexts, set of non-zero plaintext indices, and randomness.
Output: $\{i, p_i\}_{i \in I}$: original non-zero plaintext values.
  $m \leftarrow (1+\varepsilon)t$
  $\mathbf{M}_{\mathsf{c}(I)} \leftarrow 0^{m \times t}$                ▷ Initialize all zero matrix.
  for $i_j \in I = \{i_1, \ldots, i_t\}$ do
      $\mathbf{v}_{i_j} \leftarrow \mathsf{GenRandVec}(i_j, m; R)$
      $\mathbf{M}_{\mathsf{c}(I)}[:][j] \leftarrow \mathbf{v}_{i_j}$            ▷ Set the $j$th column to $\mathbf{v}_{i_j}$
  $\hat{\mathbf{p}} \leftarrow$ decryption of $\hat{\mathbf{c}}$ using $\mathcal{E}.\mathsf{Dec}$ and $\mathbf{sk}_{\mathcal{E}}$
  $p_I \leftarrow \mathsf{SolveLinearSystem}(\mathbf{M}_{\mathsf{c}(I)}, \hat{\mathbf{p}})$
  if $p_I = \perp$ then
      return $\perp$
  $\mathbf{p} \leftarrow \emptyset$
  for $i_j \in I = \{i_1, \ldots, i_t\}$ do
      $\mathbf{p} \leftarrow \mathbf{p} \cup \{(i_j, (p_I)_j)\}$
  return $\mathbf{p}$
```

Algorithm 4 SolveLinearSystem algorithm

```
Input: $\mathbf{M}, \hat{\mathbf{p}}$: LHS matrix, RHS values to solve for
Output: $\mathbf{p}$: solution to the linear system $\mathbf{M} \cdot \mathbf{p} = \hat{\mathbf{p}}$
  $(\mathbf{M}^{\pi}, \pi) \leftarrow$ column sorting of the matrix $\mathbf{M}$ in ascending band start positions, along with the corresponding permutation that produces the column sorted matrix (e.g. $\mathbf{M}^{\pi}[:][i] = \mathbf{M}[:][\pi(i)]$)
  $\mathbf{p}^{\pi} \leftarrow$ execute Gaussian elimination on $\mathbf{M}^{\pi}$ and $\hat{\mathbf{p}}$, $\perp$ if no unique solution
  if $\mathbf{p}^{\pi} = \perp$ then
      return $\perp$
  $\mathbf{p} \leftarrow 0^t$
  for $i = 1 \ldots t$ do
      $\mathbf{p}[\pi(i)] \leftarrow p^{\pi}[i]$
  return $\mathbf{p}$
```

Nonetheless, we show that solving the system remains practically efficient with this modification in our experiments (see Section 7.1). Intuitively, this is because a transpose of a random band row matrix remains similar to a random band row matrix after the columns are sorted by the band start positions. The maximum band width across the entire rows is not much larger than the column band width w, which allows the linear system to be solved efficiently just as in the random band row matrix construction. See Figure 3 for an illustration. We prove the following theorem (see Appendix A for the proof):

Theorem 1. *Consider a $m \times t$ matrix with $m = (1+\varepsilon)t$ where each column consists of a single random w-bit band. For constant $\varepsilon > 0$ and band length $w = O(\lambda + \log t)$, the random band matrix has column rank n and executing Gaussian elimination after sorting the columns by the starting location of the band runs in time $O(tw)$ except with probability $2^{-\lambda}$.*

We now formally present LSObvCompress using random band matrices. See Algorithms 1 and 3 for the description of the oblivious compression and decompression algorithms.

Next, we analyze the properties of LSObvCompress showing that it combines the good compression rates and efficient encoding/decoding times of our prior two attempts.

Failure Probability and Compression Rate. For the failure probability, we note that LSObvCompress fails only when $\mathbf{M}_{\mathsf{c}(I)}$ does not have a unique solution or that unique solution cannot be found. By Theorem 1, we know this occurs with probability at most $2^{-\lambda}$ assuming that $w = O(\lambda/\varepsilon + \log n)$.

In our experiments, we will use concrete parameters for w and ε for various values of t to obtain 2^{-40} error probability. We point readers to Section 7.1 for more details. For the compression rate, our experiments show that ε may be as small as 0.05. As a result, LSObvCompress obtains compression rates that are only 5% larger than optimal.

Running Time. We start by analyzing the compression algorithm that computes the matrix multiplication of $\mathbf{M}$ and the input ciphertext vector $\tilde{\mathbf{c}} = [\tilde{c}_1, \ldots, \tilde{c}_n]^T$. As $\mathbf{M}$ is a binary matrix with at most nw non-zero entries, this can be performed using at most nw ciphertext-ciphertext additions. For decompression, we note that the main cost is solving the linear system $\mathbf{M}_{\mathsf{c}(I)}$ that requires $O(tw)$ time by Theorem 1 that is corroborated by our experiments (see Section 7.1).

Noise Growth for SHE. Recall that for the applications to

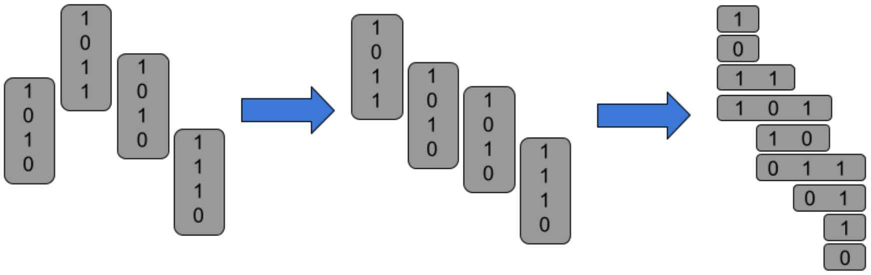

Figure 3: Example of a random band column matrix construction with band width $w = 4$. Second diagram shows the matrix after sorting the columns by the band start positions. Third diagram shows the random band row matrix view of the constructed matrix. In this example, the maximum band row width is 3.

batch PIR and labeled PSI, we will initialize LSObvCompress, using lattice-based SHE schemes. Therefore, noise growth is an important factor to consider. Suppose that the input ciphertexts $\tilde{\mathbf{c}} = [\tilde{c}_1, \ldots, \tilde{c}_n]^T$ each have error at most $\mathsf{Err}(\tilde{c}_i) \le e$. We note that the compression algorithm requires computing the sum of at most w ciphertexts. Therefore, each ciphertext in the compressed output has error at most $O(w \cdot e)$ as ciphertext-ciphertext additions only incur linear noise growth (see the full version for more details). As decompression is done after decryption, we do not need to worry about noise growth for decompression.

3.4 Comparison with Sparse Random Linear Codes [45, 48]

Liu and Tromer [48] implicitly study oblivious ciphertext compression. They observe that Sparse Random Linear Codes (SRLCs) [45], which use matrices $\mathbf{M} \in \mathbb{F}^{m \times n}$ where each column has a small number of non-zero entries drawn randomly from $\mathbb{F}$ can be used. However, each entry is sampled independently, in an unstructured way, which results in larger encodings than with LSObvCompress. Indeed, they show that such matrices can be sampled with full rank with high probability only if $m = O(t \log^2 t \log \lambda)$, which is larger than $m = 1.05t$ of LSObvCompress. Moreover, because SRLCs are unstructured, Gaussian elimination takes $O(t^3)$ time, resulting in slower $O(t^3)$ decoding time, compared to the $O(t \cdot \lambda)$ decoding time of LSObvCompress. Finally, since SRLCs use elements drawn randomly from $\mathbb{F}$, when used with FHE, large parameters must be used to handle the noise when multiplying ciphertexts by these large elements. However, LSObvCompress only uses elements from $\{0, 1\}$, which means that ciphertexts are only added together, resulting in minimal noise growth.

4 Oblivious Ciphertext Decompression

We show that similar ideas that we used to solve the oblivious ciphertext compression problem may also be used to solve the oblivious ciphertext decompression problem. As a reminder, in this problem, the compressor is given a plaintext vector, $\mathbf{p} = [p_1, \ldots, p_n]^T$, and t relevant indices $I \subset [n]$. The goal is for the decompressor to decode ciphertexts $\tilde{\mathbf{c}} = [\tilde{c}_1, \ldots, \tilde{c}_n]^T$ such that $\tilde{c}_i$ is an encryption of p_i for all relevant indices $i \in I$. There are no requirements for any $i \notin I$.

Description of LSObvDecompress. Essentially, we will apply the ideas of LSObvCompress, but in reverse. That is, we will start with a matrix $\mathbf{M}$ of dimension $n \times m$ (that both the compressor and decompressor can generate based on shared randomness), where $m = (1 + \varepsilon)t$ is the encoding length for some constant $\varepsilon > 0$. Then, based on relevant row indices $I = \{i_1, \ldots, i_t\} \subset [n]$, the compressor will solve the linear system formed by a $(t \times m)$-dimensional sub-matrix $\mathbf{M}_{\mathsf{r}(I)}$ of $\mathbf{M}$ and vector $\mathbf{p}_I = [p_{i_1}, p_{i_2}, \ldots, p_{i_t}]$, to obtain compressed plaintext vector $\hat{\mathbf{p}}$ of dimension m. Specifically, the compressor will solve the linear system for $\hat{\mathbf{p}}$ satisfying $\mathbf{M}_{\mathsf{r}(I)} \cdot \hat{\mathbf{p}} = \mathbf{p}_I$ using sub-matrix $\mathbf{M}_{\mathsf{r}(I)} = [\mathbf{M}_{i_1}^T, \ldots, \mathbf{M}_{i_t}^T]$.

Afterwards, the vector $\hat{\mathbf{p}}$ is encrypted entry-wise. The encrypted version of $\hat{\mathbf{p}}$ is the final encoding that we denote $\hat{\mathbf{c}}$. If the linear system according to $\mathbf{M}_{\mathsf{r}(I)}$ is not solvable, then the encoding fails and the compressor outputs any m encryptions. In applications, this is the point where we can utilize packing techniques where multiple plaintext values may be encrypted into a single ciphertext (as done in [14]).

For oblivious decompression, the decompressor computes $\mathbf{M} \cdot \hat{\mathbf{c}}$ homomorphically. Intuitively, this gives the decompressor ciphertext vector $\tilde{\mathbf{c}} = [\tilde{c}_1, \ldots, \tilde{c}_n]^T$ such that for each $i_j \in I$, the underlying plaintext of $\tilde{c}_{i_j}$ is $\mathbf{v}_{i_j} \cdot \hat{\mathbf{p}}$, which is exactly p_{i_j}, as desired. For every $\tilde{c}_{i_j}$ where $i_j \notin I$, the underlying plaintext will be some arbitrary linear combination of the entries of $\hat{\mathbf{p}}$, but recall that these values need not be correct.

For the choice of matrix $\mathbf{M}$, we can in fact generate it similarly as in LSObvCompress as a random band matrix of dimension $n \times m$, where each row consists of a single random band of length w. Note, this is the original random band matrix construction [28] without modification.

We present the pseudocode for LSObvDecompress in Algorithms 5 and 6.

Failure Probability. From above, we saw that the encoding is correct as long as the compressor can solve the linear system associated with $\mathbf{M}_{\mathsf{r}(I)}$. For the failure probability, we can simply calculate the probability that $\mathbf{M}_{\mathsf{r}(I)}$ does have a unique solution (or it cannot be found). As $\mathbf{M}$ is a random band matrix, we know that $\mathbf{M}_{\mathsf{r}(I)}$ is also a random band matrix. Therefore, if we set the band length $w = O(\lambda/\varepsilon + \log t)$ and

Algorithm 5 LSObvDecompress.Compress algorithm

```
Input: $\mathbf{sk}_{\mathcal{E}}, \mathbf{p} = [p_1, \ldots, p_n]^T, R$: secret additively homomorphic encryption key, plaintext values and randomness.
Output: $\hat{\mathbf{c}}$: compressed ciphertexts.
  Compute $I = \{i \mid p_i \neq 0\} \subseteq [n]$.
  If $|I| > t$, abort.
  If $|I| < t$, arbitrarily add indices to $I$ until $|I| = t$.
  $m \leftarrow (1+\varepsilon)t$
  $\mathbf{M}_{\mathsf{r}(I)} \leftarrow 0^{t \times m}$                ▷ Initialize all zero matrix.
  for $i \in I$ do
      $\mathbf{M}_i \leftarrow \mathsf{GenRandVec}(i, m, R)^T$
  $\hat{\mathbf{p}} \leftarrow \mathsf{SolveLinearSystem}(\mathbf{M}_{\mathsf{r}(I)}, \mathbf{p}_I)$
  $\hat{\mathbf{c}} \leftarrow$ encryption of $\hat{\mathbf{p}}$ using $\mathcal{E}.\mathsf{Dec}$ and $\mathbf{sk}_{\mathcal{E}}$
  return $\hat{\mathbf{c}}$
```

Algorithm 6 LSObvDecompress.ObvDecompress algorithm

```
Input: $\hat{\mathbf{c}}, R$: compressed ciphertexts and randomness.
Output: $\tilde{\mathbf{c}}$: decompressed ciphertexts.
  $m \leftarrow (1+\varepsilon)t$
  for $i \in [n]$ do
      $\tilde{c}_i \leftarrow \mathsf{GenRandVec}(i, m, R) \cdot \hat{\mathbf{c}}$
  $\tilde{\mathbf{c}} \leftarrow [\tilde{c}_1, \ldots, \tilde{c}_n]$
  return $\tilde{\mathbf{c}}$
```

$\mathbf{M}_{\mathsf{r}(I)}$ to be a $t \times (1+\varepsilon)t$, then $\mathbf{M}_{\mathsf{r}(I)}$ has a unique solution.

Compression Rate. We show that ε may be as small as 0.05 using experimental evaluation (see Section 7.1). Note, this is only 5% larger than the minimum of t ciphertexts since t plaintext values must be correctly encoded.

Running Time. The compression algorithm requires solving the linear system associated to $\mathbf{M}_{\mathsf{r}(I)}$ that is a $t \times (1+\varepsilon)t$ random band matrix. This can be done in $O(tw)$ time using only plaintext operations. Additionally, the resulting vector must be encrypted using $O(m) = O(t)$ time.

Decompression simply requires computing the matrix-vector multiplication $\mathbf{M} \cdot \hat{\mathbf{c}}$. As each row has at most w one entries, this requires $O(nw)$ homomorphic additions.

Obliviousness. Note that $\hat{\mathbf{c}}$ is always a length-m ciphertext vector. Reducing to the security of the underlying encryption scheme $\mathcal{E}$, we can replace each of these ciphertexts with encryptions of 0 meaning $\hat{\mathbf{c}}$ is independent of input plaintexts.

Noise Growth for SHE. Recall that for the applications to batch PIR and labeled PSI, we will initialize LSObvCompress, using lattice-based SHE schemes. Therefore, noise growth is an important factor to consider. We note that the compression algorithm is performed in plaintext without any homomorphic operations. Therefore, we only consider noise growth for decompression. The compressed input consists of m fresh SHE ciphertexts. Decompression adds at most w ciphertexts. If the input ciphertexts $\tilde{\mathbf{c}} = [\tilde{c}_1, \ldots, \tilde{c}_m]^T$ have error $\mathsf{Err}(\tilde{c}_i) \leq e$ for all $i \in [m]$, then each output ciphertext has error at most $O(w \cdot e)$. See the full version for further details.

5 Batch PIR

We will present three single-server PIR schemes using our compression techniques. We refer readers to Section 7.2 for our experimental evaluation to choose the best option for various settings of database size, entry size and batch size. We also present an improved two-server scheme.

5.1 Single-Server: Compressed Responses

In this section, we present our improved single-server batch PIR with compressed responses that apply for large entries that cannot leverage vectorization techniques [52].

Cuckoo Hashing Batch PIR Framework. In this section, we review the Cuckoo Hashing Batch PIR Framework by Angel *et al.* [14]. In a naive batch PIR scheme, the server would process each of the ℓ queries on the entire n database entries, resulting in a total of $O(n\ell)$ server operations. To reduce server computation, Angel *et al.* [14] presented a batch PIR framework that cleverly utilizes cuckoo hashing to encode both the batch query and the database entries. To date, this is the most practically efficient approach to constructing a batch PIR scheme. Our batch PIR will be built directly from this framework.

In this framework, the server setup works by creating $B \geq \ell$ independent single-query PIR servers and replicating each of the n database entries appropriately to a subset of $\alpha \geq 1$ servers. Consider a sparse database $D = \{(k_1, v_1), \ldots, (k_n, v_n)\} \in (\mathcal{K} \times \mathcal{V})^n$. Concretely, the choice of the α-subset is determined by the individual database entry (k_i, v_i) and α independent hash functions $H_1, \ldots, H_\alpha : \mathcal{K} \to [B]$ mapping keys to one of the B servers. In particular, (k_i, v_i) will be replicated to the servers indexed by $H_1(k_i), \ldots, H_\alpha(k_i)$. The total number of entries across all B servers will be $n\alpha$.

The hash functions that will be shared between the client and the server so that the client may also perform batch queries. Given a batch query $\{q_1, \ldots, q_\ell\}$, the client performs cuckoo hashing to map the ℓ query keys into the B buckets. In particular, each bucket will contain at most one query key after cuckoo hashing. Then, the client constructs a single-query PIR request for each of the B PIR servers. For empty buckets, the client will construct dummy "zero" requests such that the response to a dummy request will be a ciphertext that encrypts zero. Concretely, $B - \ell$ dummy requests. Finally, the server will process the B independent single-query PIR requests and send B responses back to the client.

Concrete Instantiation. Angel *et al.* [14] empirically determined that setting $B = 1.5\ell$ and $\alpha = 3$ results in an appropriate balance between the failure probability of the client allocation procedure, and efficiency. We notice that the request and response size in the cuckoo hashing framework is larger than the naive approach. In the naive approach, the request and response size are merely ℓ ciphertexts whereas the framework requires 1.5ℓ ciphertexts. This is 50% larger than the number of responses in the naive approach.

Client Mapping or Keyword PIR. One subtlety of this framework is that each of the B independent single-query PIR servers consists of a sparse database. We note that this is true regardless of whether the original batch PIR problem consists of a dense database where $\mathcal{K} = [n]$ or a sparse database where $\mathcal{K}$ could be much larger. In earlier works (such as [14, 52]), it was suggested to use $O(n)$ client mappings to convert from database indices to bucket indices. Recent work [57] instead directly uses state-of-the-art keyword PIR schemes to avoid linear client storage. Throughout the rest of our work, we will follow this approach and use single-query keyword PIR protocols for each of the B buckets.

Keyword PIR Framework [57]. We provide a brief overview of the keyword PIR framework of [57] (used for each of the B buckets), paying special attention to those details relevant to our final batch PIR scheme.

- As in recent standard PIR schemes [13, 14, 51, 53], the framework represents the n-entry database as a $d_1 \times d_2 \times \cdots \times d_z$ hypercube, where $d_1 \cdots d_z \geq n$.
- The query algorithm for some key k creates z vectors of length $d_1, \ldots, d_z$, homomorphically encrypts them, then uploads them to the server.
- The server takes in an encoding $\mathbf{E}$ of the database, a two-dimensional matrix of size $d_1 \times \lceil n/d_1 \rceil$, and homomorphic encryptions of vectors $\mathbf{v}_1, \ldots, \mathbf{v}_z$. It first applies $\mathbf{v}_1$ to $\mathbf{E}$ to obtain a $\lceil n/d_1 \rceil$ vector, arranges this vector into a $d_2 \times \lceil n/(d_1 d_2) \rceil$ matrix and applies $\mathbf{v}_2$ to obtain a vector of size $\lceil n/(d_1 d_2) \rceil$, and repeats this for all z dimensions (where for the last dimension, the vector from the previous step will not be arranged into a matrix and instead, an inner product with $\mathbf{v}_z$ will be performed). At the end of this process, the server obtains a ciphertext encrypting the queried entry of the client.
- The server then sends this ciphertext to the client, who can decrypt it to obtain their queried entry.

Our Construction. To reduce communication in batch PIR, we will apply LSObvCompress to reduce the server response communication in the cuckoo hashing framework.

Namely, recall that for the $B - \ell$ buckets which do not have an associated key, the client will construct dummy “zero” requests such that the corresponding response ciphertext will encrypt zero. This can be done by setting, e.g., $\mathbf{v}_z$ to the zero-vector, since in the last step of the response algorithm, the server computes the inner product of $\mathbf{v}_z$ with some vector to obtain the final response ciphertext. Therefore, after the server processes the $B = 1.5\ell$ requests, it obtains $B = 1.5\ell$ responses of which ℓ consist of encrypted entries, and the rest are encrypted zeros. Thus, the server can apply the compression of LSObvCompress with $n = B$ and $t = \ell$ to obtain compressed ciphertexts. Of course, the client knows the indices of the ℓ real requests. As a result, the client can execute the decompression of LSObvCompress to obtain the requested entries.

Our construction therefore results in response size with overhead as small as 1.05× the optimal, with minimal added computation, instead of the 1.5× overhead in response size of [14].

Noise Growth. For noise growth, we will assume that the keyword PIR framework [57] is applied using recent PIR schemes from SHE composition [51, 53]. We perform the same noise analysis for prior PIR schemes in the full version and see that our new PIR scheme increases the noise growth by a $O(w)$ multiplicative factor.

5.2 Single-Server: Compressed Requests

Next, we apply LSObvDecompress to compress requests for single-server batch PIR schemes.

Our Construction. In our framework using the keyword PIR from [57], the client generates $B = 1.5\ell$ requests, each containing z vectors. However, we only need correct answers from ℓ requests. Thus, the client can combine all $B \cdot z$ request vectors into one long vector, and for relevant indices I consisting only of entries corresponding to the z vectors for each of the ℓ important requests, apply LSObvDecompress. This will result in a compressed request with size overhead only 1.05× compared to the naive batch PIR, which the client can then encrypt and send to the server. This is in contrast to the 1.5× overhead in request size of [14]. We also utilize in our construction the request packing techniques from [14] to fit multiple requests into a single ciphertext.

The server will first apply the request ciphertext packing decoding and, then run decompression from LSObvDecompress to obtain the B encrypted requests. Note, only the ℓ important requests will be correct. This is sufficient as the remaining 0.5ℓ dummy requests are ignored by the client anyways. The remainder of the server processing and client decrypting remains identical.

Noise Growth. We show that applying LSObvDecompress increases noise by an $O(w)$ multiplicative factor (see the full version for further details).

Why not both request and response compression? Theoretically, one can apply compression for both requests and responses simultaneously. In practice, the noise growth is $O(w^2)$ multiplicative factor (see the full version for the analysis) that is large. We were unable to find parameters where request and response compression beat either of the PIR schemes above. We leave it as an open problem to find better SHE/PIR schemes enabling both request and response compression. The full details of using both may be found in the full version.

5.3 Single-Server: Vectorized Responses

We present a method to compress responses in conjunction with the recent vectorization techniques of Mughees and Ren [52]. The vectorization techniques [52] utilize *Single-Instruction-Multiple-Data* (SIMD) techniques. SIMD encodes multiple database entries into a single ciphertexts (leveraging additional structure of the SHE scheme) and operates on all of them simultaneously.

Our Construction. The core idea of utilizing LSObvCompress to compress responses remains the same, but we wish to leverage that multiple entries fit into a single ciphertext. To do this, we present a vectorized version of LSObvCompress that optimally packs multiple entries into a single ciphertext. If d entries fit into a single ciphertext, our vectorized LSObvCompress sends only $\lceil 1.05\ell/d \rceil$ to the client. In contrast, $\lceil 1.5\ell/d \rceil$ ciphertexts are encoded in [52].

At a high level, vectorized LSObvCompress works nearly identically as the variant of LSObvCompress described in Section 3.3. The only difference is that we apply techniques from [52] to rotate ciphertexts and pack multiple entries into a ciphertext before performing compression. Due to lack of space, we defer the full description to the full version.

5.4 Two-Server: Compressed Responses

Next, we use LSObvCompress to compress responses for two-server batch PIR. In this setting, the client sends requests to both servers. Each server holds a copy of the database and cannot communicate with each other. They then send individual responses back to the client, who uses both to reconstruct the requested entry. The same correctness and privacy conditions are required. Privacy is considered with respect to each individual server assuming non-collusion.

Two-Server PIR. To date, the most concretely efficient two-server PIR schemes are built using distributed point functions (DPF) [19, 37, 40]. A point function $f_i : \mathcal{K} \rightarrow \{0,1\}$ satisfies $f_i(x) = 1$ if and only if $x = i$. DPFs enable secret sharing f_i amongst the two servers using two functions, f_i^0 and f_i^1, satisfying $f_i(x) = f_i^0(x) + f_i^1(x)$ for all $x \in \mathcal{K}$. Both f_i^0 and f_i^1 must not individually reveal anything about f_i.

To perform a two-server keyword PIR query for key k, the client uses DPFs to create secret shares of f_k, f_k^0 and f_k^1, that are sent to each of the two servers. Suppose the database consists of n key-value pairs, $D = \{(k_1, v_1), \ldots, (k_n, v_n)\}$. For each server $j \in \{0,1\}$, the j-th server computes

$$z_j = f_k^j(k_1) \cdot v_1 + \ldots + f_k^j(k_n) \cdot v_n.$$

Finally, the client receives z_0 and z_1 and computes the final answer $z_0 + z_1$. If $k = k_i$, then $z_0 + z_1 = v_i$. Otherwise, $z_0 + z_1 = 0$ when $k \notin \{k_1, \ldots, k_n\}$.

There is a small issue that the client cannot distinguish between $v_i = 0$ and $k \notin \{k_1, \ldots, k_n\}$. To fix this, we can ensure that zero is not a valid entry. For example, one can simply append a 1-bit to the end of each entry, $v_1, \ldots, v_n$.

Two-Server Batch PIR. To our knowledge, the most efficient two-server batch PIR remains the cuckoo hashing framework of Angel *et al* [14]. In the concrete instantiation, we use a two-server, single-query, keyword PIR for each of the $B = 1.5\ell$ buckets when performing a batch query for ℓ entries.

Our Construction. Our goal is to utilize LSObvCompress to reduce the number of responses from $B = 1.5\ell$ that are sent by both servers. We will use the two-server keyword PIR based on the DPF of [19]. Note that this PIR does not use encryption and, instead, relies on the non-collusion of the two servers for security. The encryption scheme $\mathcal{E}$ for LSObvCompress in this setting is additive secret sharing. Homomorphic additions will simply be addition operations by each server.

First, recall that in order to use LSObvCompress, the 0.5ℓ dummy requests must results in additive sharings (encryptions) of 0. We will add $(k_0, 0)$ is added to each of the B buckets for special key k_0. The client will issue a keyword PIR query for k_0 for each of the 0.5ℓ dummy buckets. The servers upon receipt of these requests for all B buckets will then compute the corresponding responses. Consider the i-th response z_i^0 and z_i^1 for both servers. Let $I' \subset [B]$ be the indices of the real, non-dummy requests. For all $i \in [B]$, $z_i = z_i^0 + z_i^1$ is the i-th requested entry. If $i \notin I'$, then $z_i = z_i^0 + z_i^1 = 0$.

We can thus apply LSObvCompress as follows. Both servers will use LSObvCompress to compress their responses $\mathbf{z}^0 = [z_1^0, \ldots, z_B^0]$ and $\mathbf{z}^1 = [z_1^1, \ldots, z_B^1]$. Recall that this is done by computing the matrix-vector multiplications $\hat{\mathbf{z}}^0 = \mathbf{M} \cdot \mathbf{z}^0$ and $\hat{\mathbf{z}}^1 = \mathbf{M} \cdot \mathbf{z}^1$ where $\mathbf{M}$ is the transpose of a random band matrix. The client will compute $\hat{\mathbf{z}} = \hat{\mathbf{z}}^0 + \hat{\mathbf{z}}^1 = \mathbf{M} \cdot (\mathbf{z}^0 + \mathbf{z}^1)$ that is a compression of the requested entries, $\mathbf{z}^0 + \mathbf{z}^1$. Finally, the client runs the decompression portion of LSObvCompress on $\hat{\mathbf{z}}$ to obtain the non-dummy queried entries, z_i for all $i \in I$.

6 Labeled PSI

In this section, we show our techniques may be used to build protocols for labeled PSI in the unbalanced setting. Recall that the receiver has a set X and the sender has a labeled set $\{(y, L_y) \mid y \in Y\}$. Note, that one may interpret this as batch keyword PIR with the receiver as the client and the sender as the server. The only difference is that PSI requires privacy for both party's input. Therefore, we need to also enable privacy for the sender (server) input. We present two improved constructions using our compression techniques.

Batch Keyword PIR and Oblivious PRF. We can use the generic transformation from [32] combining our single-server batch keyword PIR with any oblivious pseudorandom function (OPRF) to build our labeled PSI protocol. An OPRF allows the receiver to input set X and learn the set of pseudorandom outputs $\{F_k(x) \mid x \in X\}$, where F is a PRF, and k is known to the sender. For security, both the sender and receiver should learn nothing else (beyond the size of $|X|$). Our scheme provides full security against a malicious receiver and privacy against a malicious sender, as in prior works [21, 25].

At a high level, the protocol goes as follows. First, the sender generates a private key k and evaluates the OPRF on its input set Y. The labels $\{L_y \mid y \in Y\}$ are encrypted using keys derived from the OPRF evaluation. The sender and receiver execute the OPRF protocol on the receiver's input and a sender's private key. Finally, the sender and receiver execute a batch PIR using the receiver's output of the OPRF protocol to retrieve the encrypted labels. Afterwards, the receiver may decrypts labels in the intersection. We point readers to the full version for more details and analysis.

As a side note, we point out that we must modify the keyword PIR construction from [57] slightly. In particular, the database encoding algorithm must have negligible failure to ensure database privacy. To do this, we can increase the band length parameter to ensure negligible encoding failures (using the analysis from [18]). See the full version for more details.

Improving Oblivious Polynomial Evaluation. Prior works [21, 22, 25] built labeled PSI using oblivious polynomial evaluation (OPE). We can apply LSObvDecompress to OPE protocols for reducing request sizes by 30%. Although, we note prior work [52] showed that batch PIR approaches result in more efficient labeled PSI protocols compared to OPE. Nevertheless, OPE is an interesting application of our compression algorithms. See the full version for more details.

7 Experimental Evaluation

We perform experimental evaluation for our new compression algorithms, LSObvDecompress and LSObvCompress, as well as their improvements to batch PIR and labeled PSI. Finally, we also benchmark our protocols for the real world application of anonymous messaging.

Experimental Setup. We implemented our experimental evaluations with around 3000 lines of C++ code. All our experiments are performed using Ubuntu PCs with 96 cores, 3.7 GHz Intel Xeon W-2135 and 128 GB of RAM with only single-threaded execution. The AVX2 and AVX-512 instruction sets with SIMD instructions are enabled. The results are the average of at least 10 experimental trials with standard deviation less than 10% of the averages. Our implementations will target error probability 2^{-40} and 128 bits of computational security. Server monetary costs are computed using Amazon EC2 savings plan pricing of t2.2xlarge instances [4] of \$0.09 per GB of traffic and \$0.021 per CPU hour at the time. We will utilize SHA256 as the hash function and AES-GCM-256 as the encryption scheme with 32 byte keys. Unless otherwise specified, we will use the compression parameter $\varepsilon = 0.05$ for our experiments.

Interpreting the Experimental Results. As our compression schemes are general schemes that can be instantiated on various protocols, they incur additional computational overhead compared to the ones that don't use our compression schemes. To assess concrete tradeoffs between the computational overhead and the communication reduction, we will use the Amazon EC2 server monetary cost model which measures the communication and computational efficiency as a dollar cost. We note that this model has been used in prior works [18, 57] for this exact purpose.

7.1 Oblivious Ciphertext Compression

We first evaluate the performance of LSObvCompress and LSObvDecompress in isolation and report results in Figure 4.

Setup. In our experiments, we will use Regev encryption [58] as the underlying scheme using the implementation from Spiral [9]. We fix the plaintext size to 8 KB and the ciphertext size to 20 KB. In the figure, t corresponds to the number of non-zero entries and n corresponds to the total number of entries including the zero entries. We fix the fraction of zero entries to $0.5t$ (thus $n = 1.5t$). Note that this corresponds to the fraction of dummy requests/responses in the cuckoo hashing framework from [14]. In our evaluations, we will target two compression sizes of $1.05t$ and $1.07t$. Note that this results in 30% and 29% request/response size reduction respectively.

Results. We see that computation time increases with better compression rate as well as larger t and n. However, we claim that our LSObvCompress and LSObvDecompress remain practically efficient for many applications; as we will show in the next sections, the additional computational cost is a relatively small fraction of the entire protocol's computation time, and the significant reduction in the communication cost will justify these small additional computational overhead.

7.2 Single-Server Batch PIR

We evaluate the single-server batch PIR schemes from Section 5 using our compression techniques to reduce communication. We report our results in Figure 5.

Setup. We implement our compression algorithms on top of the open-source Spiral implementation [9]. We use $n = 1$ million database entries for all of our results. Baseline corresponds to Angel *et al* [14]'s batch PIR framework implemented on top of Spiral [51] without our compression techniques. The parameters for the baseline were chosen using the script provided by their open-source implementation. In our evaluations, we consider three batch sizes $\ell \in \{512, 1024, 2048\}$ (in the context of ciphertext compression/decompression, the batch size ℓ corresponds to the number of non-zero entries). We follow the batch PIR setup from [14] and fix the fraction of dummy requests/responses (i.e. zero entries) to 0.5ℓ. We target compression size of 1.05ℓ.

Compression Size	Sizes & Schemes	Compression Time	Decompression Time	Total Time
$1.05t$	$\mathbf{t = 512, n = 768}$			
	LSObvCompress	2.38 s	0.66 s	3.04 s
	LSObvDecompress	0.60 s	1.65 s	2.25 s
	$\mathbf{t = 1024, n = 1536}$			
	LSObvCompress	5.15 s	1.34 s	6.49 s
	LSObvDecompress	1.25 s	3.94 s	5.19 s
	$\mathbf{t = 2048, n = 3072}$			
	LSObvCompress	10.54 s	2.67 s	13.21 s
	LSObvDecompress	2.48 s	6.88 s	9.38 s
	$\mathbf{t = 4096, n = 6144}$			
	LSObvCompress	21.45 s	5.27 s	26.72 s
	LSObvDecompress	5.05 s	14.50 s	19.55 s
$1.07t$	$\mathbf{t = 512, n = 768}$			
	LSObvCompress	1.82 s	0.53 s	2.35 s
	LSObvDecompress	0.49 s	1.34 s	1.83 s
	$\mathbf{t = 1024, n = 1536}$			
	LSObvCompress	4.12 s	1.07 s	5.19 s
	LSObvDecompress	0.96 s	3.19 s	4.15 s
	$\mathbf{t = 2048, n = 3072}$			
	LSObvCompress	8.41 s	2.12 s	10.53 s
	LSObvDecompress	1.72 s	5.80 s	7.52 s
	$\mathbf{t = 4096, n = 6144}$			
	LSObvCompress	17.06 s	4.43 s	21.49 s
	LSObvDecompress	2.76 s	11.19 s	13.95 s

Figure 4: Evaluations of LSObvCompress and LSObvDecompress for different values of t (non-zero/relevant entries) and n (uncompressed input size). We fix the plaintext size to 8 KB and ciphertext size to 20 KB for all our results.

Results. Using our LSObvCompress, we see 30% response size reduction in exchange for a reasonable additional computational cost compared to state-of-the-art PIR for large entries without compression. We see that this small additional computation cost is justified by the reduction in the server monetary cost. In particular, LSObvCompress reduces the server monetary cost by up to 10% compared to the baseline.

Using our LSObvDecompress algorithm, we see 20-24% reduction in the request size with slight increase in computation and server monetary cost.

We were unable to integrate our vectorized version of LSObvCompress into the vectorized batch PIR protocol [52] as we are unaware of an open-source implementation. Nevertheless, we still implemented our vectorized version LSObvCompress from Section 5.3 using the SEAL library [59]. The results are presented in the full version.

Choosing the Right Protocol. In general, LSObvCompress provides the best communication and server monetary cost reduction. Thus, LSObvCompress will typically be the best option for most settings.

In certain settings, we note that LSObvDecompress may be useful where we wish to minimize upload communication from the client to the server. There are many natural settings where the upload costs/speed are more expensive/slower than the download costs/speed. For applications in these scenarios, it may be critical to save as much upload communication as possible that is achieved by LSObvDecompress.

Application: Anonymous Messaging. Angel and Setty [15] introduced Pung that built an anonymous messaging protocol using any single-server batch PIR (see the full version). In Figure 6, we report our results for retrieving 288-byte messages. We fix the number of database entries to $n = 1$ million and batch size to $b = 512$.

By using our improved batch PIR constructions, we obtain more communication-efficient versions of Pung. In particular, we see that LSObvCompress reduces the server monetary cost by 7% compared to the baseline.

7.3 Two-Server Batch PIR

We implement our response-compressed two-server batch PIR from Section 5.4 on top of the two-server single-query PIR implementation in [2]. We report our results in Figure 7.

Setup. We fix the number of database entries to $n = 1$ million where each database entry is 288 bytes large. As in the single-server batch PIR experiment (Section 7.2), we fix the fraction of dummy responses to 0.5ℓ and target compression size of 1.05ℓ. We omit evaluating request sizes as they are the same for both schemes.

Results. We observe that using LSObvCompress can reduce response size by 30% in exchange for a small additional computational cost. However, the small additional computational

DB Entry Size	**Batch Size & Schemes**	**Public Param Size**	**Request Size**	**Response Size**	**Total Server Time**	**Amortized Server Time**	**Total Client Time**	**Server Monetary Cost**
8 KB	$\ell = \mathbf{512}$							
	Baseline	20.87 MB	1.81 MB	15.59 MB	**840 s**	**1.64 s**	**6.1 s**	\$0.00646
	LSObvCompress	22.62 MB	1.81 MB	**10.92 MB**	890 s	1.74 s	6.7 s	**\$0.00633**
	LSObvDecompress	20.87 MB	**1.40 MB**	15.59 MB	863 s	1.69 s	6.4 s	\$0.00656
	$\ell = \mathbf{1024}$							
	Baseline	20.87 MB	3.35 MB	31.18 MB	**1,256 s**	**1.23 s**	**6.8 s**	\$0.01043
	LSObvCompress	23.11 MB	3.35 MB	**21.84 MB**	1,369 s	1.34 s	8.2 s	**\$0.01025**
	LSObvDecompress	20.87 MB	**2.55 MB**	31.18 MB	1,323 s	1.29 s	8.0 s	\$0.01075
	$\ell = \mathbf{2048}$							
	Baseline	20.87 MB	3.96 MB	62.37 MB	**1,750 s**	**0.85 s**	**7.0s**	\$0.01617
	LSObvCompress	23.43 MB	3.96 MB	**43.67 MB**	1,871 s	0.91 s	9.7 s	**\$0.01520**
	LSObvDecompress	20.87 MB	**3.15 MB**	62.37 MB	1,812 s	0.89 s	9.4 s	\$0.01646
16 KB	$\ell = \mathbf{512}$							
	Baseline	20.87 MB	1.81 MB	31.18 MB	**1,286 s**	**2.51 s**	**6.3 s**	\$0.01047
	LSObvCompress	22.62 MB	1.81 MB	**21.84 MB**	1,348 s	2.63 s	8.4 s	**\$0.00999**
	LSObvDecompress	20.87 MB	**1.40 MB**	31.18 MB	1,308 s	2.55 s	8.3 s	\$0.01056
	$\ell = \mathbf{1024}$							
	Baseline	20.87 MB	3.35 MB	62.37 MB	**1,775 s**	**1.73 s**	**7.9 s**	\$0.01626
	LSObvCompress	23.11 MB	3.35 MB	**43.69 MB**	1,929 s	1.88 s	9.6 s	**\$0.01548**
	LSObvDecompress	20.87 MB	**2.55 MB**	62.37 MB	1,881 s	1.83 s	9.4 s	\$0.01681
	$\ell = \mathbf{2048}$							
	Baseline	20.87 MB	3.96 MB	124.74 MB	**2,634 s**	**1.29 s**	**8.5 s**	\$0.02694
	LSObvCompress	23.43 MB	3.96 MB	**87.34 MB**	2,773 s	1.35 s	12.4 s	**\$0.02439**
	LSObvDecompress	20.87 MB	**3.15 MB**	124.74 MB	2,746 s	1.34 s	12.4 s	\$0.02752

Figure 5: Evaluations of Spiral Batch PIR [14, 51] with and without our compression techniques, LSObvCompress and LSObvDecompress with $\varepsilon = 0.05$. We fix the number of entries to $n = 1$ million for all our results.

Schemes	**Public Param Size**	**Request Size**	**Response Size**	**Total Server Time**	**Total Client Time**	**Server Monetary Cost**
Baseline	20.5 MB	0.86 MB	8.90 MB	**328 s**	**1.4 s**	\$0.00279
LSObvCompress	23.1 MB	0.86 MB	**6.23 MB**	338 s	1.6 s	**\$0.00260**
LSObvDecompress	20.5 MB	**0.66 MB**	8.90 MB	347 s	1.7 s	\$0.00288

Figure 6: Instantiation of the Pung messaging system [15] using batch Spiral PIR with and without our compression techniques ($\varepsilon = 0.05$). We fix the number of database entries to $n = 1$ million and batch size to $\ell = 512$. Each entry is of size 288 B.

Batch Size & Schemes	Response Size	Server Time	Client Time	Server Monetary Cost
$\ell = 512$				
Baseline	221 KB	**9.63 s**	**0.01 s**	$0.000076
LSObvCompress	**155 KB**	9.69 s	0.08 s	**$0.000070**
$\ell = 1024$				
Baseline	442 KB	**9.76 s**	**0.01 s**	$0.000096
LSObvCompress	**310 KB**	9.92 s	0.16 s	**$0.000085**
$\ell = 2048$				
Baseline	885 KB	**9.79 s**	**0.01 s**	$0.000136
LSObvCompress	**619 KB**	10.09 s	0.33 s	**$0.000114**
$\ell = 4096$				
Baseline	1,769 KB	**9.81 s**	**0.01 s**	$0.000216
LSObvCompress	**1,238 KB**	10.53 s	0.78 s	**$0.000172**
$\ell = 8192$				
Baseline	3,539 KB	**9.82 s**	**0.01 s**	$0.000375
LSObvCompress	**2,477 KB**	11.13 s	1.80 s	**$0.000287**

Figure 7: Comparison of DPF based two server batch PIR protocol [2] with and without LSObvCompress ($\varepsilon = 0.05$). We fix the number of database entries to $n = 1$ million and each entry size to 288 B for all our results.

cost is justified by the savings in the server monetary cost. Compared to the baseline, LSObvCompress can reduce the server monetary cost by up to 24%.

7.4 Labeled PSI

Next, we evaluate our labeled PSI built from our batch PIR in Section 5.1 and an OPRF protocol (see the full version for details). We report our results in Figure 8.

Setup. Our implementation uses the single-server batch PIR implementation from Section 5.1 with the OPRF implementation from [1]. We fix the size of the sender's set to 1 million and receiver's set to 512 (note, in the context of batch PIR these corresponds to the number of database elements and the batch size respectively). We have used one of the default parameter sets available in their open-source implementations for Cong *et al* [25]'s scheme.

Results. Our scheme has 65-88% reduced communication over prior state-of-the-art works [25]. For smaller label size, our construction with Spiral [51] is slower, but we start to catch up and eventually outperform Cong *et al* [25] for larger label sizes. Note that even with these additional computation cost, we reduce the server monetary cost by 30-70%.

Due to the limitation of their open source implementation [1], we could not compare our construction on larger label sizes, but we expect our scheme to outperform significantly as the label sizes increase. In any case, our communication cost and server monetary cost is significantly smaller.

Label Size & Schemes	Total Online Comm.	Total Online Time	Server Monetary Cost
512 B			
Cong *et al.* [25]	33.2 MB	**169 s**	$0.00397
LSObvCompress	**11.4 MB**	304 s	**$0.00279**
1024 B			
Cong *et al.* [25]	66.1 MB	**331 s**	$0.00787
LSObvCompress	**11.6 MB**	355 s	**$0.00311**
1536 B			
Cong *et al.* [25]	103.6 MB	535 s	$0.01244
LSObvCompress	**11.9 MB**	**446 s**	**$0.00367**

Figure 8: Comparisons of Cong *et al.* [25]'s labeled PSI and our LSObvCompress-based PSI with $\varepsilon = 0.05$. We fix the size of the sender's set to 1 million and the receiver's set to 512.

8 Related Works

Ciphertext Compression. Variants of ciphertext compression have been studied in the past. Liu and Tromer [48] implicitly studied oblivious ciphertext compression without explicitly defining the primitive. In their scheme, they use sparse linear random codes that result in larger encodings and slower decoding time (see Figure 1), and, if instantiated with a FHE scheme, larger parameters for that scheme. Angel *et al.* [14] used packing and vectorization techniques to reduce request communication in PIR. Mughees and Ren [52] also showed vectorization techniques may be used to reduce response communication in batch PIR. Fleischhacker *et al.* [31] studied a more challenging variant of our setting where neither the decompressor (client) nor the compressor (server) know the identity of the the non-zero ciphertexts. As a result, their schemes have worse compression rate and more expensive compression and decompression algorithms. The same problem was implicitly studied in [23].

PIR. Single-server PIR was first studied by Kushilevitz and Ostrovsky [46]. Follow-up works constructed PIR from various other assumptions [20, 26, 35, 47, 55]. More recent works have studied concretely efficient protocols from lattice-based homomorphic encryption [11–15, 34, 50, 51, 53, 56, 57].

PIR has also been studied in the setting of multiple, non-colluding servers. A line of work has studied the communication efficiency with information-theoretic security (see [24, 29, 30] and references therein). Recent works have studied concretely efficient two-server PIR with computational security using distributed point functions [19, 37, 40].

Batch PIR. Batch PIR has been studied heavily in the past. Beimel *et al.* [17] presented a method to reduce server computation using matrix multiplication. Groth *et al.* [39] presented a communication-optimal scheme adapting the scheme in [35]. Another line of work (see [41, 43, 49, 60] and references therein) presented batch codes that transforms any single-query PIR into a batch PIR. More recent work [14, 15] introduced probabilistic batch codes that result in the most concretely-efficient batch PIR schemes to date. Mughees and Ren [52] introduced vectorization techniques to reduce server responses for small database entries. Patel *et al.* [57] presented keyword PIR schemes that can remove the client mapping.

Labeled PSI. Labeled PSI is a variant where each identifier has an associated data label that should be retrieved. Labeled PSI is most often studied in the unbalanced setting where the receiver's set is much smaller than the sender's set. Many recent works [21, 22, 25, 27, 44] studied labeled PSI with sub-linear communication in the larger set. The same setting where the receiver only queries for a single item has been studied as symmetric PIR [13, 36].

9 Conclusions

In this work, we present state-of-the-art constructions for both batch PIR and labeled PSI with reduced communication costs compared to prior solutions. To do this, we identify a common task in both primitives that we denote as oblivious ciphertext compression where a compressor (server) is given n ciphertexts with only $t < n$ non-zero ciphertexts. The decompressor (client) knows the location of the t non-zero ciphertexts, but the compressor is unaware of this knowledge. We present $\mathsf{LSObvCompress}$ that enables compressions consisting of $1.05t$ ciphertexts that is only 5% larger than the minimum while requiring only additive homomorphism. Using $\mathsf{LSObvCompress}$, we present batch PIR schemes with 30% smaller responses and labeled PSI protocols with 65-88% reduced communication and comparable computation.

References

[1] APSI: C++ library for Asymmetric PSI. https://github.com/microsoft/APSI.

[2] C++ DPF-PIR library. https://github.com/dkales/dpf-cpp.

[3] Certificate transparency. https://certificate.transparency.dev/.

[4] EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/.

[5] Password monitor: Safeguarding passwords in microsoft edge. https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge/.

[6] Protect your accounts from data breaches with password checkup. https://security.googleblog.com/2019/02/protect-your-accounts-from-data.html.

[7] Protecting your device information with private set membership. https://security.googleblog.com/2021/10/protecting-your-device-information-with.html.

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

[9] Spiral. https://github.com/menonsamir/spiral.

[10] Technology preview: Private contact discovery for Signal. https://signal.org/blog/private-contact-discovery/.

[11] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR: Private information retrieval for everyone. *PoPETs*, 2016(2):155–174, April 2016.

[12] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *OSDI 21*, 2021.

[13] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication-computation trade-offs in PIR. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 1811–1828. USENIX Association, August 2021.

[14] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy*, pages 962–979. IEEE Computer Society Press, May 2018.

[15] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI 16*, pages 551–569, 2016.

[16] Gilad Asharov, Moni Naor, Gil Segev, and Ido Shahaf. Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In Daniel Wichs and Yishay Mansour, editors, *48th ACM STOC*, pages 1101–1114. ACM Press, June 2016.

[17] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In Mihir Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 55–73. Springer, Heidelberg, August 2000.

[18] Alexander Bienstock, Sarvar Patel, Joon Young Seo, and Kevin Yeo. Near-optimal oblivious key-value stores for efficient PSI, PSU and volume-hiding multi-maps. In *USENIX Security 2023*, 2023.

[19] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1292–1303. ACM Press, October 2016.

[20] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 402–414. Springer, Heidelberg, May 1999.

[21] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1223–1237. ACM Press, October 2018.

[22] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu,

editors, *ACM CCS 2017*, pages 1243–1255. ACM Press, October / November 2017.

[23] Seung Geol Choi, Dana Dachman-Soled, S. Dov Gordon, Linsheng Liu, and Arkady Yerukhimovich. Compressed oblivious encoding for homomorphically encrypted search. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2277–2291. ACM Press, November 2021.

[24] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[25] Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. Labeled PSI from homomorphic encryption with reduced computation and communication. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1135–1150. ACM Press, November 2021.

[26] Ivan Damgård and Mats Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In Kwangjo Kim, editor, *PKC 2001*, volume 1992 of *LNCS*, pages 119–136. Springer, Heidelberg, February 2001.

[27] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. Pir-psi: Scaling private contact discovery. *Proceedings on Privacy Enhancing Technologies*, 2018(4):159–178, 2018.

[28] Martin Dietzfelbinger and Stefan Walzer. Efficient gauss elimination for near-quadratic matrices with one short random block per row, with applications. In *27th Annual European Symposium on Algorithms (ESA 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[29] Zeev Dvir and Sivakanth Gopi. 2-server PIR with subpolynomial communication. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 577–584. ACM Press, June 2015.

[30] Klim Efremenko. 3-query locally decodable codes of subexponential length. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 39–44. ACM Press, May / June 2009.

[31] Nils Fleischhacker, Kasper Green Larsen, and Mark Simkin. How to compress encrypted data. In *EUROCRYPT 2023, Part I*, pages 551–577. Springer, 2023.

[32] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Heidelberg, February 2005.

[33] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Oblivious key-value stores and amplification for private set intersection. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 395–425, Virtual Event, August 2021. Springer, Heidelberg.

[34] Craig Gentry and Shai Halevi. Compressible FHE with applications to PIR. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part II*, volume 11892 of *LNCS*, pages 438–464. Springer, Heidelberg, December 2019.

[35] Craig Gentry and Zulfikar Ramzan. Single-database private information retrieval with constant communication rate. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *ICALP 2005*, volume 3580 of *LNCS*, pages 803–815. Springer, Heidelberg, July 2005.

[36] Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. Protecting data privacy in private information retrieval schemes. In *30th ACM STOC*, pages 151–160. ACM Press, May 1998.

[37] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 640–658. Springer, Heidelberg, May 2014.

[38] Matthew Green, Watson Ladd, and Ian Miers. A protocol for privately reporting ad impressions at scale. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1591–1601. ACM Press, October 2016.

[39] Jens Groth, Aggelos Kiayias, and Helger Lipmaa. Multi-query computationally-private information retrieval with constant communication rate. In Phong Q. Nguyen and David Pointcheval, editors, *PKC 2010*, volume 6056 of *LNCS*, pages 107–123. Springer, Heidelberg, May 2010.

[40] Syed Mahbub Hafiz and Ryan Henry. A bit more than a bit is more than a bit better: Faster (essentially) optimal-rate many-server PIR. *PoPETs*, 2019(4):112–131, October 2019.

[41] Ryan Henry. Polynomial batch codes for efficient IT-PIR. *PoPETs*, 2016(4):202–218, October 2016.

[42] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. In *USENIX Security 2023*, 2023.

[43] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In László Babai, editor, *36th ACM STOC*, pages 262–271. ACM Press, June 2004.

[44] Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. Mobile private contact discovery at scale. In Nadia Heninger and Patrick Traynor, editors, *USENIX Security 2019*, pages 1447–1464. USENIX Association, August 2019.

[45] Tali Kaufman and Madhu Sudan. Sparse random linear codes are locally decodable and testable. In *48th FOCS*, pages 590–600. IEEE Computer Society Press, October 2007.

[46] Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *38th FOCS*, pages 364–373. IEEE Computer Society Press, October 1997.

[47] Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In *International Conference on Information Security*, pages 314–328. Springer, 2005.

[48] Zeyu Liu and Eran Tromer. Oblivious message retrieval. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 753–783. Springer, Heidelberg, August 2022.

[49] Wouter Lueks and Ian Goldberg. Sublinear scaling for multi-client private information retrieval. In Rainer Böhme and Tatsuaki Okamoto, editors, *FC 2015*, volume 8975 of *LNCS*, pages 168–186. Springer, Heidelberg, January 2015.

[50] Rasoul Akhavan Mahdavi and Florian Kerschbaum. Constant-weight PIR: Single-round keyword PIR via constant-weight equality operators. In *USENIX Security 22*, pages 1723–1740, Boston, MA, 2022.

[51] Samir Jordan Menon and David J Wu. Spiral: Fast, high-rate single-server PIR via FHE composition. In *2022 IEEE Symposium on Security and Privacy*, 2022.

[52] M. Mughees and L. Ren. Vectorized batch private information retrieval. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 437–452, 2023.

[53] Muhammad Haris Mughees, Hao Chen, and Ling Ren. Onion-PIR: Response efficient single-server PIR. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2292–2306. ACM Press, November 2021.

[54] Muhammad Haris Mughees, Gonçalo Pestana, Alex Davidson, and Benjamin Livshits. PrivateFetch: Scalable catalog delivery in privacy-preserving advertising, 2021.

[55] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 223–238. Springer, Heidelberg, May 1999.

[56] Jeongeun Park and Mehdi Tibouchi. SHECS-PIR: Somewhat homomorphic encryption-based compact and scalable private information retrieval. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *ESORICS 2020, Part II*, volume 12309 of *LNCS*, pages 86–106. Springer, Heidelberg, September 2020.

[57] Sarvar Patel, Joon Young Seo, and Kevin Yeo. Don't be dense: Efficient keyword PIR for sparse databases. In *USENIX Security 2023*, 2023.

[58] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.

[59] Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL, January 2023. Microsoft Research, Redmond, WA.

[60] Kevin Yeo. Cuckoo hashing in cryptography: Optimal parameters, robustness and applications. In *CRYPTO 2023*, 2023.

A Random Band Matrix Analysis

In this section, we analyze the variant of random band matrices from Section 3.3 where each column is generated with a random w-bit band. As these are transposes of random band matrices, we know these variants will have unique solutions. It remains to show that the running time of these random band matrices runs in time $O(nw)$ similar to the original random band matrices from [28].

To do this, we show that each row will consist of exactly one continguous section of non-zero entries of length $O(w)$. We couple the process of generating random band matrices with random column vectors as two-dimensional balls-into-bins allocation (see [16] for more details). In particular, we model each of the t columns as t lists of w items. There exists $m = (1+\varepsilon)t$ entries corresponding to each of the m rows. Each of the t lists are assigned to a random entry from $[m-w+1]$. If the i-th list is assigned to entry $j \in [m-w+1]$, then one of the w items in the list are placed into each of the entries $\{j, j+1, \ldots, j+w-1\}$. Note, the maximum load of any of m entries is equivalent to the largest consecutive section of non-zero entries in any of the m rows of the generated random band matrix after sorting by column starting location.

Prior work [16] studied the setting where each of the t lists picked one of the m entries uniformly at random. We adapt the analysis for the slightly skewed distribution used for random band matrices in our work where only one of the first $m-w$ entries are chosen uniformly at random.

Proof of Theorem 1. We use the coupling described above. Therefore, it suffices for us to analyze only two-dimensional balls-into-bins allocations. We denote binary random variables $X_{i,j}$ to be whether the random band of the i-th column will overlap with the j-th row. Therefore, $X_{i,j} = 1$ if this event is true and $X_{i,j} = 0$ otherwise. Note, $X_{i,j} = 1$ if and only if the i-th column's random band starts in the set of row indices $\{j-w+1, j-w+2, \ldots, j\}$. In other words, $\mathsf{E}[X_{i,j} = 1] \leq w/(m-w+1)$. Let B_j be the total number of columns whose random bands overlap with the j-th row. By linearity of expectation, we get that

$$B_j = \sum_{i \in [t]} \mathsf{E}[X_{i,j}] \leq \frac{tw}{m-w+1}.$$

Note that each $X_{i,j}$ is an independent random variable. Therefore, we can apply Chernoff bounds to get that

$$\Pr\left[B_j > 3 \cdot \frac{tw}{m-w+1}\right] \leq 2^{-tw/(m-w+1)}.$$

Next, we apply a Union bound over all m rows to get that B_j for all $j \in [m]$ is upper bounded by the same value with probability at most $m \cdot 2^{-tw/(m-w+1)}$. Finally, by noting that $m = (1+\varepsilon)t$ for some constant $\varepsilon > 0$ and picking $w = O(\lambda + \log t)$, we get that each row has a band length of at most $O(w)$ except with probability $2^{-\lambda}$. □